SIMULINK®

Communications Toolbox

Weizheng Wang

Modeling

Simulation

Implementation

User's Guide

Version 2

How to Contact The MathWorks:

508-647-7000 Phone

508-647-7001 Fax

The MathWorks, Inc. Mail
24 Prime Park Way
Natick, MA 01760-1500

`http://www.mathworks.com`	Web
`ftp.mathworks.com`	Anonymous FTP server
`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
`subscribe@mathworks.com`	Subscribing user registration
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

Communications Toolbox User's Guide

© COPYRIGHT 1996 - 1997 by The MathWorks, Inc. All Rights Reserved.

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.

U.S. GOVERNMENT: If Licensee is acquiring the software on behalf of any unit or agency of the U. S. Government, the following shall apply:

(a) for units of the Department of Defense:

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

(b) for any other unit or agency:

NOTICE - Notwithstanding any other lease or license agreement that may pertain to, or accompany the delivery of, the computer software and accompanying documentation, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Clause 52.227-19(c)(2) of the FAR.

Contractor/manufacturer is The MathWorks Inc., 24 Prime Park Way, Natick, MA 01760-1500.

MATLAB, Simulink, Handle Graphics, and Real-Time Workshop are registered trademarks and Stateflow and Target Language Compiler are trademarks of The MathWorks, Inc.

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

Printing History:	April 1996	First printing
	May 1997	Second printing (for MATLAB 5)

Contents

1 Before You Begin

What Is the Communications Toolbox? **1-2**
Available Functions **1-2**
Related Products and Documentation **1-3**
What is MATLAB? **1-3**
What is the Signal Processing Toolbox? **1-3**
What is Simulink? **1-4**
Recommended Products **1-4**
Installation **1-5**
Using the Toolbox **1-5**
Organization of this Manual **1-6**
Acknowledgments **1-7**

2 Introduction to Communication Systems

The Parts of a Communication System **2-3**
The Communications Toolbox Block Library **2-4**

MATLAB Functions **2-8**

Differences Between Simulink and MATLAB Simulation **2-10**
Data Flow Simulation Example **2-10**
Time Flow Simulation Example **2-12**
Comparison of Time and Data Flow Simulation Schemes . . . **2-14**

3 Tutorial

Signal Generators and Display Devices **3-5**
Random Signal Generators **3-5**
Gaussian Noise Generator **3-5**
Rayleigh Noise Generator **3-6**
Rician Noise Generator **3-6**
Poisson Random Integer Generator **3-7**
Uniform Noise Generator **3-7**
Display Devices **3-7**
Eye-Pattern Diagrams **3-8**
Scatter Plots **3-11**
Error-Rate Analysis **3-12**

Source Coding **3-13**
Scalar Quantization **3-13**
Training of Partition and Codebook Parameters **3-16**
Companders **3-18**
μ-law Compander **3-18**
A-law Compander **3-19**
Predictive Quantization **3-19**
Differential Pulse Code Modulation **3-20**

Error-Control Coding **3-23**
Vector vs. Sequential Input/Output **3-25**
Vector Input/Output Format **3-26**
Sequential Input/Output Format **3-27**
Error-Control Coding Library Blocks **3-29**
Linear Block Codes **3-30**
Example of a Linear Block Code **3-32**
Hamming Codes **3-32**
Cyclic Codes **3-33**
BCH Codes **3-33**
BCH Encoding **3-34**
BCH Decoding **3-36**
Reed-Solomon Codes **3-36**

Convolutional Codes **3-37**
The Transfer Functions of Convolutional Coding **3-38**
Convolutional Encoding **3-44**
Convolutional Decoding **3-44**
Example of Convolutional Decoding **3-48**
Modulation and Demodulation **3-52**
Passband Analog Modulation and Demodulation **3-53**
Example of the Effect of Lowpass Filters on Demodulation **3-54**
Double-Sideband Suppressed Carrier Amplitude
Modulation and Demodulation **3-56**
Double-Sideband with Transmission Carrier Amplitude
Modulation and Demodulation **3-57**
Single-Sideband Suppressed Carrier Amplitude
Modulation and Demodulation **3-60**
Quadrature Amplitude Modulation and Demodulation . . . **3-62**
Frequency Modulation and Demodulation **3-65**
Phase Modulation and Demodulation **3-66**
Passband Digital Modulation and Demodulation **3-67**
Types of Digital Modulation/Demodulation **3-69**
M-ary Amplitude Shift-Keying Modulation **3-71**
M-ary Quadrature Shift-Keying Modulation **3-73**
Square Constellation **3-74**
Circle Constellation **3-75**
Arbitrary Constellation **3-76**
M-ary Frequency Shift-Keying Modulation **3-80**
Coherent Demodulation **3-81**
Noncoherent Demodulation **3-82**
Simulink Example **3-84**
MATLAB Example **3-86**
M-ary Phase Shift-Keying Modulation **3-87**
Baseband Modulation and Demodulation **3-89**
Baseband Analog Modulation **3-90**
Baseband Analog Demodulation **3-92**
Coherent Demodulation **3-92**
Noncoherent Demodulation **3-93**
Baseband Digital Modulation **3-93**
Baseband Digital Demodulation **3-97**
Using the Baseband Simulation Functionality
in the Toolbox **3-97**
Baseband Simulation Example **3-99**

Signal Multiple Access **3-104**
Time-Share Multiplexing and Demultiplexing **3-104**
Vector Signal Redistribution **3-106**
Time Division Multiple Access (TDMA) **3-107**
Digital TDMA **3-107**
Frequency Division Multiple Access (FDMA) **3-108**
Code Division Multiple Access (CDMA) **3-110**

Transmitting and Receiving Filters **3-113**
Raised Cosine Filter **3-113**
Raised Cosine Filter Example **3-116**
Sinc Filter **3-118**
Hilbert Transform Filter **3-118**

Channels **3-120**
Passband Channels **3-120**
Baseband Channels **3-121**
Rayleigh Noise Channel **3-121**
Rician Noise Channel **3-121**
Fading Channel **3-121**

Synchronization **3-123**
Passband Signal PLL Simulation **3-123**
Baseband Signal PLL Simulation **3-125**

Utilities **3-126**
Voltage Controlled Oscillator **3-127**
Modulo Integrator **3-127**
Windowed Integrator **3-128**
Scheduled Reset Integrator **3-128**
Minimum/Maximum Index **3-128**
Array Functions **3-128**
Scalar to Vector Converter **3-128**
Vector to Scalar Converter **3-129**
Modulo Operation **3-129**
Mean and Variance **3-129**

Galois Field Calculations **3-130**
Galois Field, GF(q) **3-130**

Polynomials over GF(q) 3-131
GF(q^M), the GF(q) Field Extension 3-132
Addition in GF(q^M) 3-136
Multiplication in GF(q^M) 3-136
The Cyclotomic Cosets of GF(q^M) 3-137
Minimal Polynomials over GF(q^M) 3-137
The Roots of Polynomials over GF(q^M) 3-138
Other GF(q) Related Functions 3-138

4 Examples

Bit-Error Rate Computation **4-3**
Background 4-3
Open the COMMGUI Window 4-3

Digital Subscriber Telephone Lines (DSL) **4-7**
Channel and Noise Models 4-8
The Transceiver Architecture 4-9
Performance Simulation 4-10
The Receiver 4-12
References 4-14

Early-Late Gate Synchronization **4-15**
Background 4-15
Communication System Simulation 4-18
Signal Simulation 4-19
Communications Channel 4-20
Signal Demodulation 4-21

V.34 Modem Data Transmission **4-29**
Background 4-29
Simulation Block Diagram 4-30
Parameters and Limitations of the Simulation Design 4-33

Call Modem Simulation Implementation 4-36
Scramble, Parser, and Shell Mapping Subsystem 4-37
Differential Encoder Subsystem 4-37
Rotation Subsystem 4-38
Precoder Subsystem 4-39
Nonlinear Encoder Subsystem 4-39
Synchronization Pulses Subsystem 4-39
Trellis Code Subsystem 4-40
Answer Modem Simulation Implementation 4-44

5 MATLAB Function Reference

Formats and Conventions **5-3**

Reference Tables **5-4**

6 Simulink Block Library Reference

Sources and Sinks in Communication Systems **6-4**
Sources and Sinks Reference Table 6-6

Source Coding **6-38**
Source Coding Reference Table 6-39

Error-Control Coding **6-53**
Error-Control Coding Reference Table 6-54

Modulation and Demodulation **6-110**
Analog Modulation/Demodulation Methods
for Passband Simulation 6-111
Analog Modulation/Demodulation
Passband Reference Table 6-114

Analog Modulation/Demodulation
Methods for Baseband Simulation **6-134**
Analog Modulation/Demodulation
Baseband Reference Table **6-136**
Digital Modulation/Demodulation
Methods for Passband Simulation **6-153**
Digital Modulation/Demodulation
Passband Reference Table **6-155**
Digital Modulation/Demodulation
Methods for Baseband Simulation **6-183**
Digital Modulation/Demodulation
Baseband Reference Table **6-185**
Mapping/Demapping for Digital Modulation/Demodulation . **6-207**
Passband and Baseband Digital
Mapping/Demapping Reference Table **6-211**

Signal Multiple Access **6-235**
Multiple Access Reference Table **6-237**

Transmitting and Receiving Filters **6-249**
Transmitting/Receiving Filters Reference Table **6-250**

Channels **6-261**
Channel Reference Table **6-262**

Synchronization **6-275**
Synchronization Reference Table **6-276**

Utilities **6-285**
Utility Reference Table **6-287**

A RTW and Communications

Template Makefiles **A-3**
Updating the Template Makefile for the Toolbox **A-3**

Example: TCM Simulation **A-5**

RTW Guidelines **A-9**
Writing MEX-File S-Functions for RTW **A-9**
Block Names and Code Generation **A-10**

Limitations And Notes **A-11**

B **Bibliography**

1

Before You Begin

What Is the Communications Toolbox? 1-2

Related Products and Documentation 1-3

Installation 1-5

Using the Toolbox 1-5

Organization of this Manual 1-6

Acknowledgments 1-7

This chapter provides a brief overview of the Communications Toolbox. It explains how to use this guide and points you to additional manuals for information on installation, the working environment, and related products.

What Is the Communications Toolbox?

The Communications Toolbox is a collection of computation functions and simulation blocks for research, development, system design, analysis, and simulation in the communications area. The toolbox is designed to be suitable for both experts and beginners in the communications field. The toolbox contains ready-to-use functions and blocks, which you can easily modify to implement your own schemes, methods, and algorithms. The Communications Toolbox is designed for use with MATLAB® and Simulink®.

The Communications Toolbox includes:

- Simulink blocks
- MATLAB functions

You can use the toolbox directly from the MATLAB workspace. You can use the Simulink environment to construct a simulation block diagram for your communication system. For convenience, the Communications Toolbox provides online interactive examples that use many of the functions found in this toolbox.

Available Functions

This toolbox supports a variety of functions in the communications area. These functions include:

- Data source
- Source coding/decoding
- Error-control coding
- Modulation/demodulation
- Transmission/reception filters
- Transmitting channel
- Multiple access
- Synchronization
- Utilities

Related Products and Documentation

The Communications Toolbox requires:

- MATLAB
- The Signal Processing Toolbox

What is MATLAB?

MATLAB is a powerful collection of tools for algorithm development, computation, and visualization. It provides more control and flexibility compared to a traditional high-level programming language. Unlike such languages, MATLAB is compact and easy to learn, letting you express algorithms in concise, readable code. In addition, MATLAB provides an extensive set of ready-to-use functions including mathematical and matrix operations, graphics, color and sound control, and low-level file I/O. MATLAB is readily extensible — you can use the MATLAB language to easily create functions that operate as part of the MATLAB environment.

Note: The Communications Toolbox requires version 4.2c or later of MATLAB.

The *Using MATLAB* guide describes how to work with the MATLAB language. It discusses how to enter and manipulate data and use MATLAB's extensive collection of functions. It explains how to perform command-line computations and how to create your own functions and scripts. The online MATLAB Function Reference provides reference descriptions of supplied MATLAB functions and commands.

What is the Signal Processing Toolbox?

The Signal Processing Toolbox is a collection of tools built on the MATLAB numeric computing environment. The toolbox supports a wide range of signal processing operations, from waveform generation to filter design and implementation, parametric modeling, and spectral analysis.

The *Signal Processing Toolbox User's Guide* describes the toolbox in detail. It discusses how to use the toolbox functions to address a variety of signal processing tasks, and provides tutorial information, examples, and individual function reference pages.

Note: The Communications Toolbox requires version 3.0b or later of the Signal Processing Toolbox.

What is Simulink?

Simulink is a dynamic system simulation environment. It allows you to represent systems as block diagrams, which you build using your mouse to connect blocks and your keyboard to edit block parameters. The Communications Toolbox, if used in conjunction with Simulink, is part of this environment — many blocks are actually masked Simulink models.

The *Using Simulink* guide describes how to work with Simulink. It explains how to manipulate Simulink blocks, access block parameters, and connect blocks to build models. It also provides reference descriptions of each block in the standard Simulink libraries.

Note: The Communications Toolbox requires version 1.3c or later of Simulink if you want to run the toolbox in the Simulink environment.

Recommended Products

The Communications Toolbox takes full advantage of the functionality and structure of Simulink. Although the Communications Toolbox does not require Simulink to run, using Simulink with the toolbox is highly recommended for system simulation.

There are other optional MathWorks products that you can use with the Communications Toolbox. While the Communications Toolbox provides some support for channel noise generators, you can find a broader selection of noise generators in the Statistics Toolbox. The Real-Time Workshop™ is an optional software component that you can use to generate code from Simulink models. The DSP Blockset is a collection of block libraries designed for use with Simulink. These block libraries are designed specifically for digital signal processing applications, such as convolution and Fourier transforms. All these products are compatible with the Communications Toolbox.

Installation

The Communications Toolbox follows the same installation procedures as the MATLAB toolboxes.

- To install this toolbox on a workstation, see the *Installation Guide for UNIX*.
- To install the toolbox on a PC or Macintosh, refer to the *Installation Guide for PC and Macintosh*.

To determine if the Communications Toolbox is already installed on your system, check for subdirectories named `comm`, `commsim`, and `commfun` under the `comm` subdirectory within the main toolbox directory or folder.

Using the Toolbox

If you are new to the communications field, read through the rest of this chapter, the *Introduction to Communication Systems* chapter, and the *Tutorial* chapter. These two chapters provide the necessary information for you to understand the key techniques in the communications field.

If you are an experienced communications user, you need to read through the rest of this section. You may want to skip the *Introduction to Communication Systems* chapter. You can refer back to the *Tutorial* and *Examples* chapters for further details about any specific topic.

If you just want to start as soon as possible, read through the rest of this section. You may also want to read through the "Differences between Simulink and MATLAB Simulation" section in Chapter 2.

All toolbox users can learn to use the toolbox by using the examples provided in the Simulink block library. Type

```
commlib
```

in the MATLAB workspace.

Double-clicking any block in the `commu` window brings up a function block sublibrary or a more detailed category sublibrary. Except for self-explanatory utility functions, every function block in this toolbox has a companion example that shows how to use the block. All function blocks are black with a white background. All example blocks are black with a cyan background. By double-clicking a Simulink block diagram, you can begin its simulation by choosing **Start** in the **Simulation** menu.

The Communications Toolbox supports an online interactive example that you can access by typing

```
commgui
```

in the MATLAB workspace. A description of how to use this example is provided in the “MATLAB Functions” section in Chapter 2.

To view all MATLAB functions in the toolbox, type

```
help comm
```

for a list of all toolbox commands.

The Communications Toolbox is divided into MATLAB functions and Simulink block libraries. The subdirectory `comm` contains the MATLAB functions. The subdirectory `commsim` contains the Simulink block library.

`comm` – MATLAB functions

`commsim` – Simulink block library

In addition, there is a `commsfun` sublibrary, which includes all S-function files.

If you use the MATLAB functions, see the online MATLAB Function Reference for information on specific MATLAB functions. The function reference descriptions include a synopsis of the function’s syntax, as well as a complete explanation of options and operation.

If you use the Simulink block library, you should read the *Using Simulink* guide for information on specific function blocks. The block reference descriptions include the parameter settings of the blocks as well as a complete explanation of options and operation.

Organization of this Manual

Chapter 2 provides a brief overview of communication systems.

Chapter 3 provides a thorough description of the various communication techniques implemented in the Communications Toolbox. The discussion is based on interaction with the functions and blocks provided in this toolbox. This chapter does not discuss theoretical issues. You can use this chapter to scan quickly through each area in the communication systems field. For your convenience, Appendix B, *Bibliography*, lists standard reference books in the communications field.

Chapter 4 provides simulation examples for communication systems. Some results of the examples are shown using figures created by MATLAB Handle Graphics®. A number of the examples are based on practical communication systems.

Chapter 5 contains detailed descriptions of all MATLAB M-file functions in the Communications Toolbox. The chapter lists the functions alphabetically.

Chapter 6 contains the Simulink block library. Each block in the Simulink block library is discussed in detail. The Simulink library is divided into several sublibraries based on functionality. Refer to the beginning of chapter 6 for a discussion of that chapter's organization.

Appendix A discusses using the Real-Time Workshop with the Communications Toolbox, and Appendix B contains a bibliography.

Acknowledgments

The author would like to acknowledge the following people:

Darrel Judd, Harman Motive, reviewed the software and documentation, and provided the early-late gate synchronization example in Chapter 4.

Walter Chen, Texas Instruments, provided the HDSL example in Chapter 4.

Irvine Reed, USC, provided constructive comments and suggestions during the early phase of development.

Eric Yang, AT&T, provided expertise in the communications field.

Richard Townsend, GTE, reviewed the software and documentation.

Bill Balint helped write the documentation. Donna Sullivan and Peg Theriault edited the documentation and provided printing support. The cover was designed by Ken Hyman. Jun Wu converted the error-control coding functions to C for Real-Time Workshop compatibility. He also conducted tests and assisted in documentation. Ken Karnofsky and Yama Habibzai organized the beta tests. Andy Grace reviewed the software and the documentation.

2

Introduction to Communication Systems

The Parts of a Communication System 2-3
The Communications Toolbox Block Library 2-4

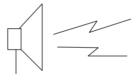
MATLAB Functions 2-8

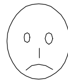
Differences Between Simulink and MATLAB Simulation 2-10
Data Flow Simulation Example 2-10
Time Flow Simulation Example 2-12
Comparison of Time and Data Flow Simulation Schemes . . 2-14

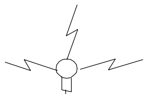
"Communication" involves the transfer of information from one place to another. Communication can take many forms; talking is a common way for people to communicate between each other. However, one person's voice is not loud enough to be audible over a long distance. So people must rely on other communication systems to transfer information over long distances. For example, Bostonians are used to getting weather information by looking at the lights on the top of the old John Hancock building since people in the Boston area know the lights mean:

Steady blue – Clear view
Flashing blue – Clouds due
Steady red – Rain ahead
Flashing red – Snow instead

(Flashing red in the summer means that the Red Sox baseball team is playing in town that night.)







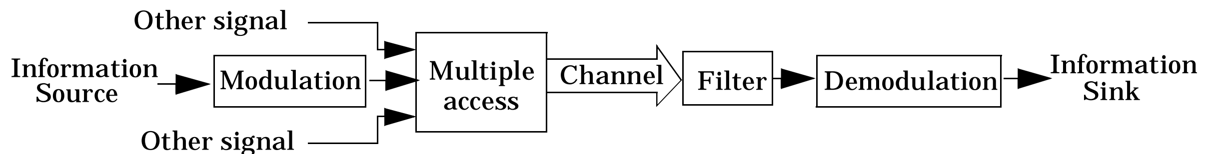
In most cases, light travels longer distances than sound. Using this principle, engineers developed various ways to transfer messages. One of the most popular methods is to use radio frequency signals to transfer human voice signals over a long distance. The process of using a radio frequency signal (a higher frequency signal) to carry a human voice (a lower frequency signal) is known as *modulation*. The receiver uses a demodulation method to recover the received modulated signal. There are different methods for signal modulation/ demodulation. The demodulation method must match the modulation method to recover the transmitted signal correctly.

The Parts of a Communication System

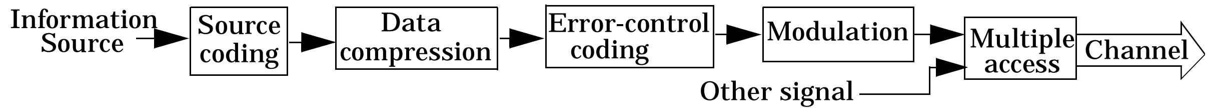
A simple communication system contains three parts: the transmitter, the channel (the transmitting medium), and the receiver. The information source and information sink are also considered part of the communication system.

The task of communication system engineers is to design and maintain a communication system such that the receivers can correctly demodulate, decode, and correct errors in the transmitted message.

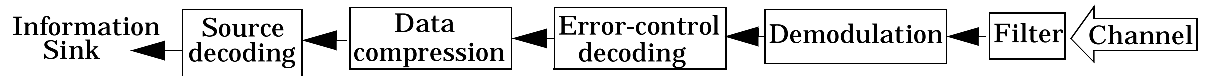
In general, a communication system can be classified as either analog or digital. A transmitter for an analog communication system may include signal modulation and multiple access. *Multiple access* is a “sharing” technique whereby signals are sent in turn. A receiver for an analog communication system may include a signal filtering bank and signal demodulation.



The messages in a digital communication system are represented by using digital numbers or bits.A transmitter for a digital communication system may include source coding, data compression, error-control coding, digital modulation, and multiple access. *Source coding* is a process using a digital signal to represent an analog signal. *Data compression* is a process that converts the message into a compressed message, meaning a message that has fewer information bits than the original message. The receiving side can recover the original message from the compressed message by a data decompression process. *Error-control coding* adds redundant error-checking and/or error-correction bits. The receiving side can use the extra bits to detect and/or correct the transmission errors. *Digital modulation* maps the digital signal to an analog signal and then uses a higher frequency signal to carry the analog signal. This figure shows the parts of the transmission process:



A receiver is conceptually the inverse of the transmitter. In most cases, the receiving process method should match exactly the transmitting process method to recover the transmitted message. This figure shows the parts of the receiving process:.



The channels in communication systems may vary in different applications. In general, a channel may introduce noise, fading, phase shifting, and interference to the transmitted signal.

The techniques used in the digital communications field are also used in other applications. For instance, source coding, data compression, and error-control coding techniques are widely used in computer storage applications.

Except for data compression, this toolbox provides ready-to-use tools to design, analyze, and simulate communication systems. These systems include both analog and digital communication systems.

The Communications Toolbox Block Library

To use the block library in the Communications Toolbox, you must have Simulink installed. To open the block diagram of the Simulink block library, type the command

```
commlib
```

in the MATLAB workspace.

The block diagram of the top level of the Simulink library shows the communication system structure:

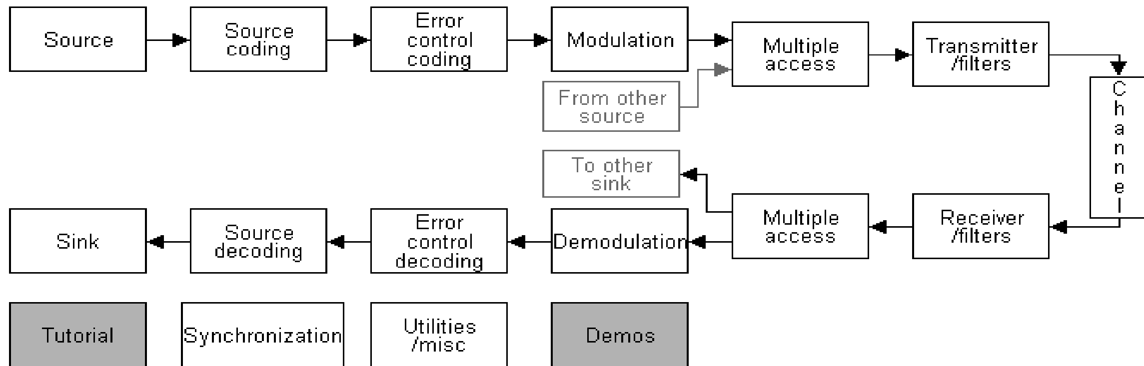


Figure 2-1: The Communications Toolbox Simulink Block Library

The upper row of blocks is the transmitting part of a communication system. The bottom row contains the receiving part of a communication system. The receiving part is a mirror image of the transmitting part. In most — but not all — cases, the receiving computation is exactly the reverse of the transmitting computation. Below the communication system diagram are utilities and synchronization sublibraries that play supporting roles in the transmitter-channel-receiver sequence.

Double-clicking any block in this library opens a sublibrary. The same block library opens whenever you double-click a transmitting block or its mirror-image receiving block. For example, double-clicking the Error control coding or Error control decoding blocks opens up the same sublibrary.

Two sublibraries contain further subcategories. These are:

- Error-control code/decode
- Modulation/demodulation

Double-click any subcategory block to reach the function blocks.

A sublibrary usually contains two different block types: function blocks and demo blocks. The demo blocks are cyan colored (or gray on black and white screens). You can use the function blocks to build the block diagram of your own communication system. Double-clicking a function block opens the parameter dialog box to set the parameters for that block. In some sublibraries, there are lines to link the transmitting blocks and receiving blocks, which means that they are direct pairs of the transmitting and the receiving process. In general, the blocks must be used together in a communication system. Double-clicking a demo block opens a block diagram of a simple system which illustrates the use of blocks located on the left of the demo block in the sublibrary.

The figure on the next page shows an example of the hierarchical structure of the Communications Toolbox Simulink Block Library. Double-clicking a block opens a block diagram in the next level.

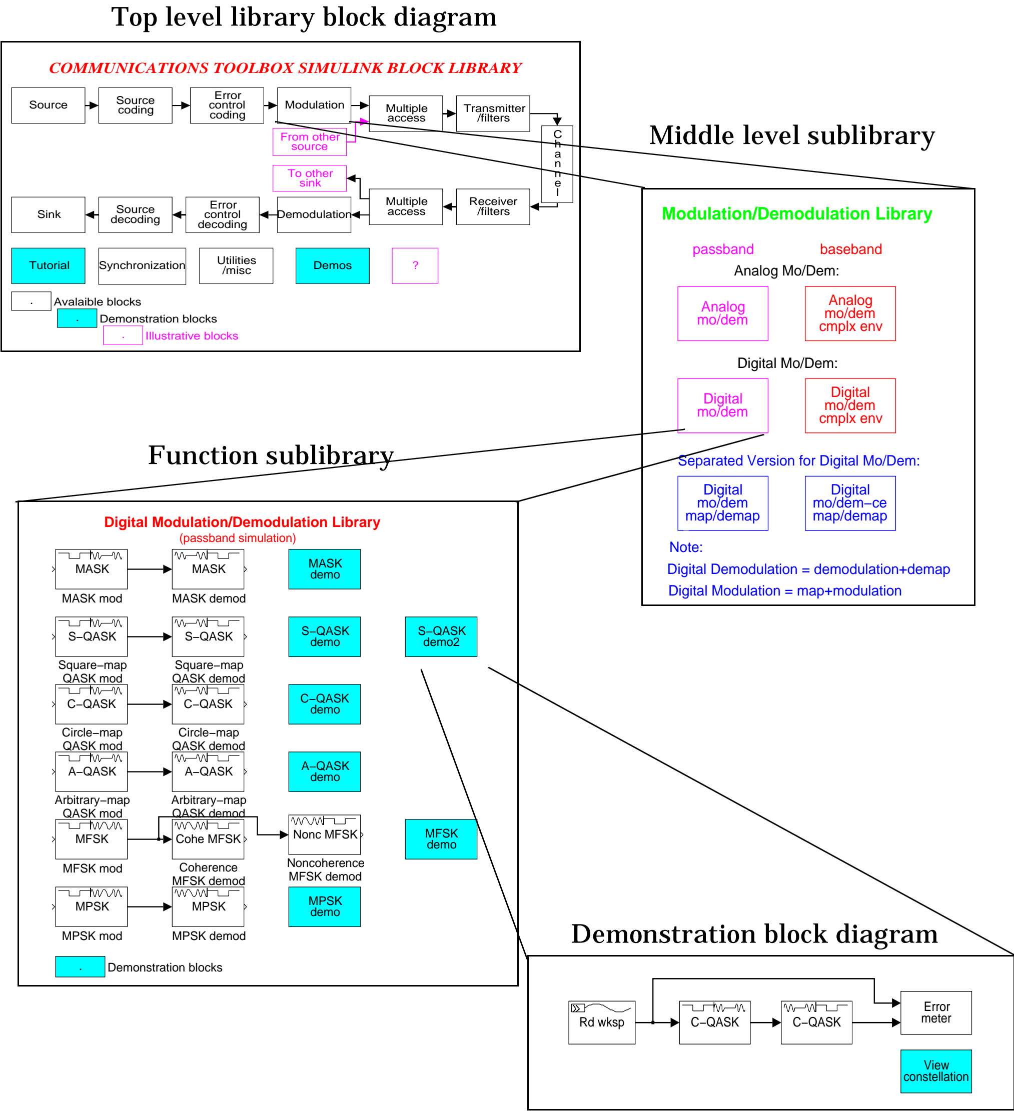


Figure 2-2: The Hierarchical Structure of the Communications Toolbox Simulink Block Library

MATLAB Functions

The Communication Toolbox includes the most commonly used functions for communication system design, analysis, and simulation. The MATLAB functions are in subdirectory `comm`. You can get a list of the contents by typing

```
help comm
```

or

```
hthelp commhelp
```

Often there are various ways to implement a given concept in this toolbox. For instance, there are six different analog modulation techniques supported in the Communications Toolbox. You can find general information for using the analog modulation function by typing

```
help amod
```

on the command line. The help utility lists a set of 'method strings' that describe each of the modulation techniques. For help on how to use a specific modulation method, type `amod` followed by the method string. For instance, to find out about the QAM method, type:

```
amod qam
```

Note that this command produces the help file for the QAM method even though the word ‘help’ doesn’t appear on the command line. This table lists all the functions that use this particular help command:

Function	Description
amod	Passband analog modulation. Methods of choice: amdsb-sc, amdsb-sc, amssb, qam, fm, pm.
ademod	Passband analog demodulation.
amodce	Baseband analog modulation. Methods of choice: amdsb-sc, amdsb-sc, amssb, qam, fm, pm.
ademodce	Baseband analog demodulation.
dmod	Passband digital modulation. Methods of choice: ask, psk, qask, fsk, msk.
ddemod	Passband digital demodulation.
dmodce	Baseband digital modulation. Methods of choice: ask, psk, qask, fsk, msk.
ddemodce	Baseband digital demodulation.
modmap	Digital signal to analog signal mapping. Methods of choice: ask, psk, qask, fsk, msk.
demodmap	Analog signal to digital signal mapping.
encode	Error-control encoding. Methods of choice: hamming, block, cyclic, bch, rs, convolution.
decode	Error-control decoding.

Differences Between Simulink and MATLAB Simulation

There are differences between simulation with Simulink block diagrams and MATLAB functions. In Simulink simulation, each block in the diagram advances at each time step. In other words, all the Simulink blocks are performed concurrently during each time step; this is called time flow simulation. In MATLAB simulation, the functions sequentially follow the data flow. This means that the processed data goes through one computation stage before activating the next stage. This is known as data flow simulation. Specific applications may require one scheme over the other.

Data Flow Simulation Example

Let's use an example to explain the data flow simulation concept. The example shows a 16-QASK modulation with BCH error-control coding. The simulation uses the MATLAB functions `encode` and `decode` for error-control coding and decoding. It uses the MATLAB functions `dmod` and `ddemod` for digital modulation and demodulation. The message source used in the simulation is a sequence of random integers. In the command lines, the lines starting with '%' are comment lines:

```
% Define the multiple number, codeword length, and message
% length.
M = 16; N = 15; K = 11;


% Define the carrier frequency, digital transfer frequency,
% and simulation frequency.
Fc = 10000; Fd = 1000; Fs = 100000;

% Generate a 1100-by-1 matrix that contains the digital message
% to be transferred. The multiple number is M.
% randint generates random int.
msg = randint(K*100,1);

% Use BCH code. The codeword length is N
% and the message length is K.
code = encode(msg,N,K,'bch');
```

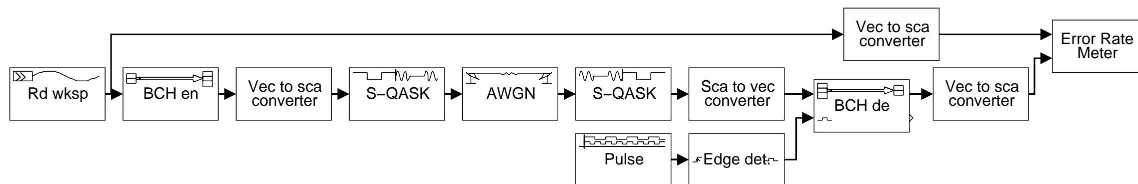
```
% Use a 16-QASK digital modulation method.
% The multiple number is M=16.
modu = dmod(code,Fc,Fd,Fs,'qask',M);

% Assume the STD of the added Gaussian noise is 0.1.
std_value = 0.1;

% Add AWGN with STD equal to std_value.
modu_noise = modu+randn(length(modu),1)*std_value;

% Demodulation process
demo = ddemod(modu_noise,Fc,Fd,Fs,'qask', M);

% BCH decode
% The codeword length is N and the message length is K.
msg_r = decode(demo,N,K,'bch');
rate = biterr(msg,msg_r,M)
```

In the simulation, the entire data vector is processed as a unit. The computation process generates a 1100-by-1 random integer vector. The message signal is encoded by BCH encoding functions. The encoded signal is sent to the next procedure, the QASK modulation, and so on. The modulation process does not begin until the error control encode is finished. The data process of the above MATLAB commands follows the arrows indicated in this block diagram:

Once the simulation finishes, you can use MATLAB to post-process the data. The last line of the above MATLAB commands compares the original data with the recovered data from the receiving side. In general, when using data flow simulation, you have to wait until the end of the simulation to see the overall results and to compare the results to previous runs. It is not possible to reset parameters in the middle of a simulation. If you want to change a parameter in the MATLAB code, you must rerun the simulation after making the change.

Time Flow Simulation Example

Using the Simulink block library, you can build a Simulink block diagram that closely follows the parts of the communications process described earlier in this chapter. This figure shows a Simulink block diagram that performs the same functions as the "Data Flow Simulation Example":



The long sequence of blocks in the center row of the figure are the function blocks. The Vector to scalar converter, Vector pulse, and Edge detector blocks are I/O and timing blocks that support the basic communications functions in the figure. Like the block diagram introduced in the "Data Flow Simulation Example," this block diagram introduces the Signal generator (Sampled read variable block), BCH encode, QASK modulation, Channel, QASK demodulation, and BCH decode blocks. The Error Rate Meter block compares the original message to the recovered signal on the receiving side. It corresponds to the last line of the MATLAB code in the "Data Flow Simulation Example."

Unlike the MATLAB simulation, Simulink recovers and displays the received message bits at each simulation time step. This figure shows the error rate comparison map generated by the Error Rate Meter block:

Sender	Receiver
0	0
15	15
0	0
14	14
1	1
3	3
5	5
7	7
9	9
11	11
13	13
15	15
6	6
10	10
11	11
10	10
4	4
13	13
3	3
13	13
Symbol Transfered	43
Error Number	0
Error Rate	0
Bit Transfered	172
Error Number	0
Error Rate	0

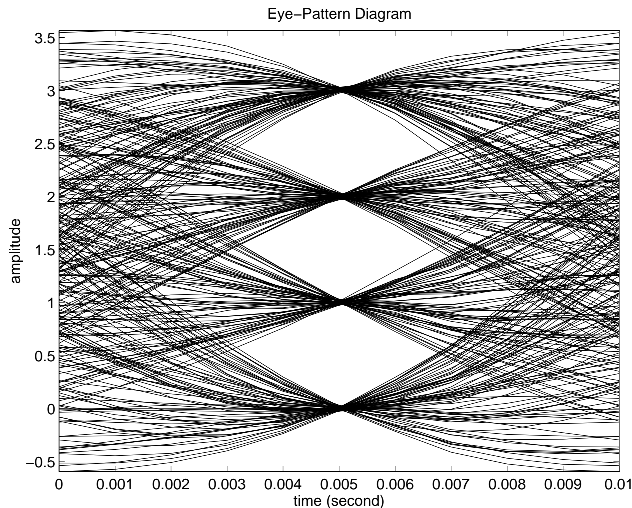
Message symbols

Recovered symbols

Symbol error

Bit error

In the simulation the timing for the demapping can be chosen by tuning the pulse generator. The process for determining the optimal demapping rate involves using an *eye-pattern plot*, which is a multiple overlaid trace plot of the received signal. The decision is determined by finding where the "eye" is most widely open:



The decision point in this eye-pattern plot is at t=0.005. This means that the trigger signal for the QASK demodulation block should be set to 1/0.005 = 200 Hertz.

The Simulink simulation is an interactive simulation. This means that you can reset most of the parameters in a block during the simulation (not all parameters can be reset during the simulation). You can use the display blocks to show the simulation results while the simulation is still running. You can also save the data into files or into MATLAB workspace variables and do post-processing on the saved data using MATLAB functions.

Comparison of Time and Data Flow Simulation Schemes

Data flow and time flow simulations result in the same solutions (within computational tolerances). Since data and time flow simulation schemes handle simulation time differently, one method may be more suitable than the other in a given application. In the examples described in the last two

subsections, the data flow and time flow computations can be viewed as processes with different timing. This figure is a qualitative comparison of the timing used in the two schemes:

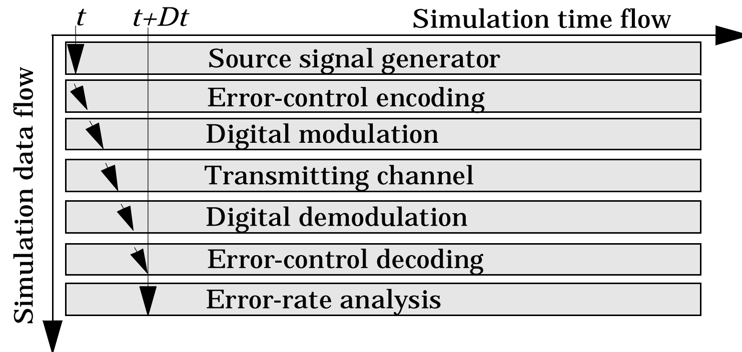


Figure 2-3: Comparison of Time and Data Flow Simulations

The arrows inside the process category blocks indicate the simulation delay in the process. If the arrow is strictly vertical, there is no delay in the process. If the arrow is not strictly vertical, there is a delay in the process. Because of cumulative delays, the data sent at time t from the data source is fully recovered at time $t+Dt$, where Δt is the *transmission delay*. The transmission delay can be modeled in both data flow and time flow simulation. Using the Simulink user interface in the time flow simulation, you can adjust the estimated time delay in the communication system by changing the offset of the trigger pulse signals. The time delay is an important factor in the design of a communication system as well as in the communication simulation. From the synchronization point of view, the time flow simulation is more favored than the data flow simulation.

In data flow simulation, you can easily design a filter in the domain or perform a spectrum analysis at any stage in the simulation. For instance, you can implement a perfect cut-off frequency filter. The implementation of a perfect cut-off frequency filter is almost impossible when working in the time domain. Since a data flow simulation uses batch file processing, you can perform a spectral analysis on the entire data set and shape the data by using the desired filters. This is useful for theoretical research. In a practical communication system, the signals are processed in real time. To use a special filter, you can either implement the required filter as a time domain filter, or hold the

transfer data in a buffer, processing the buffered data using frequency analysis. This figure illustrates the buffering-data procedure:

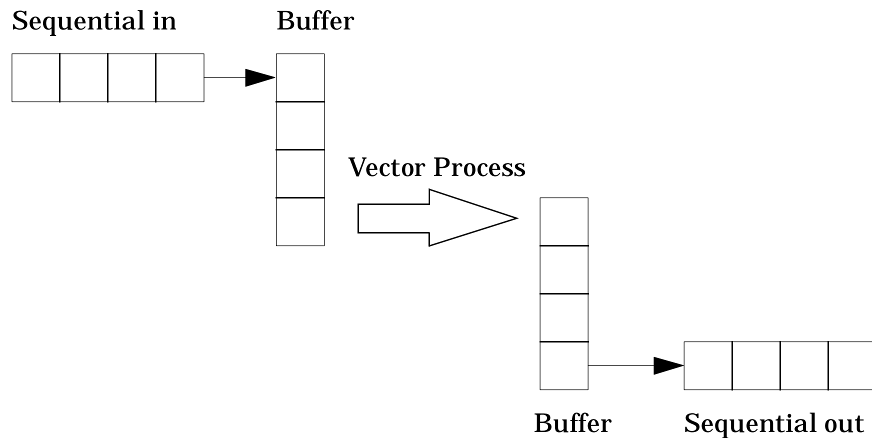


Figure 2-4: Data Buffering

Frequency filtering can be done at the "vector process" stage in the figure. Note that the buffer process introduces a longer time delay compared to the time domain filter. It also introduces more computational complexity. The longer the buffer length, the more accurate the process, but the cost is a longer delay.

In the extreme case, when the buffer size is the same as the entire data vector size, the time flow simulation is equivalent to the data flow simulation. In this case, *Dt* equals the entire length of the process.

3

Tutorial

Signal Generators and Display Devices 3-5

Source Coding 3-13

Error-Control Coding 3-23

Signal Multiple Access 3-104

Transmitting and Receiving Filters 3-113

Channels 3-120

Synchronization 3-123

Utilities 3-126

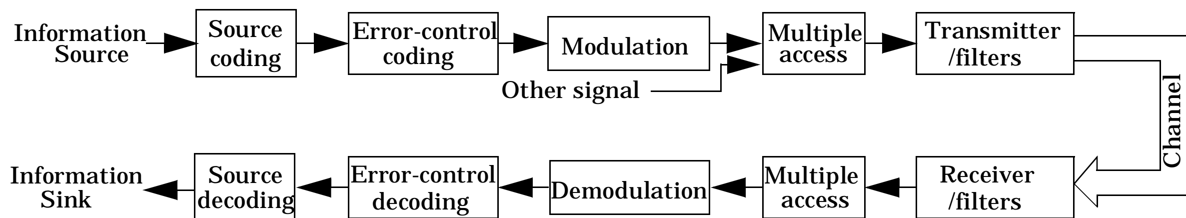
Galois Field Calculations 3-130

This chapter provides an introduction to the techniques implemented in the Communications Toolbox. This chapter also reviews communication systems concepts. It does not contain theoretical discussions. See Appendix B at the end of this *User's Guide* for references to more detailed discussions of communications theory. If you are a beginner in the communication field, you can learn communications techniques while using the toolbox. If you are an expert, you can modify the functions and the blocks for your own applications.

This chapter assumes that you know basic communications terminology, such as the distinction between bits and symbols. If you don't know these distinctions, refer to a general reference in communications theory, such as Lee & Messerschmidt or Proakis.

This chapter examines communication techniques using the M-files and Simulink blocks provided in the toolbox. For descriptions of individual Simulink blocks and MATLAB functions, read Chapter 5, *MATLAB Function Reference*, and Chapter 6, *Simulink Block Library Reference*.

A typical communication system includes a signal source, sink, and channel as well as processes for transmitting and receiving. Except for the channel, these concepts divide into pairs. The sections in this chapter follow the transmitting/ receiving diagram flow as shown in the figure:



This toolbox contains a Simulink block sublibrary and a MATLAB function category for each item in the figure. In addition, the toolbox also contains various synchronization and utilities functions and blocks. The table below lists the basic processes involved in communications and the functions associated with each process. The MATLAB functions are listed in parentheses. Each of these processes is discussed in this chapter:

Process	Associated Functions
Signal Generator and Display Devices	Random signal generator Display devices (eyescat) Error-rate analysis (symerr/biterr)
Source Coding	Scalar quantization (quantize/lloyd) A-law compander (compander) μ-law compander (compander) Predictive quantization (dpcmenco/dpcmdeco/dpcmopt)
Error-Control Coding	Linear block code Hamming code (encode/decode for all coding methods) Cyclic code BCH code Reed-Solomon code Convolutional code
Modulation and Demodulation	Passband analog (amod/ademod) Baseband analog (amodce/ademodce) Passband digital (dmod/ddemod) Baseband digital (dmodce/ddemodce)

Process	Associated Functions
Signal Multiplex and Demultiplex (Multiple Access)	Time division Frequency division Code division
Transmitting and Receiving Filters	Raised cosine filter Hilbert transform filter Complex parameter filter
Channels	Additive white Gaussian noise Rayleigh fading channel Rayleigh noise channel
Utilities/ Miscellaneous	Synchronization – phase-locked loop Format conversion Voltage-controlled oscillator Galois field computations (MATLAB functions only)

Signal Generators and Display Devices

This toolbox includes a number of signal generator and signal display blocks and functions. The Source sublibrary contains the signal generator and signal display blocks. The signal generators introduced in this section, all of which generate random signals, are used in the transmitting channel. The display devices introduced, including eye-pattern diagrams and scatter plots, are among the most commonly used techniques in error analysis of communication systems.

Random Signal Generators

This toolbox provides Gaussian, Rayleigh, Rician, Poisson, and uniform noise generators in the Simulink block library.

You can find the MATLAB functions for generating the random variables in the Statistics Toolbox.

Gaussian Noise Generator

Gaussian noise is the most widely used noise model in communication system analysis. The probability density function (pdf) of the n-dimensional Gaussian noise is

$$f(x) = \frac{1}{(2\pi)^{\frac{n}{2}}\sqrt{\det(K)}}\exp\left(-\frac{1}{2}(x-\mu)^T K^{-1}(x-\mu)\right)$$

where x is a length n vector, K is the n-by-n covariance matrix associated with x, and μ is the mean value vector of x. Gaussian noise is used in the most popular channel model in communication systems, the AWGN (additive white Gaussian noise) channel.

You can find the Gaussian Noise Generator block in the Source sublibrary and the AWGN block in the Channel sublibrary. `randn`, which generates normally distributed numbers, is available as a built-in MATLAB function.

Rayleigh Noise Generator

The Rayleigh Noise Generator block generates signals that are Rayleigh distributed. Rayleigh noise is noise with Rayleigh distributed amplitude. A Rayleigh distribution is equivalent to that of the root sum square (RSS) of a two-dimensional, zero mean Gaussian variable. Let y_1 and y_2 be the two independent Gaussian random variables with zero mean and a common variance σ^2. The Rayleigh random variable x is $x = \sqrt{y_1^2 + y_2^2}$. The pdf of the Rayleigh noise is:

$$f(x) = \frac{x}{\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right)$$

The mean value of this distribution is $\sigma\sqrt{\pi/2}$ and its variance is $(2-\pi/2)\sigma^2$. The Rayleigh noise is frequently used in radio channels, such as cellular radio channels.

Rician Noise Generator

The Rician Noise Generator block generates signals that follow a Rician distribution. Let y_1 and y_2 be two independent Gaussian random variables with means m_1 and m_2 and a common variance σ^2. The quantity $v2$, called the *noncentrality parameter*, equals $m_1^2+m_2^2$. The pdf of the Rician noise is:

$$f(x) = \begin{cases} \dfrac{x}{\sigma^2} I_0\left(\dfrac{xv}{\sigma^2}\right) e^{-\frac{x^2+v^2}{2\sigma^2}} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where function $I0(u)$ is the modified zeroth order Bessel function of the first kind given by:

$$I_0(u) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{u\cos t}\,dt$$

The mean value of Rician noise is (2+m) and the variance is 4(1+2m).

Poisson Random Integer Generator

The Poisson Random Integer Generator block generates integers that follow a Poisson distribution. This toolbox uses the Poisson random integer generator for error-control code testing. The pdf of the Poison distribution is:

$$f(x) = \begin{cases} \frac{\lambda^x}{x!}e^{-\lambda} & \text{when x=0,1,2,...} \\ 0 & \text{otherwise} \end{cases}$$

The mean and variance of this distribution are both equal to λ. When λ is less than 1, the probability of the nonzero data generated by the Poisson random number generator is approximately equal to λ. The binary error channel in this toolbox uses this property to generate binary noise signals. You can find the Poisson Random Integer Generator block in the Source sublibrary.

Uniform Noise Generator

The Uniform Noise Generator block, which generates uniformly distributed noise signals, is commonly used in the estimation of source coding distortion. For $b_{up} > b_{low}$, the uniformly distributed random variable x satisfying the distribution is:

$$f(x) = \begin{cases} \frac{1}{b_{up} - b_{low}} & \text{when } b_{low} \le x \le b_{up} \\ 0 & \text{otherwise} \end{cases}$$

The mean value of this distribution is $(1/2)*(b_{up}+b_{low})$ and the variance is $(1/12)*(b_{up}+b_{low})^2$. You can find the Uniform Noise Generator block in the Source sublibrary `rand`, which generates uniformly distributed numbers, is available as a built-in MATLAB function.

Display Devices

The Communications Toolbox provides display devices that make it easier to analyze the performance of a communication system. Included in display devices are the eye-pattern diagram, scatter plot, and error-rate computation. This toolbox supports both Simulink blocks and MATLAB functions that produce these display devices.

Eye-Pattern Diagrams

In communications, it is often useful to quantify the degradation of a transmission signal. Eye-pattern diagrams, so-called because they are eye-like in appearance, are a widely used graphical illustration of this degradation. A common way to generate eye-pattern diagrams is to set an oscilloscope to sweep a signal at the rate of 1/T, where T is the symbol period. The continuous overlay of sweeps produces the eye-pattern plot.

Digital transmission simulation involves two different sampling frequencies: the decision frequency, F_d, at which symbol value decisions are made, and the simulation sample frequency, F_s, where $F_s > F_d$. Let x be the signal at the demodulator output. It is sampled once during each symbol period, and the value of that sample is used to determine which symbol was most likely transmitted. There are F_s/F_d data points in the $1/F_d$ symbol period. The one which is used to make the decision regarding the estimated value of the transmitted symbol is called the *decision point*. Note that in this toolbox the symbol decision is called *demapping*.

In message transmission, a number of factors may distort the received signal, such as the transmitting/receiving filters, channel noise, channel interference, and the filters used in the demodulation. A communications engineer must develop an algorithm for deciding what message was actually sent. You can use an eye-pattern diagram to determine the decision point, which makes it possible to decide what message was most likely sent. Usually, the decision point is the point where the "eye" is most widely opened. This is the point at which the signal is least sensitive to phase variations and timing jitter.

This toolbox provides the Eye-Pattern Diagram block in the Simulink library. There is also a MATLAB function `eyescat` for plotting eye-pattern diagrams. You can also use the digital demodulation functions, such as `dmod`, `dmodce`, and `demodmap`, to plot eye-pattern diagrams.

Eye-pattern diagrams are also useful in evaluating the effect of intersymbol interference (ISI).

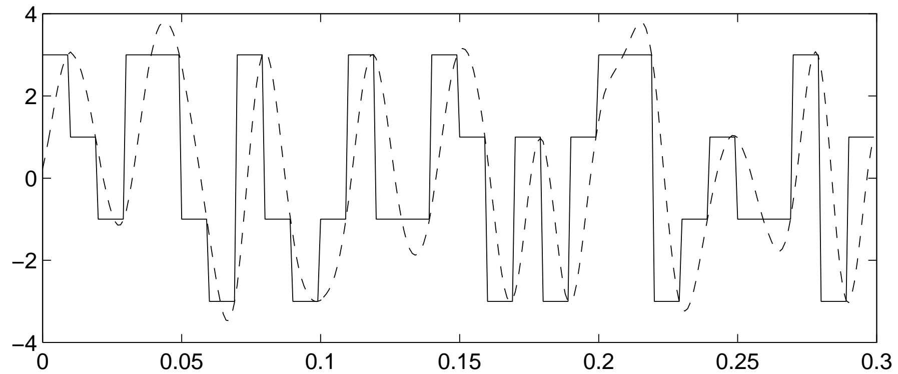
Eye-Pattern Diagram Example . This example demonstrates the use of the eye-pattern plot in the digital demodulation for digital transmission. Given that the *M-ary number* (the number of different symbols than can be transmitted) of the signal source is 16 and that the digital modulation method is QASK, these command lines demonstrate the use of the eye-pattern plot:

```
% Define the M-ary number, calculation sample frequency.
M = 16; Fs = 10;
% Define the number of points in the calculation.
Pd = 100;
% The default symbol rate is Fd = 1.
% Generate an integer message in range [0,M-1].
msg_d = randint(Pd,1,M);
% Use square constellation QASK method for modulation.
msg_a = modmap(msg_d,1,Fs,'qask',M);
% Assume the channel equivalent to a raised cosine filter.
rcv_a = rcosflt(msg_a,1,Fs,'Fs');
% Compare the signal before and after raised cosine filter.
plot([0:1/Fs:Pd-1/Fs],msg_a,[0:1/Fs:Pd-1/Fs],rcv_a)
% Plot the eye-pattern diagram of the received signal.
eyescat(rcv_a,1,Fs,0,10);
```

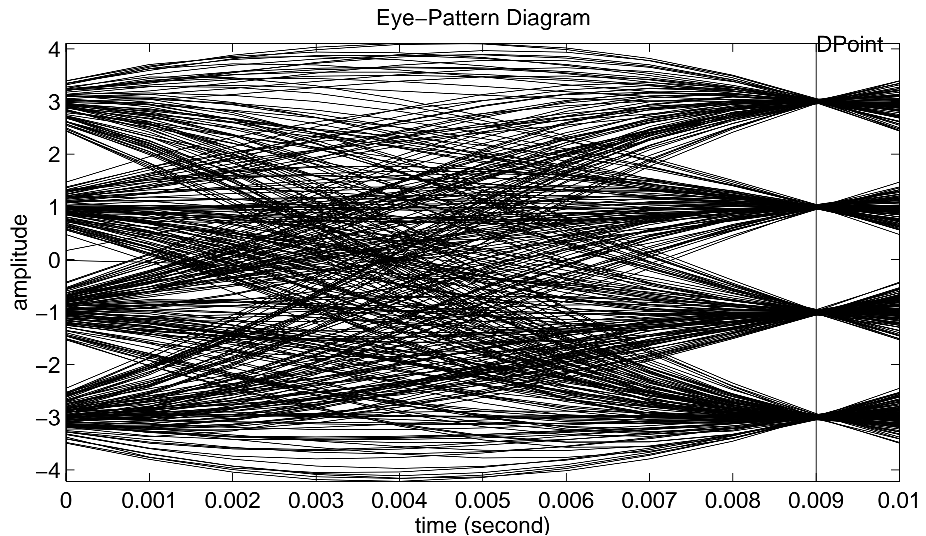
You must tune the fifth parameter in the above `eyescat` command to make the D-point line (the line that locates the decision point) in the eye-pattern diagram cross over the “eye” at the eye’s most widely opened point.

```
% Use the offset information for digital demapping.
rcv_d = demodmap(rcv_a,1,[10,10],'qask',16);
% Symbolic error analysis
symerr(msg_d,rcv_d)
ans=
     0
```

The figure generated by the `plot` command is a comparison between the transmitted message and the received signal. In the figure, the solid lines are the transmitted digital signal; the dotted curves are the received signal:



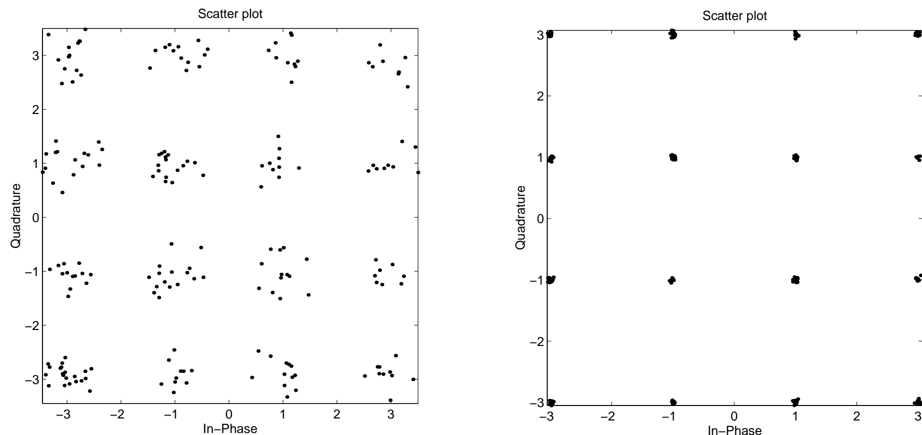
The eye-pattern diagram repeatedly overlays the received signal over the digital transmission period. The figure is generated by the command `eyescat`. It is clear that the eye is widely opened at the D-point line.



Scatter Plots

The scatter plot is closely related to the eye-pattern diagram plot. The scatter plot records the received data value at the decision point. In the MATLAB function library, you can use the function `eyescat` to generate scatter plots. In Simulink, you can use the Scatter Plot block.

The plots below compare two scatter plots that were generated using two different offset points in the computation. The first, which is shown in the left-hand side of the figure, has shifted one data point from the optimal decision point. The second, which is shown at the right-hand side of the figure, has the decision point exactly at the optimal point:



The scatter plot for a two-column vector is a two-dimensional plot with the *x*-axis equalling the in-phase component and the *y*-axis equalling the quadrature component.

The commands used in generating the figure are:

```
eyescat(rcv_a, Fd, Fs, 9, '.');
eyescat(rcv_a, Fd, Fs, 10, '.');
```

Refer to the Eye-Pattern and Scatter Plot blocks in the Simulink reference pages for more information on how to generate eye-pattern and scatter plot diagrams.

Error-Rate Analysis

Error-rate analysis is frequently used to evaluate the quality of a communication system design or the performance of a special technique or algorithm. The *number of error bits* is the number of bit errors resulting from comparing the message sent to the message received. The *bit error rate* is the number of bit errors divided by the total number of bits transmitted. The *number of symbol errors* and the *symbol error rate* are defined similarly on the transmitted symbol level.

In the MATLAB function library, there are two functions, `symerr` and `biterr`, dedicated to symbol-error number and error-rate computation. In the Simulink block library, there is an Error-Rate Meter block to display the symbol to symbol comparison between the sender and the receiver(s) and to compute the symbol error number, symbol error rate, bit error number, and bit error rate. A snapshot of the Error-Rate Meter is shown in the figure below:

Sender	Receiver1	Receiver2
0	3	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	10	2
7	7	7
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	7	6
7	7	7
Symbol Transfered	2428	
Error Number	209	25
Error Rate	0.086079077	0.01029654
Bit Transfered	7284	
Error Number	318	30
Error Rate	0.043657331	0.0041186161

Errors in transmission

Symbol to symbol comparison between sender and receivers

Number of symbol errors and error rate

Number of bit errors and error rate

As indicated in the figure, after 7284 bits transmitted, the block detects 318 error bits in receiver 1. The error rate is:

318/7284 = 0.043657

The block detects 30 error bits in receiver 2; the error rate is 0.004118.

Source Coding

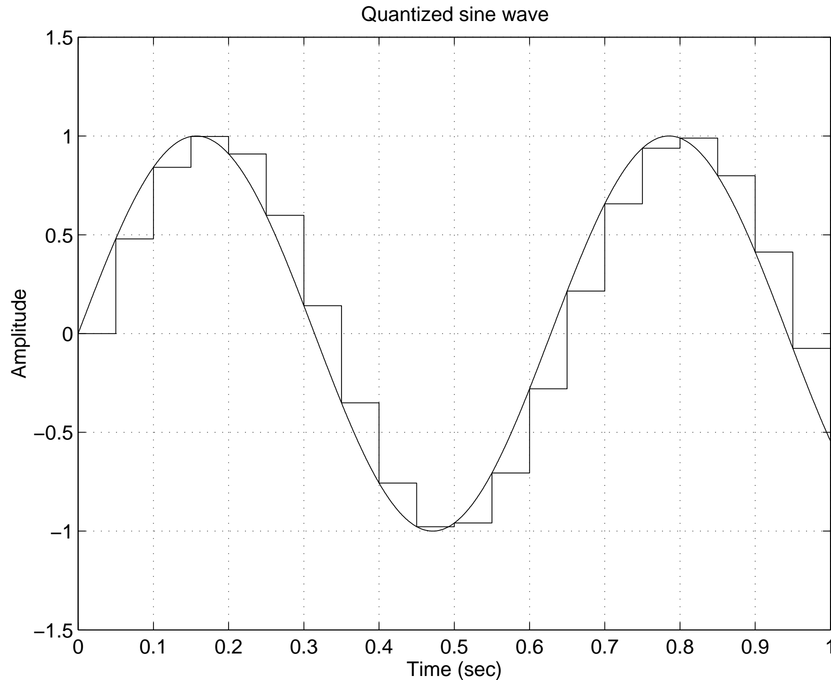
The Communications Toolbox includes some basic functions for source coding. Source coding, also known as quantization or signal formatting, includes the concepts of analog-to-digital conversion and data compression.

Source coding divides into two basic procedures: source encoding and source decoding. Source encoding converts a source signal into a digital code using a quantization method. The source coded signal is represented by a set of integers {0, 1, 2, ..., N-1}, where N is finite. Source decoding recovers the original information signal sequence using the source coded signal. This toolbox includes two source coding quantization methods: scalar quantization and predictive quantization. A third source coding method, vector quantization, is not included in this toolbox.

Scalar Quantization

Scalar quantization is a process that assigns a single value to inputs that are within a specified range. Inputs that fall in a different range of values are assigned a different single value. An analog signal is in effect digitized by

scalar quantization. For example, a sine wave, when quantized, will look like a rising and falling stair step:



Scalar quantization requires the use of a mapping of N contiguous regions of the signal range into N discrete values. The N regions are defined by a partitioning that consists of N-1 distinction partition values within the signal range. The partition values are arranged in ascending order and assigned indices ranging from 1 to N-1. Each region has an index that is determined by this formula:

$$indx(x) = \begin{cases} 0 & x \le \text{partition}(1) \\ i & \text{partition}(i) < x \le \text{partition}(i+1) \\ N-1 & \text{partition}(N-1) < x \end{cases}$$

For a signal value *x*, the index of the corresponding region is indx(x).

To implement scalar quantization you must specify a length N-1 vector *partition* and a length N vector *codebook*. The vector partition, as its name

implies, divides the signal input range into N regions using N-1 partition values. The partition values must be in strictly ascending order. The codebook is a vector that assigns a value, typically either an endpoint of the region or some average value of the interval, to each region defined in the partition. Since each region must have an assigned output value, the length of the codebook must equal the length of the partition. Another way to view this is that the codebook functions as a table lookup with each element assigned to a partition.

The index value *indx(x)* is the output of the quantization encode function. The codebook contains the values that correspond to sample points. There is no function for quantization decoding, which is simply constructing the quantized signal using the stream of index values output by the quantizer. Construct the quantized signal by using the MATLAB command:

```
y = codebook(indx+1);
```

In general, a codebook has the following relation with the vector partition:

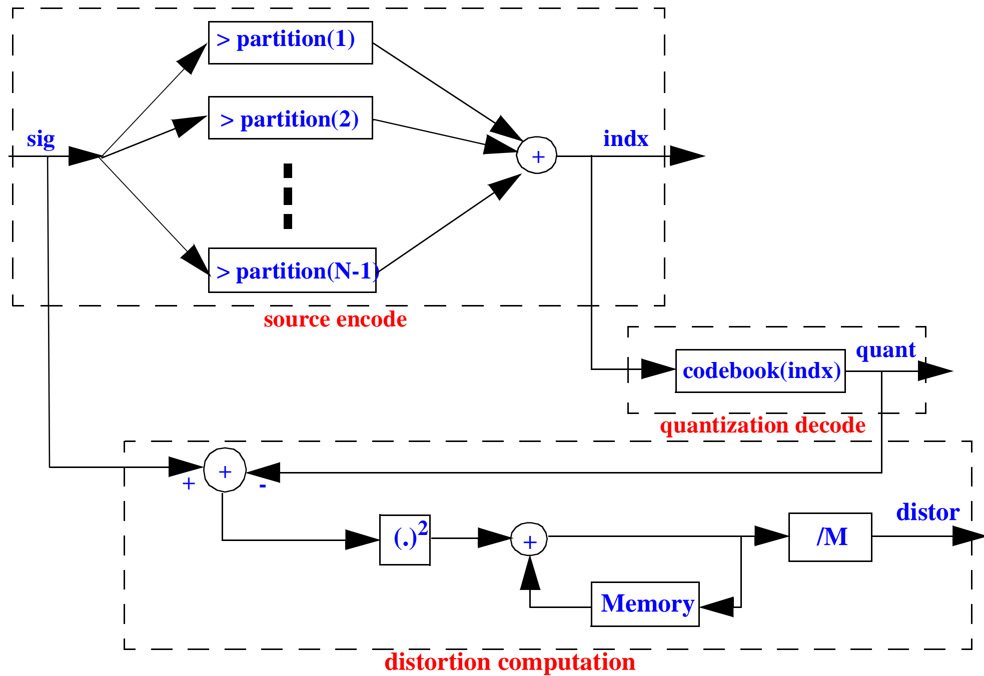
$$\text{codebook}(1) \le \text{partition}(1) \le \text{codebook}(2) \le \text{partition}(2) \le \ldots$$
$$\ldots \le \text{codebook}(N-1) \le \text{partition}(N-1) \le \text{codebook}(N)$$

The quality of the quantization, called the distortion, is the mean-square error between the original signal data `sig` and the quantized signal `quan`:

$$\text{distortion} = \frac{1}{M}\sum_{i=1}^{M}(\text{sig}(i) - \text{quan}(i))^2$$

where M is the number of samples of the source signal `sig`.

The computation procedure for the quantization source coding and decoding is shown in the figure below:



The dashed square at the top of this figure is the source encode algorithm, which assigns an index after deciding in which region the input signal value falls. The dashed square in the middle of the figure is the quantization decode algorithm, which maps the input index to whatever value the codebook assigns to that particular index. The dashed square at the bottom of the figure is the distortion computation, which calculates a cumulative average in which M is the total number of points used in the computation.

The MATLAB function `quantiz` computes all three outputs shown in the above figure. The Simulink Scalar Quantizer block is available in the source code sublibrary.

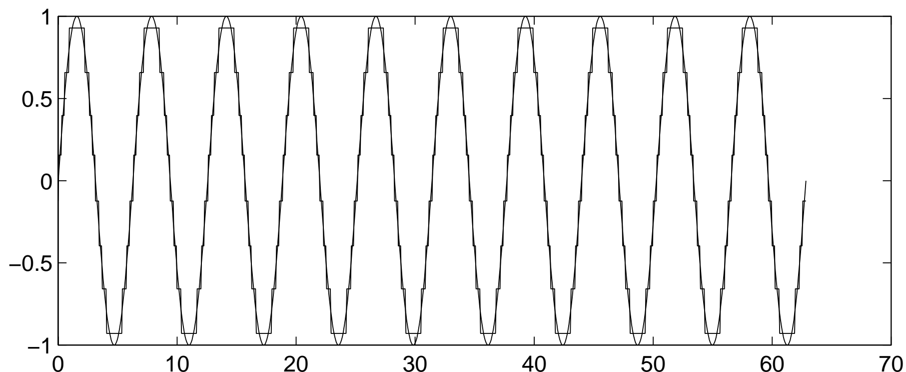
Training of Partition and Codebook Parameters

The key functions in quantization are the assignment of the partition and codebook parameters. In large signal sets with a fine quantization scheme, the

selection of all the correct parameters can be tedious. In the Communications Toolbox you can train these two parameters by using the MATLAB function `lloyds`. To train the parameters, you must prepare a training set, which typically represents function input data. The function `lloyds` finds the partition and codebook parameter vectors by minimizing the distortion using the provided training data. Here is an example of the data training for a sinusoidal signal:

```
N = 2^3; % three bits transfer channel
t = [0:1000]*pi/50;
sig = sin(t); % one complete period of sinusoidal signal
[partition,codebook] = lloyds(sig,N);
[indx,quant,distor] = quantiz(sig,partition,codebook);
plot(t,sig,t,quant,'--');
```

In the above commands, `sig` is a sinusoidal signal to be quantized. The peak amplitude of the input signal must be one. The trained codebook and the partition can be used for sinusoidal signals of any frequency. The above code generates a figure that compares the original signal (the smooth curve) to the quantized signal (the digital curve):



The decoding procedure is simple using the basic MATLAB computation format. Use the following command to obtain the decoded result:

```
quant = codebook(indx+1)
```

Companders

The quantization discussed above is linear. In certain applications, you may need to quantize a signal based on the power level of the input signal. In this case, it is common to use a logarithm computation before the quantization operation. Since a simple logarithm computation can only handle a positive signal, some modification of the input signal is needed. The logarithm computation is known as a *compressor*. The reverse computation of a compressor is called an *expander*. The combination of a compressor and expander is called a compander (compress and expand). This toolbox supports two companders: the μ-law and A-law companders. The selection of either method is a matter of user preference.

The MATLAB function `compand` is designed for compander computation. The Simulink block library includes four blocks for the μ-law and A-law compander computations: μ-Law Compressor, μ-Law Expander, A-Law Compressor, and A-Law Expander.

μ-law Compander

For a given signal x, the output y of the μ-law compressor is

$$y = \frac{V \log(1 + \mu|x|/V)}{\log(1 + \mu)} \operatorname{sgn}(x)$$

where V is the peak value of signal x, which is also the peak value of y. μ is the μ-law parameter of the compander. The function `log` is the natural logarithm and `sgn` is the sign function.

The μ-law expander is the inverse of the compressor:

$$x = \frac{V}{\mu}(e^{|y|\log(1 + \mu)/V} + 1) \operatorname{sgn}(y)$$

The MATLAB function for μ-law companding is `compand`. The corresponding Simulink μ-Law Compressor and μ-Law Expander blocks also support μ-law companding.

A-law Compander

For a given signal x, the output y of the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\operatorname{sgn}(x) & \text{for } 0 \le |x| \le A/V \\ \dfrac{V(1+\log A|x|/V)}{1+\log A}\operatorname{sgn}(x) & \text{for } A/V < |x| \le V \end{cases}$$

where V is the peak value of signal x, which is also the peak value of y. A is the A-law parameter of the compander. The function `log` is the natural logarithm and `sgn` is the sign function.

The A-law expander is the inverse of the compressor:

$$x = \begin{cases} |y|\dfrac{1+\log A}{A}\operatorname{sgn}(y) & \text{for } 0 \le |y| \le \dfrac{V}{1+\log A} \\ e^{|y|(1+\log A)/V-1}\dfrac{V}{A}\operatorname{sgn}(y) & \text{for } \dfrac{V}{1+\log A} < |y| \le V \end{cases}$$

The MATLAB function for A-law companding is `compand`. The corresponding Simulink A-Law Compressor and A-Law Expander blocks also support A-law companding.

Predictive Quantization

The quantization introduced in the “Scalar Quantization” section is usually implemented when there is no *a priori* knowledge about the transmitted signal. In practice, a communications engineer often has some *a priori* information about the message signals. An engineer can use this information to predict the next signal to be transmitted based on past signal transmissions; i.e., he or she can use the past data set $x=\{x(k-m),\ldots,x(k-2), x(k-1)\}$ to predict $x(k)$ by using some function f(.). The most common way to implement predictive quantization is to use the differential pulse code modulation (DPCM) method. The Communications Toolbox provides the tools necessary to implement a DPCM predictive quantizer.

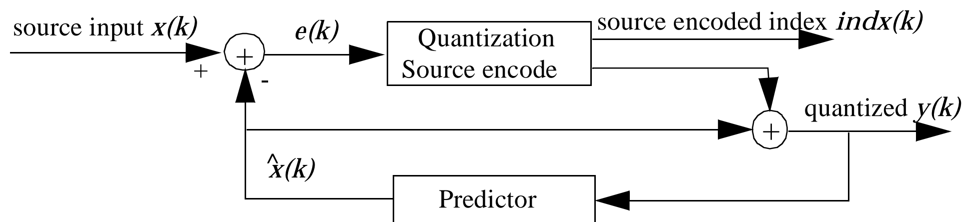
Differential Pulse Code Modulation

Using the past data set and predictor as described above, the predicted value is assumed to be

$$\hat{x}(k) = f(x(k-m), \ldots, x(k-2), x(k-1))$$

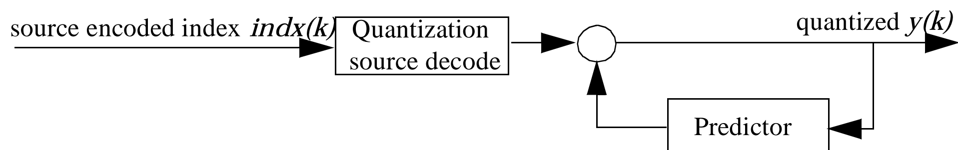
where k is the computation step index. The function f(.) is called the *predictor*; the integer m is the predictive order. The predictive error $e(k) = x(k) - \hat{x}(k)$ is quantized by using the method discussed in the "Scalar Quantization" section.

The structure of predictive quantization is:



This method is known as the differential pulse code modulation method (DPCM). In the figure, indx(k) is the source encoded index, and y(k) is the quantized output. The DPCM method transfers the bit length reduced indx(k) instead of the real data x(k). At the receiving side, a quantization decoder recovers the quantized y(k) from indx(k).

The figure below shows the quantization source decoding method:



The predictor must be the same one used in the encoding figure.

This toolbox uses a linear predictor:

$$\hat{x}(k) = p(1)x(k-1) + \ldots + p(m-1)x(k-m+1) + p(m)x(k-m)$$

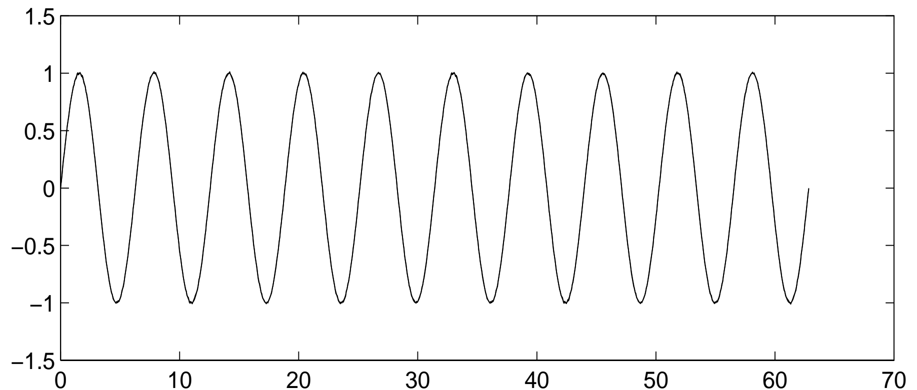
The transfer function of this predictor is represented by a polynomial. The vector p_trans $= \left[0, p(1) \; \dots \; p(k-m+1), p(k-m)\right]$ represents the finite impulse response (FIR) transfer function:

A special case of the DPCM source code method is the widely used delta-modulation method, in which the linear predictor is a first order predictor with

The Communications Toolbox provides MATLAB functions dpcmenco and dpcmdeco for the source encoding and source decoding using the DPCM method. This toolbox also provides the function dpcmopt, which uses a set of training data to generate an optimal transfer function of the predictor p_trans, the partition, and the codebook. The training data represents the input signal used in the DPCM quantization. For example, you can use dpcmopt to find the parameters needed to encode/decode a sinusoidal signal using the delta-modulation method. This example is a continuation of the example provided in the “Scalar Quantization” section:

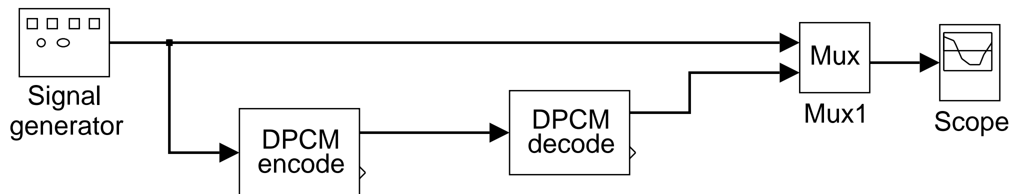
```
% Generate the optimal predictive transfer function,
% the partition, and the codebook.
[p_trans,partition,codebook] = dpcmopt(sig,1,N);
% Encode the signal using DPCM.
indx = dpcmenco(sig,codebook,partition,p_trans);
% Decode using DPCM.
quant = dpcmdeco(indx,codebook,p_trans);
% Compare the original and the quantized signal.
plot(t,sig,t,quant,'--')
```

Note that the sample time is important in the DPCM quantization. The figure below shows the plot generated from the code:



The predictor must be the same one used in the encoding figure. Comparing the result generated in this example to the one generated by scalar quantization, notice that the DPCM quantization is of much better quality. The distortion here is 6.2158e-5, which is much lower than the distortion value of 5.002e-3 achieved by the scalar quantization. Both methods used three bit symbols in the quantization.

Simulink blocks for the DPCM encode and decode are available in the source code sublibrary. A simple block diagram example of using DPCM encode and decode blocks for source coding is:



This block diagram encodes a generated signal using the DPCM method and then recovers the signal by DPCM decoding. The scope in the block diagram compares the original signal with the quantized signal. The curve displayed on the scope is the same as the curve shown in the plot generated from the MATLAB code.

Error-Control Coding

Error-control coding techniques are used to detect and/or correct errors that occur in the message transmission in a digital communication system. The transmitting side of the error-control coding adds redundant bits or symbols to the original information signal sequence. The receiving side of the error-control coding uses these redundant bits or symbols to detect and/or correct the errors that occurred during transmission. The transmission coding process is known as *encoding*, and the receiving coding process is known as *decoding*.

There are two major classes in error-control code: block and convolutional. In block coding, successive blocks of K information (message) symbols are formed. The coding algorithm then transforms each block into a codeword consisting of N symbols where N>K. This structure is called an *(N, K) code*. The ratio K/N is called the *code rate*. A key point is that each codeword is formed independently from other codewords.

Convolutional codes differ from block codes in that there are no independent codewords. The encoding process can be envisioned as a sliding window, M symbols wide, which moves over the sequence of information symbols in steps of K symbols. M is called the *constraint length* of the code and K is usually equal to 1. With each step of the sliding window, the encoding process generates N symbols based on the M symbols visible in the window. The code rate is K/N.

There are two main types of decoding used with error-control codes: algebraic and probabilistic. Algebraic decoding is based on the mathematical properties of linear algebra and Galois (finite) fields. It is usually performed on block codes. There is a set of MATLAB functions for Galois field calculations in this toolbox. The most common type of probabilistic decoding is Viterbi decoding, which is almost always performed on convolutional codes. This toolbox provides an easy-to-use user interface that enables you to use Simulink block diagrams to construct a convolutional code transfer function. The decoding of convolutional codes in this toolbox uses the Viterbi algorithm.

An error-control code is a *linear code* if the transmitted signals are a linear function of the information symbols. The code is called systematic code if the information symbols are transmitted without being altered. Most block codes are systematic, whereas most convolutional codes are nonsystematic. Almost all codes used for error control are linear. The symbols in a code can be either binary or nonbinary. Binary symbols are the familiar 0 and 1. Nonbinary codes

are usually elements of a Galois field GF(p^M). There are p^M possible values for these symbols where p is a prime number and M is a positive integer. The value of p is usually 2. Reed-Solomon codes are the most popular nonbinary codes.

You can execute all encode computations by calling the function encode. The format for encode is

```
code = encode(msg, N, K, method, opt);
```

where msg is the message signal (or information signal sequence), N is the codeword length, and K is the message length (or number of information bits). method is a string that specifies what type of encoding process to use, and opt is an optional parameter used in some of the methods. The table below lists the method strings, their meaning, and the corresponding opt parameter:

Method	Meaning	opt Parameter
hamming	Hamming code	Not used
block	Linear block code	Generator matrix
cyclic	Cyclic code	Generator polynomial
bch	BCH code	Not used
rs	Reed-Solomon code	Not used
convolution	Convolutional code	Octal form transfer function

We encourage you to understand the error-control coding techniques. However, you do not need to be an expert in the coding methods to set the parameters. The rest of this section provides a brief discussion and a few examples that explain how to use the error-control coding functionality of this toolbox.

This example uses a codeword length 15 and 11 information bits to encode a binary message. The example uses BCH code for the encoding process:

```
N = 15;
K = 11;
row_num = 100;
msg = randint(K*row_num, 1, 2);
code = encode(msg, N, K, 'bch');
```

Decoding is the inverse of the encoding process. Use the MATLAB function decode for all decoding processes. The format for function decode is:

```
msg = decode(code, N, K, method, opt1, opt2);
```

The decode optional parameters vary depending what coding method you use. You must use the same method in the encode and decode functions to recover the information signal sequence. Continuing the encoding example, assume there is some transmission error in the channel. As a test, randomly add one bit error to each codeword:

```
nois = randbit(row_num, N, .3);
code = rem(code(:)+nois(:), 2);
```

Decode the message by typing:

```
rcv = decode(code, N, K, 'bch')
```

The bit error in the decode is:

```
err = biterr(rcv, msg)
err =
      0
```

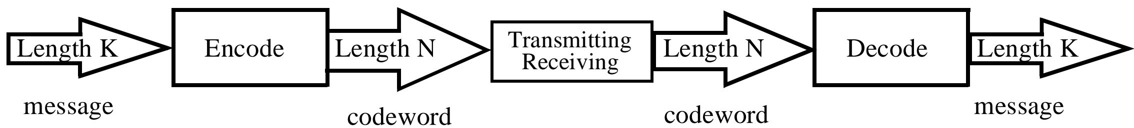
There is a MATLAB block sublibrary containing functions dedicated to the error-control coding/decoding process. These functions are also available to the Simulink block library.

Vector vs. Sequential Input/Output

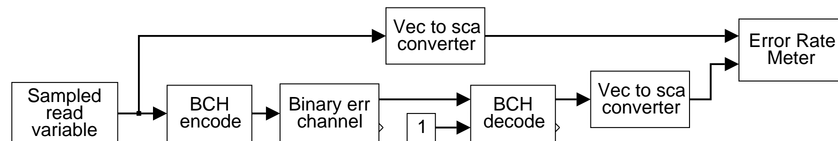
There are two main input/output formats for the blocks in the Error-Control Coding sublibrary: the vector input/output format and the sequential input/output format.

Vector Input/Output Format

For vector input/output format, the encode blocks accept length K vectors as their input signals and output length N codeword vectors. The input/output length for decoding blocks is the reverse of the encoding blocks. A decode block accepts a length N codeword vector and outputs a length K recovered message. This figure shows the vector input/output format for encoding and decoding a message:



Vector Input/Output Format Block Diagram. You can build a Simulink block diagram to simulate the BCH code using the vector input/output format. A simple block diagram is shown below:

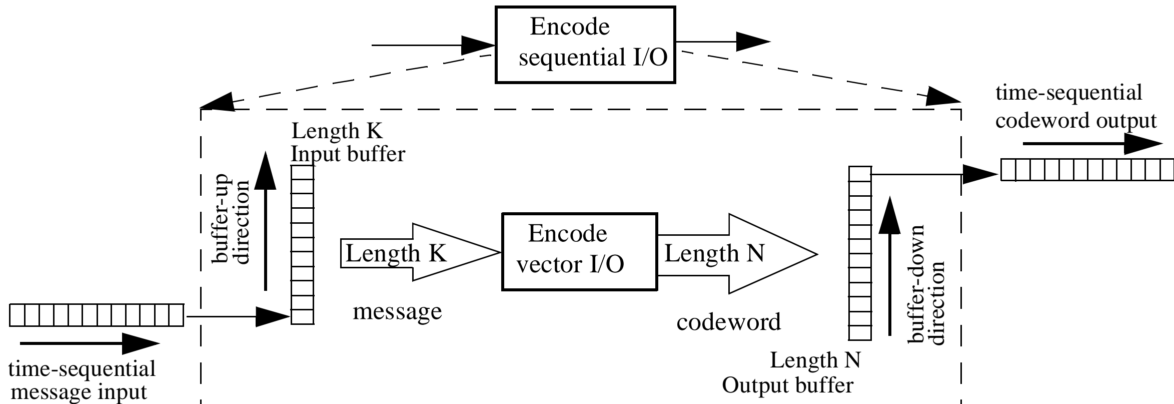


In the block diagram, the Sampled Read Variable block is the data source. The BCH encode and decode blocks are dedicated to the encoding and decoding processes respectively. The Binary Err Channel block simulates a digital transmission channel, which adds errors into the encoded binary signal. The Error Rate Meter block compares the source data and the decoded data and calculates the error rate. The simulation error-rate data is shown in the figure opened from the error-rate meter. A snapshot of the error-rate meter is:

Sender	Receiver
Symbol Transfered	1673
Error Number	11
Error Rate	0.0065789474
Bit Transfered	6688
Error Number	20
Error Rate	0.0029904306

Sequential Input/Output Format

In the sequential input/output format, the blocks automatically buffer up the sequential input signal into a vector signal. The output mechanism automatically buffers down the vector signal to a sequential signal. The buffer up/down procedure causes delays. The delay time is the input buffer length multiplied by the input sampling time, or equivalently, it is the output buffer length multiplied by the output sampling time. Refer to the Simulink reference page for each block for its delay time. Note that the sample time for an input signal is different from the sample time for an output signal. The sample time for the encode block is based on the block input sequence and the sample time for the decode block is based on the block output sequence. The figure below shows a sequential input/sequential output encoding block diagram.

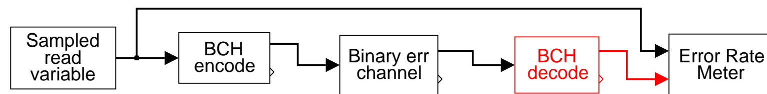


When input buffer is filled, the filled-up signal triggers the processing of the following: encode, empty input buffer, and refresh output buffer. The time delay of this block fills up the input buffer, which equals the input_sample_time * K or the output_sample_time * N. Note that the input_sample_time need not equal the output_sample_time.

The sequential input/output decoding process is the exact inverse of the encoding process.

To obtain a correct decode result, the decode method and parameters must match the encode counterpart exactly, except for timing and phase offsets, which may differ.

Example of the Sequential Input/Output Format. A sequential input/sequential output example can be built to simulate the BCH code encoding and decoding process. The block diagram for the sequential input/sequential output block diagram is:



To learn how to set the parameters correctly, see the Simulink reference entries on BCH block sequential code/decode blocks. See Chapter 5, “MATLAB Function Reference,” for descriptions of the Galois field computation functions.

Error-Control Coding Library Blocks

The blocks included in the error-control coding library and the categories they belong to are as follows. The encoding computations in the following list can be done by using function encode. The decoding computations in the following list can be done by using function decode.

Category	Blocks
Linear Block Code	Linear Block Encode Vector In/Out
	Linear Block Decode Vector In/Out
	Linear Block Encode Sequential In/Out
	Linear Block Decode Sequence In/Out
Hamming Code	Hamming Encode Vector In/Out
	Hamming Decode Vector In/Out
	Hamming Encode Sequential In/Out
	Hamming Decode Sequential In/Out
Cyclic Code	Cyclic Encode Vector In/Out
	Cyclic Decode Vector In/Out
	Cyclic Encode Sequential In/Out
	Cyclic Decode Sequential In/Out
BCH Code	BCH Code View Table
	BCH Encode Vector In/Out
	BCH Decode Vector In/Out
	BCH Encode Sequence In/Out
	BCH Decode Sequence In/Out

Category	Blocks
Reed-Solomon Code	Reed-Solomon Encode Integer Vector In/Out
	Reed-Solomon Decode Integer Vector In/Out
	Reed-Solomon Encode Binary Vector In/Out
	Reed-Solomon Decode Binary Vector In/Out
	Reed-Solomon Encode Integer Sequence In/Out
	Reed-Solomon Decode Integer Sequence In/Out
	Reed-Solomon Encode Binary Sequence In/Out
	Reed-Solomon Decode Binary Sequence In/Out
Convolutional Code	Convolutional Encode Vector In/Out
	Convolutional Decode Vector In/Out
	Convolutional Encode Sequential In/Out
	Convolutional Decode Sequential In/Out

Linear Block Codes

Linear block coding is a generic coding method. Other coding methods, such as Hamming and BCH codes, are special cases of linear block coding. The codeword vector of a linear block code is a linear mapping of the message vector. The codeword v and the message u have the relationship $v = uG$.

where G is a K-by-N matrix with all elements in GF(2). G is known as the generator matrix.

Linear block code is called a systematic linear code if the generator matrix has the form $G = [P, I]$, where P is an (N-K)-by-K matrix and I is a K-by-K identity matrix. (Note that some authors define the generator matrix as $[I, P]$.) A systematic linear code renders a length K message into a length N codeword where the last K bits are exactly the original message and the first (N-K) bits are redundant. These redundant bits serve as parity-check digits. Any linear

code can be mapped into a systematic linear code, known as the *equivalent systematic linear code.*

The recovery of the length K message from the codeword involves the calculation of a matrix called the syndrome. The syndrome is a matrix that, when multiplied by the codeword vector, produces the original message plus a length (N-K) set of error-correction bits. Within the limitations of the designed code, the syndrome detects the number of errors contained in the received message.

Assume that the received sequence r is the codeword v plus an error sequence e in the channel $r = v + e$.

The decoding process includes these four steps:

1 Compute the syndrome of r.

2 Locate the error e by using the syndrome.

3 Decode the received vector into the code vector $v=r+e$.

4 Recover the message vector from the reconstructed codeword vector v.

The syndrome is computed by $s = rH^T$ where H is called a *parity-check matrix*. H is a null space matrix of the generator matrix G. H is a (N-K)-by-K matrix, which has the property that $GH^T = 0$

Since $r=v+e$, $v=uG$, and $GH^T=0$, we have $s=rH^T=eH^T$, which means that the syndrome is a linear function of the transmission error. This toolbox provides a function called `gen2par` to compute H from G for systematic linear block code. The error location can be determined by a logic circuit truth table using the syndrome. In the toolbox, a single error-detection truth table can be calculated by using the command `htruthtb`. Once e is known, v can be calculated simply by using GF(2) addition; that is, addition using the XOR logical operator. In order to recover the message vector from the codeword, a matrix Q with $GQ=I$ has to be found, where I is a K-by-K identity matrix. For a systematic (N, K) code, Q is

$$Q = \begin{bmatrix} 0_{K\times(N-K)} \\ I_{K\times K} \end{bmatrix}$$

Example of a Linear Block Code

In this example, assume the generator matrix is

$$G = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

which means that the linear block code is a (7, 4) code. An encode/decode process using the MATLAB functions encode and decode is:

```
G = [[1 1 0;0 1 1;1 1 1;1 0 1],eye(4)];
[K, N] = size(G);
msg = randint(K*200,1,2);
code = encode(msg,N,K,'block',G);
code_noise = rem(code+rand(N*200,1)>.95,2);
rcv = decode(code_noise,N,K,'block',gen2par(G));
disp(['Error rate in the received code: ',...
    num2str(symerr(code,code_noise)/length(code))])
Error rate in the received code: 0.05
disp(['Error rate after decode: ',...
    num2str(symerr(msg, rcv)/length(msg))])
Error rate after decode: 0
```

Hamming Codes

A Hamming code is a linear block code. In this toolbox, the codeword length N of Hamming code is a fixed number equal to 2^M-1, where M is an integer larger than or equal to 3. M is also the parity check length. The message length of Hamming code is 2^M-M-1. Hamming code is a single error-correction code, which means that only one-bit corrections are possible per codeword. A list of some valid Hamming code parameters (codeword length N, message length K, and parity check length M) is provided in the table below:

N	7	15	31	63	127	255	511	1023	2047	4095	8191	16383	32767
K	4	11	26	57	120	247	502	1013	2036	4083	8178	16369	32752
M	3	4	5	6	7	8	9	10	11	12	13	14	15

The generator matrix and the parity check matrix for Hamming code can be produced by the function `hammgen`. The truth table to find the error location from the syndrome for Hamming code can be found by using the command `htruthtb`.

In communications theory, it is possible to reduce N, the codeword length, in certain situations where throughput is an issue or when there is more redundancy in the code than is necessary for the application at hand. The resulting code is called *shortened code*. This toolbox does not support shortened code.

Cyclic Codes

Cyclic code is a linear code produced by a cyclic generator polynomial. The generator polynomial *f(X)* of a (N, K) cyclic code is degree N-K polynomial over the GF(2) field. Cyclic codes are attractive in that they possess certain algebraic properties that allow practical decoding implementations. This toolbox provides a function `cyclpoly` to find cyclic generator polynomials. For example, you can find all valid (15, 9) cyclic generator polynomials by using:

```
pl=cyclpoly(15, 9)
pl=
1 0 0 1 1 1 1
1 0 1 1 1 0 1
1 1 1 1 0 0 1
```

Given a cyclic polynomial, you can find the cyclic generator matrix and parity-check matrix by using the function `cyclgen`. You have the choice of either `'normal'` or `'systematic'` code. A systematic cyclic code is recommended for most applications.

With the generator matrix and the parity-check matrix available, the encode and decode procedures are the same as those discussed in the “Linear Block Code” section.

BCH Codes

BCH code is another linear block code. Unlike Hamming codes, BCH code permits multiple-error bit corrections. BCH code is named after the inventors of the code: Bose, Chaudhuri, and Hocquenghem. The codeword length of a BCH code is 2^M-1, where M is an integer larger than or equal to 3. The number of information bits is determined by the construction of the cosets of GF(2^M).

You can use the function `gfcoset` to generate cosets of GF(2^M). For a discussion of what cosets are, refer to Lin & Costello.

BCH Encoding

The error-correction capability of BCH code is a function of the codeword and the message lengths. There is no closed-form formula for determining the BCH code error-correction capability. This toolbox provides the MATLAB function `bchpoly` to calculate the BCH generator polynomial. This function also displays the valid BCH codeword length, message length, and error-correction capability. For example, you can find the valid message lengths for a given codeword length by:

```
bchpoly(N)
```

If `N` is not a valid codeword length, the function will find the smallest valid codeword larger than the given integer. For $N \leq 1023$, the function `bchpoly` produces a list of valid message words and error-control capability. For $N>1023$, the function provides a list of valid message lengths only. When the function
`bchpoly` is called with no input and output variables, it lists the codeword length, message length, and error-control capability for all codeword lengths no larger than 511. The next figure lists all message lengths and error-correction capabilities for all valid codeword lengths up to 2^{11}-1=511:

N	K	T	N	K	T	N	K	T
7	4	1	255	199	7	511	358	18
15	11	1		191	8		349	19
	7	2		187	9		340	20
	5	3		179	10		331	21
31	26	1		171	11		322	22
	21	2		163	12		313	23
	16	3		155	13		304	25
	11	5		147	14		295	26
	6	7		139	15		286	27
63	57	1		131	18		277	28
	51	2		123	19		268	29
	45	3		115	21		259	30
	39	4		107	22		250	31
	36	5		99	23		241	36
	30	6		91	25		238	37
	24	7		87	26		229	38
	18	10		79	27		220	39
	16	11		71	29		211	41
	10	13		63	30		202	42
	7	15		55	31		193	43
127	120	1		47	42		184	45
	113	2		45	43		175	46
	106	3		37	45		166	47
	99	4		29	47		157	51
	92	5		21	55		148	53
	85	6		13	59		139	54
	78	7		9	63		130	55
	71	9	511	502	1		121	58
	64	10		493	2		112	59
	57	11		484	3		103	61
	50	13		475	4		94	62
	43	14		466	5		85	63
	36	15		457	6		76	85
	29	21		448	7		67	87
	22	23		439	8		58	91
	15	27		430	9		49	93
	8	31		421	10		40	95
255	247	1		412	11		31	109
	239	2		403	12		28	111
	231	3		394	13		19	119
	223	4		385	14		10	121
	215	5		376	15			
	207	6		367	16			

N: code word length; K: message length; T: error–correction capability

Figure 3-1: Message Lengths and Error-Correction Capabilities for Valid Codeword Lengths in BCH Code

You can obtain generator polynomials by using the function bchpoly with a valid codeword length and message length specified.

With a given generator polynomial, you can use cyclgen to produce the generator matrix and then follow the cyclic encode procedure for the encoding computation.

BCH Decoding

The decoding procedure for the BCH code is much more complicated than the encoding process. The decoding process includes these four steps:

1. Compute syndrome from the received codeword (codeword corrupted by noise).
2. Determine the error-location polynomial from the syndrome.
3. Find the roots of the error-location polynomial, which are the error-location numbers.
4. Reconstruct the corrected codeword and determine the message from the codeword.

The algorithm is rather complicated. Refer to Lin & Costello for more information or see the function bchcore for the implementation of BCH decoding.

Reed-Solomon Codes

Reed-Solomon (RS) codes are burst error-correction codes. All of the other coding methods provided in this toolbox are random error-correction codes. A burst error occurs when noise corrupts several consecutive bits. Burst errors frequently occur in computer data storage access. Because of this, RS code is widely used in computer data storage environments, especially in CD-ROM error-correction systems.

The theory of Reed-Solomon code is based on finite field theory. In particular, the fields used are of the form $GF(q^M)$, where q is any prime number and M any positive integer. This toolbox supports RS code on $GF(2^M)$. The elements of $GF(2^M)$ are defined by a power series format, i.e., α^{-Inf}, α^0, α, α^2,.... The message length K can be any positive integer smaller than N, where N is the codeword length.

The basic parameters of RS code are:

- Codeword length: $N = 2^M - 1$
- Number of check symbols: N-K = 2*T
- Error-correction capability: T = floor((N-K)/2)

For an efficient RS code, N-K should be an even number. With the exception that the elements of RS code are in $GF(2^M)$ instead of in GF(2), all computation procedures are the same as those for BCH code.

The generator polynomial for RS code is a degree 2*T polynomial with its coefficients in $GF(2^M)$. This toolbox provides the MATLAB function `rspoly` to produce RS code generator polynomials.

The decode procedure is the same as for the BCH code. Refer to Lin & Costello for more information or see the function `rscore` for the implementation of RS decoding. The data input for RS code/decode can be one of the three forms: binary, integers in the range from 0 to 2^M-1, or power with the elements in $GF(2^M)$.

Convolutional Codes

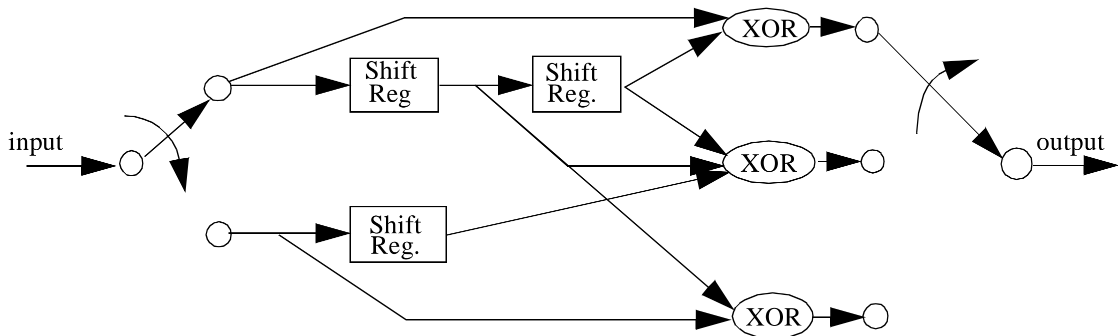
Up to now, all the decoding methods discussed are algebraic decoding methods, which are *memoryless* codes. A memoryless code makes decoding decisions with only the current codeword; code with memory uses information from previously received codes to make decoding decisions about the most recently received codeword. Since convolutional code makes decisions about the current codeword based on past information, it contains memory elements. To include the memory elements in the code, an extra parameter M is used to describe a convolutional code. The parameter M is referred to as the *constraint length* of the code. The constraint length is the maximum number of information symbols upon which the symbol may depend. Two other parameters are needed to describe the code. The parameter N designates the number of symbols generated by the encoder from any given block of M contiguous information symbols. K is the number of informations symbols by which the block is advanced each time that code symbols are generated. A convolutional code so constructed is called an (N, K, M) code.

Convolutional codes are commonly used in applications that require relatively good performance with low implementation cost.

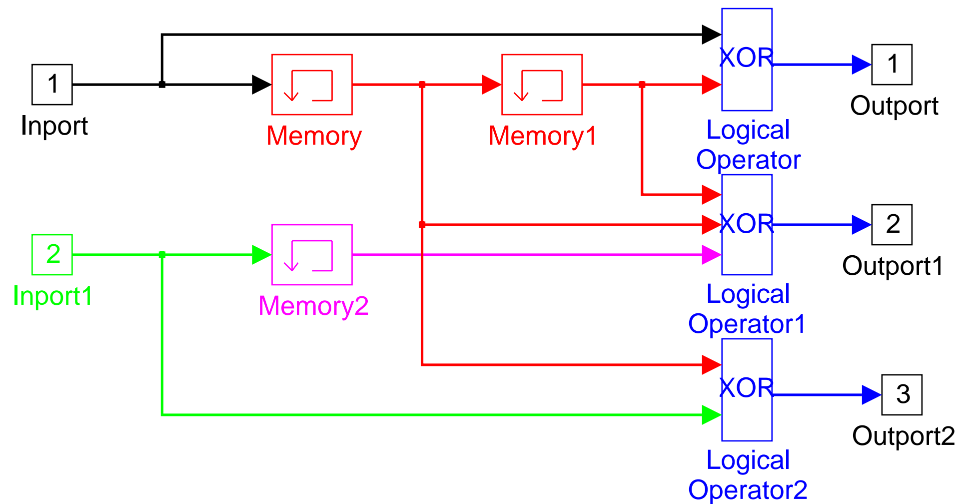
The Transfer Functions of Convolutional Coding

The convolutional code transfer function determines the structure of the code. Many different convolutional code transfer functions may share the same (N, K, M) number, so exact specification of a convolutional code requires knowledge of its transfer function. In this toolbox, the transfer function matrix for convolutional code can be represented by two different forms: octal and binary. This toolbox provides an interface using Simulink block diagrams to generate a convolutional code transfer function.

A block diagram example of a (3, 2, 2) convolutional code is:



This (3, 2, 2) block diagram of a convolutional code uses this Simulink block diagram representation:



Note that five different Simulink blocks have been used in the convolutional code structure definition: Inport, Memory, XOR, and Outport blocks. The functionality of each block can be found by comparing the above two figures. You can convert a Simulink block diagram into a transfer function matrix by using the command `sim2gen`. `sim2gen` converts a Simulink block diagram to an octal form transfer function matrix.

Transfer functions for convolutional codes can be either linear or nonlinear. In this toolbox, linear transfer functions fall into three categories:

- binary form
- octal form
- state space form

Nonlinear transfer functions for convolutional code have only one format in the Communications Toolbox. This nonlinear format can be viewed as an input/output mapping function that maps all possible inputs and states to the possible outputs. The next four section discuss the linear and nonlinear transfer formats in turn.

The Binary Form . The binary transfer function matrix `trans_func` is a K-by-N*(M+1) binary matrix. `tran_func` can be written in the following form:

$$\text{tran_func} = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1N} \\ \cdots & & \cdots & \\ t_{K1} & t_{K2} & \cdots & t_{KN} \end{bmatrix}$$

where t_{ij} is a length M+1 binary row vector, which is the transfer function from ith input to jth output. The row vector t_{ij} represents an ascending ordered polynomial. For example the (3, 2, 2) convolutional code

$$\begin{bmatrix} 1 + X^2 & X + X^2 & X \\ 0 & X & 1 \end{bmatrix}$$

can be represented by the binary form with the specified `code_param=[3 2 2]` and

$$\text{tran_func} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The Octal Form . The octal form of the transfer function matrix `gen` has the form:

$$\text{tran_func} = \begin{bmatrix} g_{11} & g_{12} & \cdots & g_{1N} \\ \cdots & & \cdots & \\ g_{K1} & g_{K2} & \cdots & g_{KN} \end{bmatrix}$$

in which g_{ij} is the ith input to the jth output octal form transfer function. Note that the binary form of the transfer function is in ascending order. The octal form is a conversion of the binary vector along the ascending direction. The octal form of the transfer function `gen` is:

$$\text{gen} = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 2 & 4 \end{bmatrix}$$

The State-Space Form. The state-space transfer function is represented in a linear system form. A linear system can be written as a difference equation

$$\begin{aligned} x(k+1) &= \mathrm{A}x(k) + \mathrm{B}u(k) \\ y(k) &= \mathrm{C}x(k) + \mathrm{D}u(k) \end{aligned}$$

where $u(k)$ is the input vector, $x(k)$ is the state, and $y(k)$ is the output vector. Multiplication and addition are binary operations. This function outputs the transfer function in the form of

$$\text{gen} = \left[\begin{array}{cc|c} \mathrm{A} & \mathrm{B} & \mathrm{V} \\ \mathrm{C} & \mathrm{D} & \end{array}\right]$$

where V is a column vector with its first three element being the output vector length, the input length, and the state vector length. The last element of the vector is assigned to be *-Inf*.

Using the function `sim2tran`, the state space form of the transfer function can be obtained by using the command:

```
gen = sim2tran('fig_10_9')
gen =
```

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 3 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & -Inf \end{bmatrix}$$

Using command `gen2abcd`, you can find the `[A, B, C, D]` state space matrices described in the difference equation defined in this section.

Nonlinear Transfer Functions for Convolutional Coding. A nonlinear system can be written as the set of difference equations

$$\begin{aligned} x(k+1) &= f(x(k), u(k)) \\ y(k) &= g(x(k), u(k)) \end{aligned}$$

where $u(k)$ is the input vector, $x(k)$ is the state, and $y(k)$ is the output vector. The structure of the nonlinear transfer function is

$$\begin{bmatrix} \mathrm{Inf} & N \\ M & K \\ U & V \end{bmatrix}$$

where N is the output vector length, K is the input vector length, M is the state vector length, and U and V are input and output column vectors that have lengths equal to 2^{K+M}. The symbol Inf in the first row, first column entry position indicates that this transfer function is in the nonlinear format. The rest of this section discusses the structure of the nonlinear format, and how it can be viewed as an input/output mapping function.

In convolutional coding, the input signals, state values, and outputs are binary. While it is possible to construct the nonlinear transfer function in binary form, you can convert a length M binary state vector into an integer format. The discussion that follows assumes that the input, state, and output vectors have been converted from binary to integer. In convolutional coding, the range of possible inputs, states, and outputs are finite. After converting from binary to integer values, the possible values for the states and the outputs are in the range $[0 : 2^M-1]$. The possible input values are in the range $[0 : 2^K-1]$.

The column vector U contains the possible input values of u(k). The matrix that represents the nonlinear transfer function maps inputs of given values to the resulting outputs:

- The first 2^M elements of V are the values of y(k) when u(k) = 0 and $x = [0 : 2^M - 1]$.
- The second 2^M elements of V are the values of y(k) when u(k) = 1 and $x = [0 : 2^M - 1]$.
- This process continues until it exhausts all possible input and state values.
- The last 2^M elements of V are the values of y(k) when $u(k) = 2^K-1$ and $x = [0 : 2^M - 1]$.

This is a mapping of all the possible inputs u(k) and states x(k) to the outputs y(k). The nonlinear transfer function can also be viewed as a lookup table whose lookup variables are the input and state values.

Note that the nonlinear transfer function constructed here does not share the mathematical properties of linear transfer functions. For example, none of the

spectral theory of linear transfer functions applies. One advantage of nonlinear transfer functions is computational speed. Since no mathematical operations occur other than the assignment of an output to given input and state values, the output is immediately available.

For example, you can construct a nonlinear transfer function based on the Simulink block diagram of (3, 2, 2) convolutional code discussed in the "Transfer Functions for Convolutional Code" section of this chapter. For this to work, replace the Memory blocks by Unit Delay blocks in the block diagram and save it as `nlconcd`. Use the `sim2tran` function to generate the nonlinear transfer function:

```
gen = sim2tran('nlconcd')
gen =
```

$$\begin{bmatrix} \mathrm{Inf} & 3 \\ 2 & 2 \\ 0 & 0 \\ 0 & 4 \\ 1 & 0 \\ 1 & 4 \\ 3 & 5 \\ 3 & 1 \\ 2 & 5 \\ 2 & 1 \\ 2 & 6 \\ 2 & 2 \\ 3 & 6 \\ 3 & 2 \\ 1 & 3 \\ 1 & 7 \\ 0 & 3 \\ 0 & 7 \end{bmatrix}$$

The result is the nonlinear transfer function. For more information on nonlinear transfer functions, refer to the entry for `sim2tran` in Chapter 5, "MATLAB Function Reference."

Convolutional Encoding

You can perform convolutional encoding by using the command `convenco`. In the encoding process, a convolutional code transfer function can be converted into a discrete-time transfer function

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{aligned}$$

in which u is the length K input vector and y is the length N output vector. The matrices A, B, C, and D are constant matrices with all elements in GF(2). The number k is an non-negative integer that specifies the computation time. The computation assumes the initial state $x(0)$ is a zero vector.

Convolutional Decoding

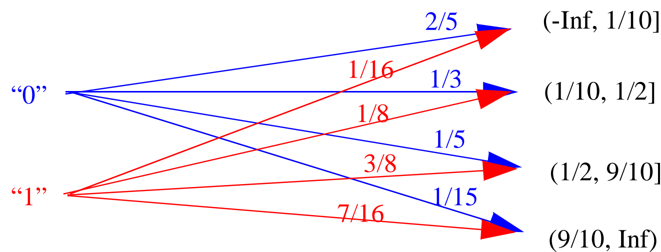
This toolbox uses the Viterbi method for decoding of convolutional codes. The Viterbi algorithm is a maximum likelihood decoding procedure that takes advantage of the fact that a convolutional encoder is a finite state device. If the constraint length, M, is relatively small, i.e. $M \le 8$, the decoder can perform optimal decoding with relatively few computations.

The Viterbi decoder performs either *hard decision* or *soft decision* decoding. A hard decision decoder accepts bit decisions from the demodulator. These decisions assign either a 0 or 1 to each received bit. Soft decision decoding involves using demodulator outputs that are proportional to the *log likelihood* of the demodulated bit being either 0 or 1. The log likelihood is the logarithm of the probability of a bit being, for example, 0 if the demodulated bit was 1. If the transmitted bit is variable x and the demodulated bit is variable z, this expression is the log likelihood that x is 0 given that z is 1:

Soft decision decoding generally produces better performance at the cost of increased complexity. The next two sections discuss soft and hard decision decoding in turn.

Soft Decision Decoding. For soft decision decoding, the received codeword can be any real number. Because of channel fading, noise, interference, and other factors, a signal transmitted as a 0, for example, may be received as some nonzero real value x. From *a priori* knowledge of the channel, you can estimate the probabilities of the value x falling in certain regions. A similar analysis

holds for the transmission of a signal 1. Use the notation P(rcv|msg) to represent the probability of receiving rcv data while sending msg data. Using the definition, we can describe a transmitting channel by a set of transfer probabilities. For instance, in the following channel,



the probability of signal 0 being received in range (-Inf, 1/10] equals 2/5 or P((-Inf, 1/10]|0) = 2/5. The probability of signal 1 being received in range (-Inf, 1/10] equals 1/16 or P((-Inf, 1/10]|1) = 1/16. This table presents the probability structure associated with the communication channel:

Receiving Probability When Sending 0	Receiving Probability When Sending 1
P((-Inf, 1/10]|0) = 2/5	P((-Inf, 1/10]|1) = 1/16
P((1/10, 1/2]|0) = 1/3	P((1/10, 1/2]|1) = 1/8
P((1/2, 9/10]|0) = 1/5	P((1/2, 9/10]|1) = 3/8
P((9/10, Inf)|0) = 1/15	P((9/10, Inf)|1) = 7/16.

Round brackets indicate an open range. Open ranges do not include the boundary points. Square brackets indicate a closed range. Closed ranges include the boundary points.

The optimum path for a soft decision maximizes the metric of the received signal *r* and the corrected signal *c*. For a codeword length N transmission, the *metric* M(r|c) is defined as

$$M(r|c) = \sum_{i=1}^{N} \log P(r_i|c_i)$$

where r_i is the ith element of the received codeword vector and c_i is the ith element of the corrected vector.

The MATLAB functions `decode` and `viterbi` and the Simulink blocks Cyclic Decode Vector In/Out and Cyclic Decode Sequence In/Out require that you define a transfer probability variable `trans_prob`, a three row matrix. The first row of `trans_prob` is the lower boundary of the received range. The second row is the probability of receiving signal in the range specified in the first row given that the transmitter sent a 0 signal. The third row is the probability of receiving signal in the range in the first row given that the transmitter sent a 1 signal. The format is:

$$\text{trans_prob} = \begin{bmatrix} LB1 & LB2 & \ldots \\ P\langle[LB1,LB2]|0\rangle & P\langle[LB2,LB3]|0\rangle & \ldots \\ P\langle[LB1,LB2]|0\rangle & P\langle[LB2,LB3]|0\rangle & \ldots \end{bmatrix}$$

Continuing with the channel example, the transfer probability is:

$$\text{trans_prob} = \begin{bmatrix} -\infty & \frac{1}{10} & \frac{1}{2} & \frac{9}{10} \\ \frac{2}{5} & \frac{1}{3} & \frac{1}{5} & \frac{1}{15} \\ \frac{1}{16} & \frac{1}{8} & \frac{3}{8} & \frac{7}{16} \end{bmatrix}$$

Hard Decision Decoding. For hard decision decoding, the received codewords are binary data. The probability of signal 0 being received as 0 is q, or P(0 | 0) = q; P(1 | 0) = 1-q. The probability of signal 1 being received as 1 is q, or P(1 | 1) = q; P(0 | 1) = 1-q. This toolbox always assumes q>1/2. In case q<1/2, you can simply reverse ones and zeros prior to the decoding process. The optimum path is the one that has the minimum Hamming distance between the received codeword and the corrected codeword (know as the *survivor*). The Hamming distance between two binary vectors is defined as the number of indices where the two vectors do not match.

The Viterbi Algorithm. The Viterbi algorithm is good for both soft and hard decision decoding. The criterion used for decision making is the metric for soft decision decoding and the Hamming distance for hard decision decoding.

A step-by-step description of the Viterbi algorithm is:

1 Assign initial conditions:

   i = 0; the current computation index.
   j = 0; the current decision index.
   $x_1(0) = 0$; the initial state.
   $m_1=0$; the initial criterion (metric for soft decision decoding and Hamming distance for hard decision decoding).

2 Using all valid $x_b(i)$, compute all possible input paths to $x_e(i+1)$. If there is more than one input path to $x_e(i+1)$, keep the one that is optimal under the criterion you are using. This is the maximum metric value, called the *metric accumulation*, in soft decision decoding and the minimum Hamming distance in hard decision decoding. Keep the input path leading to $x_e(i+1)$ and the criterion value of that path. Note that b and e are possible numbers between 1 and $2^{\text{length}(x)}$.

3 Tracing from $x_e(i+1)$, remove all paths that cannot reach step i+1.

4 For a limited memory case, where you are evaluating paths of fixed length only, check if the buffer is full, i.e., if i-j is greater than the fixed path length. If it is full, use the optimal value calculated under the criterion you are using to decide which state to keep and clear the memory space of path segments that are no longer being used (this means you must set j=j+1). For an unlimited memory case, check if the beginning of the whole path x(j) to x(j+1) is a single connection. If it is, keep the path as the result of the decision. j=j+1.

5 If i+1 is not the encode of the codeword transfer, i=i+1, go to 2. Otherwise go to 6.

6 Take $x_e(i+1)=0$ as the final state, trace back to j all of the paths in the decision.

This toolbox provides function `convenco` for convolutional encoding. Function `viterbi` implements the convolutional decode Viterbi algorithm for both limited memory length and unlimited memory length.

A *trellis* is the set of paths of the possible transitions from one state to another. You can view the trellis using `viterbi`. In trellis plots, the path with bold lines is the decision path, which is always the optimal path for the given conditions.

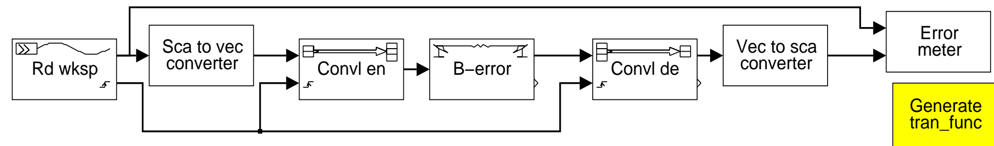
You can also use functions `encode` and `decode` with option `convolution` for convolutional encode and decode calculations. Convolutional encode/decode Simulink blocks are also available in the Error-Control Code sublibrary.

Example of Convolutional Decoding

This example uses the (3, 3, 2) convolutional code discussed on page 3-42 in the "Transfer Functions of Convolutional Code" section of this chapter. You can view the block diagram of the transfer function by typing:

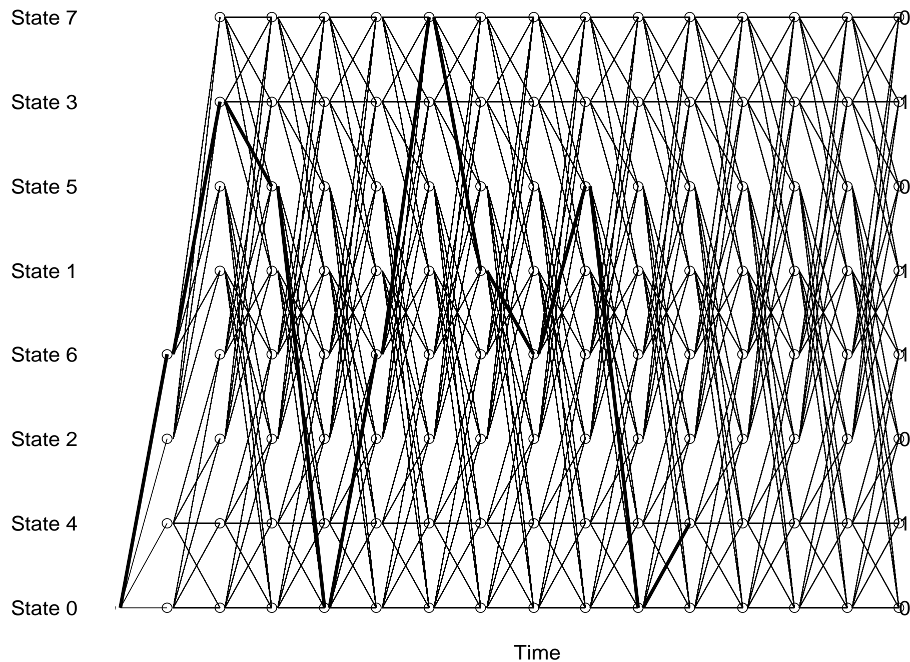
```
tutconmd
```

The figure below shows a block diagram used for testing the convolutional code with the (3, 3, 2) convolutional code transfer function:



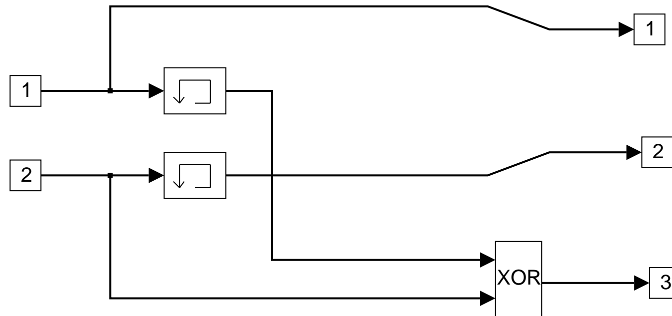
Convolutional decoding uses the Viterbi algorithm. You can view the trellis used in the Viterbi decoding process while running the simulation by entering the trellis length in the **Trellis figure plot length** parameter in the

Convolutional Decode block in the above diagram. This figure shows the trellis used in the computation:

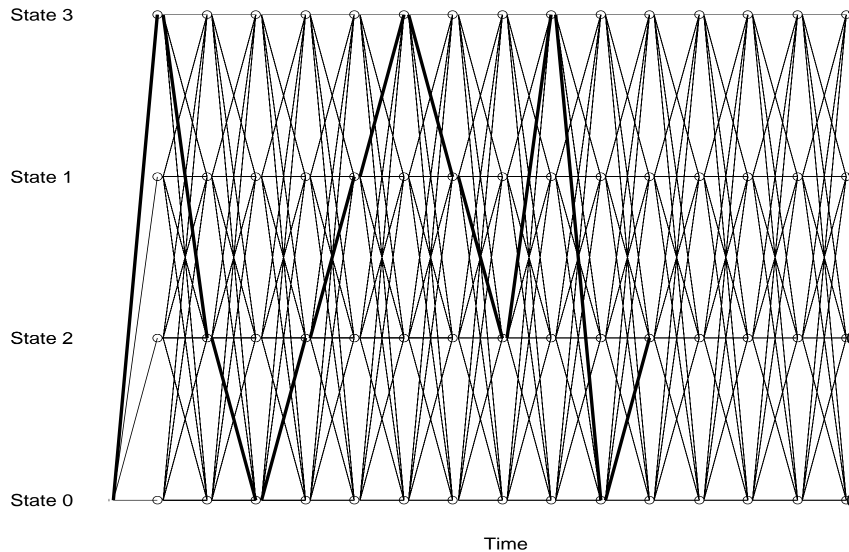


In the trellis plot, the lines are all the possible decision paths. The bold lines are the optimal path, which is used as the decoding path. The numbers on the right-hand side are the Hamming distance in the decoding. You can change the

convolutional code transfer function by changing the Simulink block diagram for convolutional code structure. For instance, you can change the structure to:



The simulation with the changed block diagram has a different trellis diagram. There are now only four states in the trellis plot instead of eight:



If you know the transfer function you intend to use, you can directly use it instead of using the block diagram method. For example, assume that the information signal sequence is in vector `msg`. Instead of using Simulink blocks,

generate the transfer function using sim2tran and work directly with it. The MATLAB code needed to run the encode and decode process using the transfer function is:

```
trans_func = sim2tran('tutconmd');
code_word = encode(msg,size(trans_func,2),...
    size(trans_func,1),'convolution',trans_func);
recovered = decode(msg,size(trans_func,2),....
    size(trans_func,1),'convolution',trans_func);
```

Refer to the entries for encode and decode in Chapter 5, "MATLAB Function Reference," for a more detailed discussion of how to use these functions.

Modulation and Demodulation

In most media for communication only a fixed amount of the frequency spectrum is available for transmission. Although the signal to be transmitted may have a relatively small bandwidth, its frequency spectrum may not fall within the necessary frequency range. The usual solution is to modulate the input signal with a carrier sinusoid that is within the desired frequency range. The resulting waveform will be within the desired frequency range and can be transmitted. At the receiver the waveform can be demodulated to recover the original signal.

There are two major categories in modulation/demodulation: *analog* and *digital*. In analog modulation, the input message signal is analog. In digital modulation, the input message signal is discrete.

In the simulation of modulation techniques, there are two options for representing the modulation/demodulation process: *passband* and *baseband*. In passband simulation, the carrier signal is included in the transmission model. The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. By the Nyquist sampling theorem, the simulation sampling frequency must be at least two times larger than the carrier signal frequency to recover the message correctly, but the simulation of a high frequency signal can be very slow and inefficient. To speed up the simulation process, a baseband simulation is commonly used when the parameter selection and performance of the modulation and demodulation are not the current design issue.

Baseband simulation, also known as the low-pass equivalent method, uses the *complex envelope* of a passband signal. The Baseband Analog Demodulation section of this chapter presents the mathematical definition of a complex envelope; here it is important to note that it is an important tool in baseband modulation.

The sections below discuss passband analog modulation/demodulation and passband digital modulation/demodulation in turn. Since the demodulation techniques used in baseband significantly differ from the modulation techniques, this chapter discusses each of these separately.

The sections under “Modulation and Demodulation” are:

- Passband analog modulation and demodulation
- Passband digital modulation and demodulation
- Baseband modulation and demodulation
- Baseband analog modulation
- Baseband analog demodulation
- Baseband digital modulation
- Baseband digital demodulation

Passband Analog Modulation and Demodulation

There are a number of methods for analog modulation: amplitude modulation (AM), frequency modulation (FM), and phase modulation (PM). The amplitude modulation includes double sideband suppressed carrier (DSB-SC) AM, double sideband with transmission carrier (DSB-TC) AM, single sideband suppressed carrier (SSB) AM, and quadrature amplitude modulation (QAM).

The function `amod` in this toolbox is designed for the analog passband modulation computation, and the function `ademod` for the analog passband demodulation computation. You can select modulation/demodulation methods by assigning an input string parameter `method`. The demodulation method chosen must match the modulation method in order to recover the message signal. The table below lists the modulation methods available:

Method	Meaning

amdsb-sc	Double-sideband suppressed amplitude modulation
amdsb-tc	Double-sideband with transmission carrier amplitude modulation
amssb	Single-sideband suppressed carrier amplitude modulation
fm	Frequency modulation
pm	Phase modulation
qam	Quadrature amplitude modulation

To use the function ademod and analog demodulation blocks, you must design a lowpass filter. The Signal Processing Toolbox provides a variety of functions for filter design, such as the functions butter, cheby1, cheby2, ellip, etc. This toolbox requires discrete-time filters in the demodulation functions and blocks. The algorithms in the demodulation process require that the lowpass filters eliminate the carrier signal. As a rule of thumb, you can define the cut-off frequency to be $f_c/2$, half of the carrier frequency. For example, if the computation sampling frequency is Fs, you can use the function butter to design an Nth order Butterworth lowpass filter with cut-off frequency Fc:

```
[num,den] = butter(N,Fc/Fs);
```

The vector outputs, num and den, are the numerator and the denominator of the lowpass filter transfer function.

The carrier frequency must be higher than the bandwidth of the message signal. The cut-off frequency limits the bandwidth of the message signal. When the cut-off frequency is too high, the carrier frequency may not be filtered out. When the cut-off frequency is too low, it can narrow the bandwidth of the message signal.

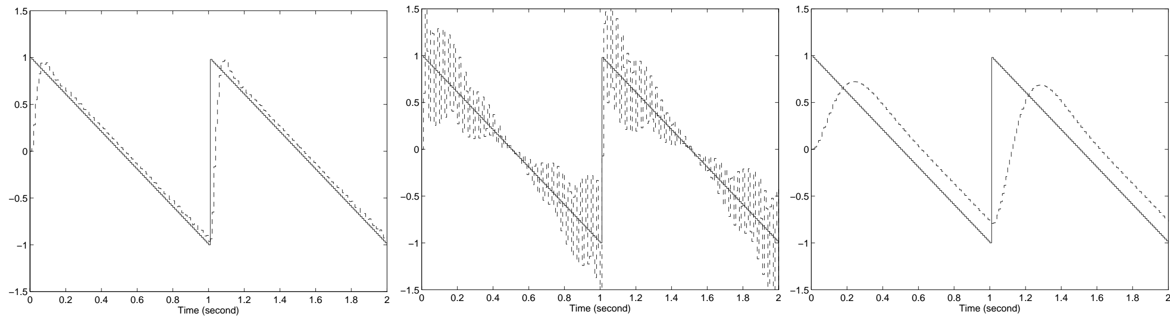
Example of the Effect of Lowpass Filters on Demodulation

In this example, a sawtooth message signal is modulated by using frequency Fc with computation sampling frequency Fs. The resulting signal is demodulated

by using three different cut-off frequency filters. The lowpass filters are designed by the command:

```
[num,den] = butter(2,F);
```

where F is the cut-off frequency in the design. In the figure below, three plots show the effect of the different lowpass filters:



In each plot, the dotted curve is the demodulation output. The left plot uses the suggested cut-off frequency F = Fc/Fs. The middle plot uses a higher cutoff frequency F = 4 Fc/Fs. The right plot uses a lower cut-off frequency F = Fc/Fs/4. The higher cut-off frequency allows the carrier signal to mix in with the demodulated signal, while a lower cut-off frequency significantly narrows the bandwidth of the demodulated signal. Both effects are undesirable.

Different demodulation methods have different properties; you may need to test several filters before finding the best filter for your application. The order of the filter is also an important issue. In general, a higher-order filter has steeper rolloff characteristics compared to a lower-order filter. However, computational issues may arise when the order is too high.

There is invariably a time 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.

Double-Sideband Suppressed Carrier Amplitude Modulation and Demodulation

The double-sideband suppressed carrier amplitude modulation (DSB-SC AM) method modulates a message signal *m(t)* using the formula

$$y(t) = m(t)\cos(2\pi f_c t + \phi_c)$$

where *y(t)* is the modulated signal, f_c is the carrier frequency (Hertz), and ϕ_c is the initial phase (rad).

The corresponding demodulation method, DSB-SC ADM, recovers the message signal *m(t)* from the received modulated signal *y(t)*. The demodulation procedure is shown in the block diagram below:

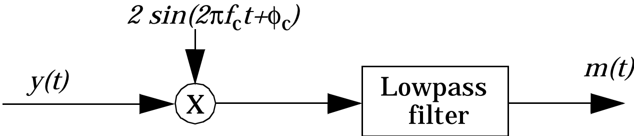


Figure 3-2: DSB-SC ADM Block Diagram

Costas Loop Demodulation Method with DSB-SC ADM. The DSB-SC ADM method requires knowledge of the modulation phase. If the exact modulation phase is not known, you may use the Costas phase-locked loop demodulation method, which recovers the phase from the received signal. The method is shown in this figure.

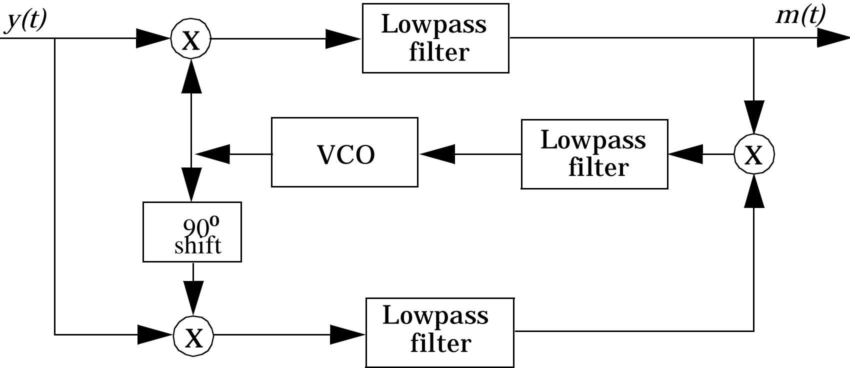


Figure 3-3: Costas Phase-Locked Loop Demodulation Block Diagram

In the figure, VCO is the voltage controlled oscillator. A discussion of voltage controlled oscillation is provided in the Utilities section of this chapter. In the Simulink block library, you can directly use the DSB-SC ADM block, which contains a Costas phase-locked loop. In function `ademod`, append `'/costas'` to the end of the `method` variable to use the Costas loop demodulation method. For example, if the modulated signal is stored in the variable `moded`, the demodulation process can be

```
recovered = ademod(moded,Fc,Fs,'amdsb-sc',initial_phase)
```

or, using the Costas loop method:

```
recovered = ademod(moded,Fc,Fs,'amdsb-sc/costas')
```

Double-Sideband with Transmission Carrier Amplitude Modulation and Demodulation

The double-sideband with transmission carrier amplitude modulation (DSB-TC AM) is a classical textbook method. This method is not used in practice because it is inefficient compared with other modulation methods in terms of energy costs. This method is very simple, however, and is commonly introduced in standard communications textbooks.

DSB-TC AM modulates a message signal *m(t)* using the form:

$$y(t) = (m(t) + \text{offset})\cos(2\pi f_c t + \phi_c)$$

where *y(t)* is the modulated signal, f_c is the carrier frequency (Hertz), and ϕ_c is the initial phase (rad). *offset* is the offset for the amplitude of the input signal. This figure compares DSB-SC AM and DSB-TC AM outputs:

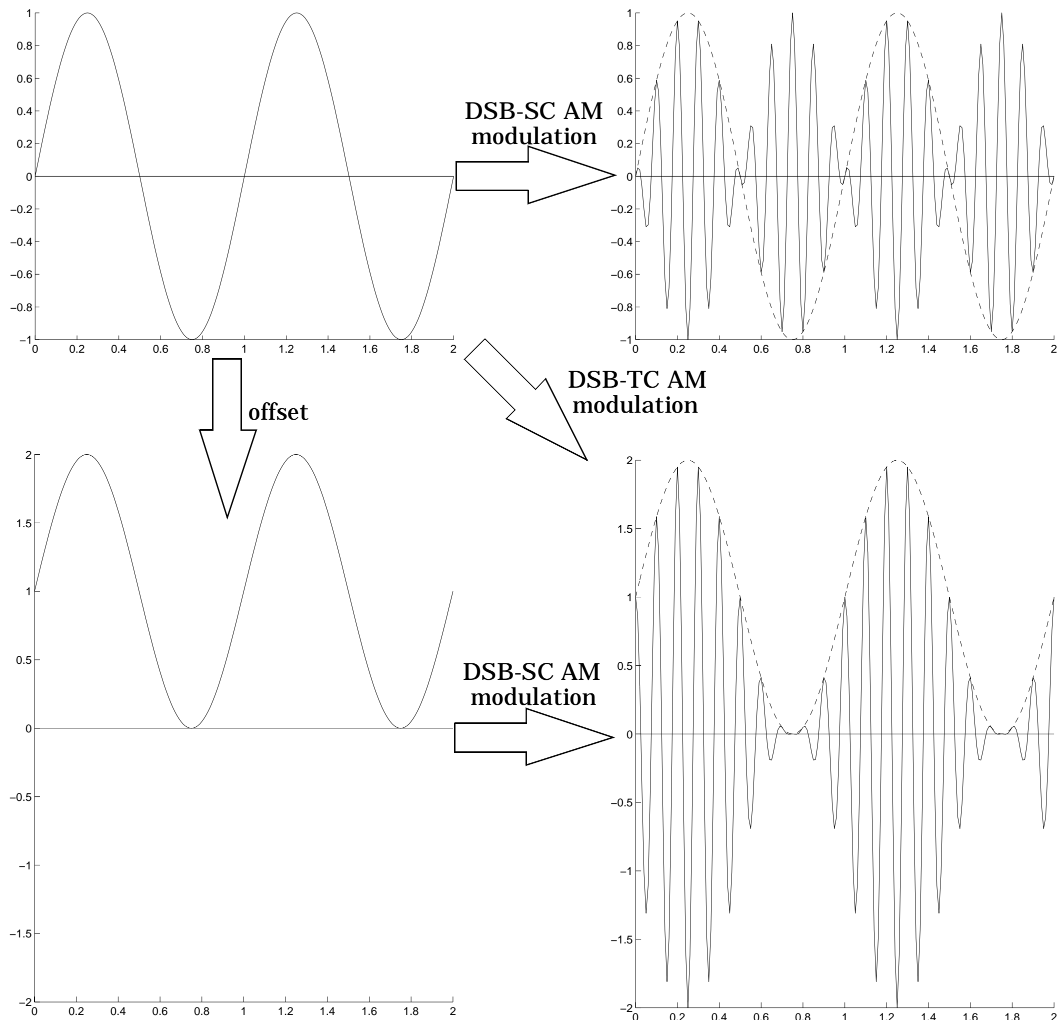


Figure 3-4: Comparison of DSB-SC AM and DSB-TC AM

In the figure, the upper left plot is the message signal,. The upper right plot is the DSB-SC AM modulated signal. On the bottom left is the message signal after shifting, and on the bottom right is the DSB-TC AM modulated signal. The relationships between the figures are shown by the arrows and the

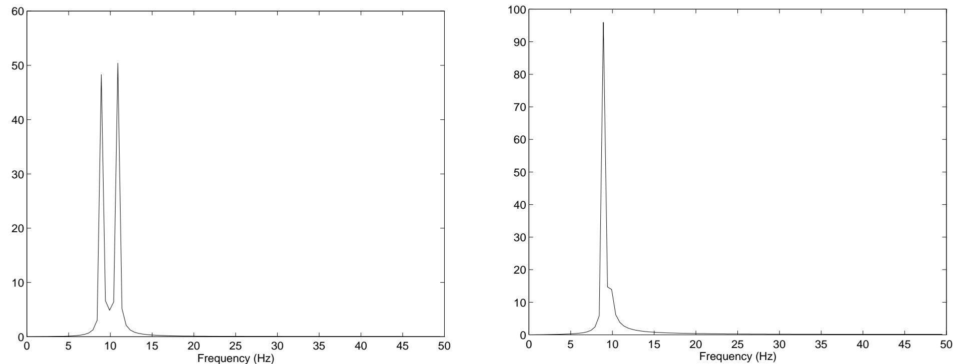
accompanying text. The command lines used in preparing the data to generate the above figures are:

```
% Computation time; sample rate 100 Hz. Carrier frequency 10 Hz.
Fs = 100; t = [0:200]/Fs; Fc = 10;
% Message signal is a 1 Hz sin; data for the upper-left plot.
x1 = sin(2*pi*t);
% Modulated signal; data for the upper-right plot.
% Modulation carrier frequency: 10 Hz.
y1 = amod(x1,Fc,Fs,'amdsb-sc');
% Message signal with offset; data for the left-bottom plot.
x2 = x1+1;
% DSB-SC AM modulates x2; data for the right-bottom plot.
y2 = amod(x2,Fc,Fs,'amdsb-sc');
% DSB-TC AM modulates x1; data for the right-bottom plot.
y3 = amod(x1,Fc,Fs,'amdsb-tc',1);
```

The corresponding demodulation method, DSB-TC ADM, recovers the message signal *m(t)* from the received modulation signal *y(t)*. The demodulation method is simply an edge detection process, which can be done by using a lowpass filter. In this toolbox the demodulation of a DSB-TC AM modulated signal uses the same algorithm as in the DSB-SC AM demodulation. The `offset` is subtracted after the DSB-SC ADM demodulation.

Single-Sideband Suppressed Carrier Amplitude Modulation and Demodulation

It is instructive to analyze the spectrum of signal y1 in the example provided in the DSB-TC AM discussion. The figure below compares the spectrums of the DSB-TC AM modulated signal and the same signal modulated by SSB-TC AM:



The message signal is a 1 Hz sinusoidal signal and the carrier signal Fc is a 10Hz sinusoidal signal. The spectrum in the left plot has twin peaks, which are the lower and the upper sidebands of the modulated signal. The two sidebands are symmetrical with respect to the 10 Hz carrier frequency, Fc. The spectrum of a DSB-SC AM modulated signal is twice as wide as the input signal bandwidth. To solve this problem, you can use a single-sideband suppressed carrier amplitude modulation (SSB AM) technique. SSB AM transmits either the lower sideband or the upper sideband signal. The right plot is the SSB AM modulated signal. You can generate the data for the SSB AM modulated signal using the command

```
y4 = amod(x1, Fc, Fs, 'amssb')
```

and generate a spectrum analysis by the commands (replacing y by y1 and y4):

```
z = fft(y);
z = abs(z(1:length(z)/2+1));
frq = [0:length(zz)-1]*Fs/length(z)/2;
plot(frq, z);
```

You can also use the Simulink blocks to do the analysis.

SSB AM uses a Hilbert transform filter in the modulation process. The method is shown in the block diagram below:

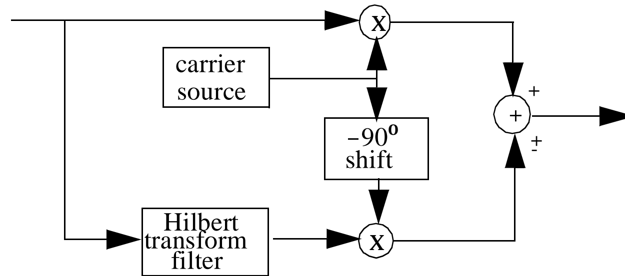


Figure 3-5: SSB AM Block Diagram

The carrier source is a sinusoidal signal at a given carrier frequency. The Hilbert transform filter has the transfer function

$$H(f) = -j\operatorname{sgn}(f)$$

which has the impulse response:

$$h(t) = \frac{1}{\pi t}$$

The Hilbert transform filter outputs a -90° phase shifted signal. The modulated signal is a lower sideband SSB version if the sign at the summation block is positive and an upper sideband SSB version if the sign at the summation block is negative. The toolbox provides MATLAB commands `hilbatf`, `hilbiir`, and `hilbana` for the Hilbert transfer filter design. Please refer to the "Transmitting and Receiving Filters" section in this chapter for a discussion of the algorithms of the Hilbert transform filter implementation.

The corresponding demodulation method, SSB-ADM, is the same as the method described in the first figure of the DSB-SC ADM demodulation.

A limitation of the SSB AM method is that you cannot use it to transmit a signal with a DC component.

Quadrature Amplitude Modulation and Demodulation

The quadrature multiplex double-sideband amplitude modulation (QAM) modulates an in-phase signal $m_I(t)$ and a quadrature signal $m_Q(t)$ using the formula

$$y(t) = m_I(t)\cos(2\pi f_c t + \phi_c) + m_Q(t)\sin(2\pi f_c t + \phi_c)$$

where $y(t)$ is the modulated signal, f_c is the carrier frequency (Hertz), and ϕ_c is the initial phase (rad). Because the sine and cosine signals are orthogonal, the original signals can be recovered later using demodulation techniques.

The picture below describes the QAM method:

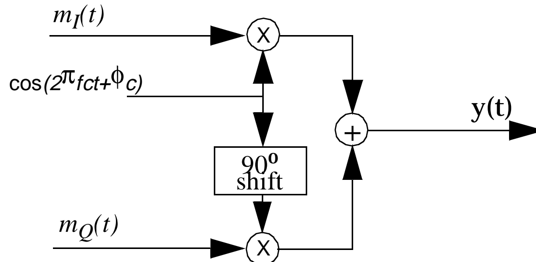


Figure 3-6: QAM Block Diagram

The corresponding demodulation method, QADM, recovers both the in-phase and quadrature signals, $m_I(t)$ and $m_Q(t)$ respectively, from the modulated signal $y(t)$. The demodulation procedure is shown in the block diagram below:

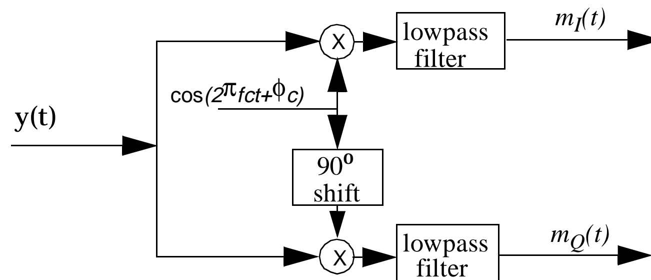
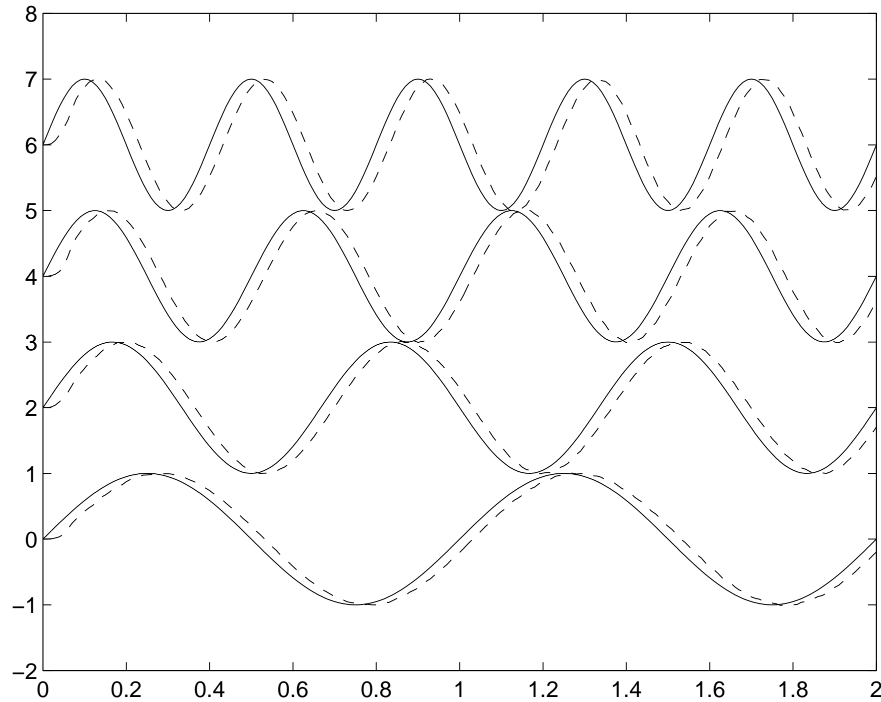


Figure 3-7: QADM Block Diagram

MATLAB QAM Example . Unlike the other formats in the amod and ademod functions, QAM requires that the message signal input be a matrix with an even number of columns. Each consecutive pair of columns are the in-phase and quadrature components of the input signal. The odd numbered columns are the in-phase signal components and the even numbered columns are the quadrature signal components. The code below performs a QAM modulation/ demodulation on a set of sinusoidal input signals:

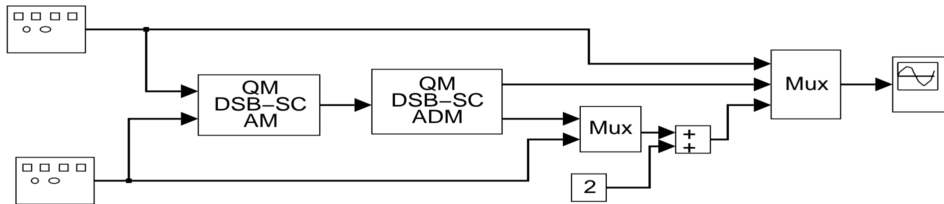
```
% The computation sampling frequency is 100 Hz.
Fs = 100;
% The carrier frequency is 15 Hz.
Fc = 15;
% The time vector.
t = [0:200]/100;
% Signal is a four column matrix.
% Each column is a sinusoidal signal, the frequencies
% of which are 1 Hz, 1.5 Hz, 2 Hz, 2.5 Hz respectively.
x = sin([2*pi*t',3*pi*t',4*pi*t',5*pi*t']);
% Use amod to modulate the signal with carrier frequency 15 Hz.
y = amod(x,Fc Fs,'qam');
% The demodulation process.
z = ademod(y,Fc,Fs,'qam');
% Plot the demodulated signal with vertical shifting.
plot(t,[x(:,1),x(:,2),x(:,3),x(:,4)],...
    t,[z(:,1),z(:,2),z(:,3),z(:,4)],'--');
```

The command `plot` generates this figure:

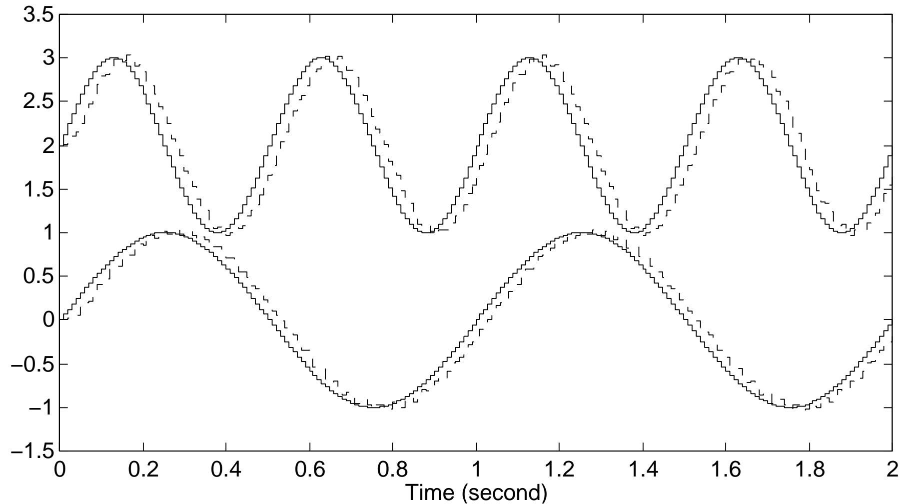


In the plot, the solid curves are the original signals. The dashed curves are the demodulated signals. There is a time shift between the original signal curves and the curves recovered by demodulation. This time lag is caused by the lowpass filters used in the demodulation process in the simulation.

Simulink QAM Example. This example compares the source signal to the recovered signal after the QAM modulation/demodulation process. You can run this example by typing `tutqam` on the command line:



The two signal generators on the left generate 1 and 2 Hertz sinusoidal signals respectively. After two seconds of simulation time, the plot from the scope shows the curve in the figure below:



The lowpass filters in the demodulation cause the time shift of the demodulated signals shown by the dashed curves.

Frequency Modulation and Demodulation

The frequency modulation (FM) modulates a message signal *m(t)* by varying the frequency of the output signal y(t) as a function of the amplitude of the input signal. The formula for the modulated signal is

$$y(t) = \cos(2\pi f_c t + 2\pi\theta(t) + \phi_c)$$

where *y(t)* is the modulated signal, f_c is the carrier frequency (Hertz), and ϕ_c is the initial phase (rad). θ(t) is the modulation phase, which changes with the amplitude of the input *m(t)*. The formula for θ(t) is

$$\theta(t) = k_c \int_0^t m(t)\,dt$$

where k_c is the *sensitivity factor*, which is a gain on the integrator output. The FM functionality is also known as a voltage controlled oscillator (VCO). The amplitude of the input signal voltage controls the oscillation frequency of the output signal. The demodulation process for a FM modulated signal uses a phase-locked loop method:

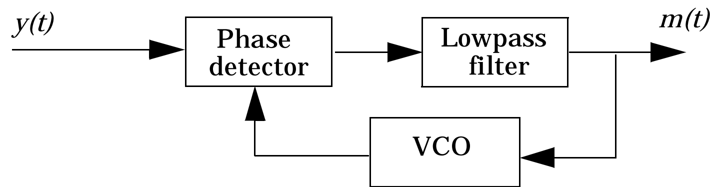


Figure 3-8: FM Demodulation Block Diagram

The feedback loop is known as a phase-locked loop, which recovers the correct phase from the received signal. The frequency demodulation uses a multiplier phase detector. The FM method implemented in this toolbox is a narrow band FM, which as its name implies, restricts the frequency modulation to a fixed bandwidth. For the differences between narrow and wideband FM, refer to Hambley.

Phase Modulation and Demodulation

Like FM, the phase modulation (PM) method is an angle modulation method. The amplitude of the input signal *m(t)* directly affects the phase of the output signal *y(t)* in the PM modulation process. The formula of the modulated signal is

$$y(t) = \cos(2\pi f_c t + 2\pi\theta(t) + \phi_c)$$

where $y(t)$ is the modulated signal, f_c is the carrier frequency (Hertz), and ϕ_c is the initial phase (rad). $\theta(t)$ is the modulation phase, which changes with the amplitude of the input $m(t)$. The formula for the change in phase is

$$\theta(t) = k_c m(t)$$

where k_c is the sensitivity factor. The PM demodulation uses the same phase-locked loop as that in the FM demodulation, but adds an integrator to the system after the PLL loop:

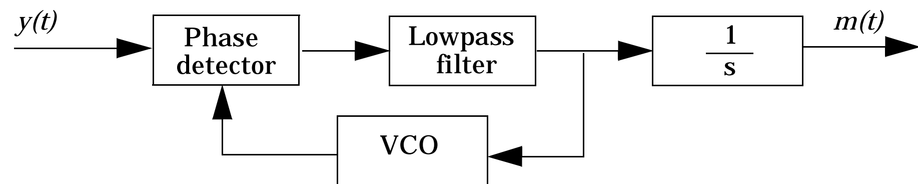


Figure 3-9: PM Demodulation Block Diagram

In the block diagram, 1/s indicates an integrator.

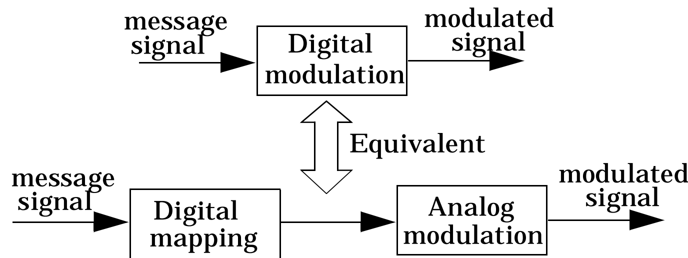
You can choose to use `method='pm'` in the `amod` and `ademod` commands for PM and its demodulation, or use the PM and PDM blocks in the modulation block sublibrary for Simulink simulation.

Passband Digital Modulation and Demodulation

This section discusses passband digital modulation and demodulation methods. The modulation/demodulation techniques discussed in the analog modulation/demodulation section of this chapter are relevant to the discussion of digital methods. Please read through the "Passband Analog Modulation and Demodulation" section to understand fully the discussion here.

Digital modulation includes two parts: digital to analog mapping and analog modulation. Digital to analog mapping techniques convert the received digital data into analog signals. Analog modulation then modulates the mapped

signals. The diagram below shows how digital modulation divides into a digital-to-analog mapping and an analog modulation:



Digital to analog mapping involves several key concepts. Typically, digital signals form a finite set of symbols. For example, in binary transmission it is possible to consider pairs of binary numbers as symbols. In this case, the symbol set is '00', '01', '10', and '11', making four distinct symbols. A digital mapping algorithm must have at least four mapping points to map the symbols uniquely. The number of points in the signal set is called the *M-ary number* in this toolbox. In the communications field, the M-ary number is also known as the alphabet size. A good choice in this example for the M-ary number is four. The mapping algorithm must assign a different set of analog signals for each mapping point; here a good choice might be $\sin(f_c 2\pi t) + \cos(f_c 2\pi t)$ for '00', $\sin(f_c 2\pi t) - \cos(f_c 2\pi t)$ for '01', and so on until the four possible mapping points are used. The arrangement of the signal set in the *signal space*, the space that digital mapping algorithm uses to map the mapping points, is called the *constellation*. The picture below shows the signal space of the mapping just described:

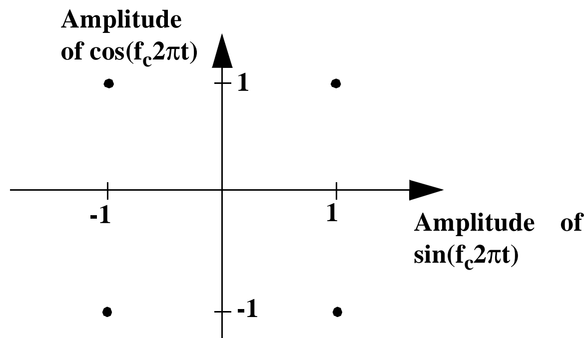
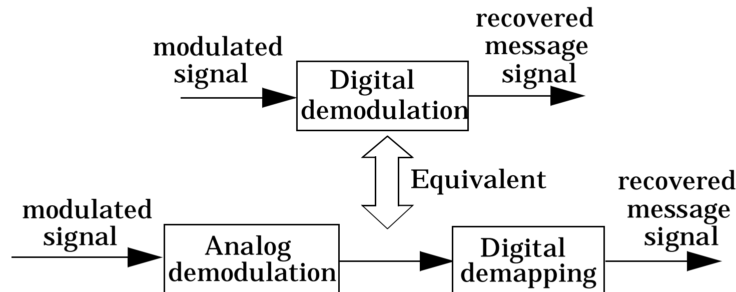


Figure 3-10: QAM Signal Space

This depicts a 4-ary square constellation for the digital mapping associated with quadrature amplitude modulation (QAM).

In general, digital demodulation is the inverse of the modulation process. A modulated signal is demodulated and then demapped from an analog signal to a digital signal. The diagram below shows how digital demodulation divides into an analog demodulation and an analog to digital demapping:



This demodulation structure applies to all digital demodulation methods except for multiple frequency shift keying (M-FSK) and multiple phase shift keying (M-PSK). These methods use correlation techniques.

Types of Digital Modulation/Demodulation

In the simulation of analog modulation, two different frequencies are involved: the carrier frequency (f_c) and the simulation sampling frequency (f_s). In digital modulation, there is one more frequency, the baud rate (f_d). To recover the message signal from the demodulation side correctly, the relationship between the three frequencies must be $f_s > f_c > f_d$.

The Simulink block library provides a digital modulation/demodulation sublibrary. The blocks in this sublibrary combine mapping and modulation functionality in each block. The sublibrary also combines the corresponding demodulation and demapping techniques. These combined blocks are for your convenience. The Communications Toolbox also provides separate digital modulation/demodulation and mapping/demapping sublibraries. These sublibraries separate the digital mapping/demapping blocks and analog modulation/demodulation blocks. This allows you direct access to all aspects of mapping/demapping and modulation/demodulation functionality.

In the MATLAB function library, the function dmod supports digital modulation. The corresponding digital demodulation is the function ddemod. The string variable method is a parameter in both functions. You can use method to specify the modulation and demodulation techniques used in the modulation/demodulation process. When the string '/nomap' is appended to the method string, the functions process the modulation and demodulation computations only, suppressing the mapping and demapping processes. You can then do mapping and demapping separately from the digital modulation/demodulation by using the functions modmap and demodmap, respectively.

This table lists the available strings for the method parameter and their meanings (for use with the functions dmod, ddemod, modmap, and demodmap):

Method	Meaning
ask	M-ary amplitude shift-keying modulation
qask	M-ary quadrature shift-keying modulation with square constellation
qask/cir	M-ary quadrature shift-keying modulation with circle constellation
qask/arb	M-ary quadrature shift-keying modulation with arbitrary constellation
fm	Frequency shift-keying modulation (coherent demodulation)
fm/noncoh	Frequency shift-keying non-coherent demodulation
pm	Phase shift-keying modulation
sample	A utility that supports modulation functions. sample performs up- or down-sampling the input signal.

This toolbox provides individual blocks in the Simulink block library for the same functionality.

The rest of this section discusses each of the modulation methods in the table, except for sample, which is a utility that supports the modulation methods. The discussion topics are divided into multiple amplitude shift keying (M-ASK),

quadrature amplitude shift keying (QASK), multiple frequency shift keying (M-FSK), and multiple phase shift keying (M-PSK).

M-ary Amplitude Shift-Keying Modulation

M-ary amplitude shift-keying (M-ASK) modulation includes two parts: M-ASK mapping followed by analog amplitude modulation. M-ASK mapping is a one dimensional coding process that maps the input digital symbols into real numbers in the interval [-x, x], where x is the specified maximum number. The input symbols are integers in the range [0, M-1], where M is the M-ary number. The outputs are signals with maximum amplitudes equal to assigned values between [-x, x]. The following figure depicts the constellation of M-ASK mapping for the M-ary numbers 2, 4, 8, and 16:

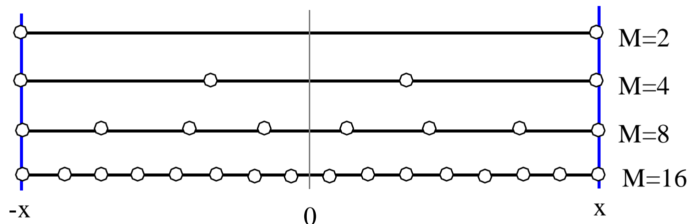


Figure 3-11: 2, 4, 8, and 16 Constellation M-ASK Mapping

This toolbox sets the M-ASK maximum value x equal to 1. In the figure, the circles along the lines are the mapping points. The number M must equal 2^K, where K is a positive integer. You can view the M-ASK constellation plot by using the command:

```
modmap('ask',M)
```

The Communications Toolbox gives you the option to do the mapping and modulation separately. You can do the mapping process by using the MATLAB function `modmap` or the Simulink MASK Map block. You can use the output from the mapping process as the input signal to the analog modulation function or block. For analog modulation, you can use the MATLAB function `amod` or the Simulink DSB SC-AM block located in the Passband Digital Modulation/Demodulation sublibrary. The methods for the modulation and demodulation processes must match to recover the signal correctly.

You can do the mapping and modulation together in this toolbox. The MATLAB function dmod computes both the mapping process and analog modulation process. The analog modulation method used in the function is the DSB-SC AM method. In Simulink you can use the MASK Mod block to perform both functions.

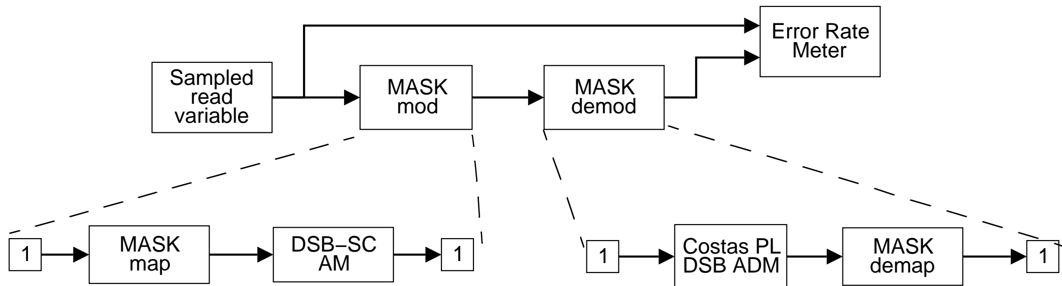
Demodulation is the inverse process of modulation. The demodulation process includes the digital demodulation and demapping. You can use either the MATLAB function demod or the Simulink MASK Demod block to simulate the digital demodulation and demapping together.

You can do the demodulation and demapping processes separately by using the MATLAB functions ddemod with the '/nomap' option and demodmap. In Simulink, use the DSB SC ADM and MASK Demap blocks for demodulation and demapping respectively.

MATLAB M-ASK Example. This example demonstrates the M-ASK modulation and demodulation MATLAB functions:

```
% Assign M-ary number M and calculation data length N.
M = 16; N = 200;
% Assign digital transfer frequency Fd, Carrier frequency Fc, and
% computation sampling frequency Fs.
Fd = 1; Fc = 10; Fs = 100;
% The message source
x = randint(N,1,M);
% Digital to analog signal mapping.
mp = modmap(x,Fd,Fs,'ask',M);
% Modulation
y1 = dmod(mp,Fc,Fd,Fs,'ask/nomap',M);
% Modulation with mapping
y2 = dmod(x,Fc,Fd,Fs,'ask',M);
% y1 and y2 are identical.
% Demodulation
z = ddemod(y1,Fc,Fd,Fs,'ask',M);
% Error rate
err = symerr(x,z)
err =
     0
```

Simulink M-ASK Example. This example uses the M-ASK Mod and Demod blocks located in the Digital Modulation/Demodulation sublibrary to build a simple digital modulation/demodulation example. The figure below depicts the block diagram structure of the M-ASK mod/demod example:



The Sampled Read Variable block generates the input message signal. The Error Rate Meter block compares the original message to the recovered message. The M-ASK Mod and M-ASK Demod blocks simulate the modulation and demodulation process. The M-ASK mod block is a masked block, which contains the M-ASK map block and DSB-SC AM block. The M-ASK Demod block is also a masked block, which contains the Costas PL DSB ADM block and the M-ASK Demap block.

M-ary Quadrature Shift-Keying Modulation

The M-QASK method is the most commonly used digital modulation method in communication systems. In general, an M-QASK process takes the input digital symbol and maps it into two independent components: in-phase and quadrature. An analog QAM method is used to modulate the in-phase and quadrature signals. On the receiving side, the signal is demodulated into in-phase and quadrature signals. A demapping process recovers the message signal by using the in-phase and quadrature signals.

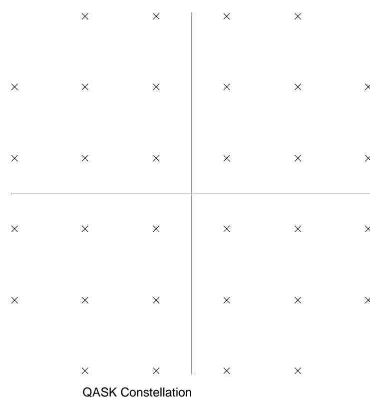
There are many mapping schemes available to map the input symbol to the in-phase and quadrature components. This toolbox provides three schemes for M-QASK constellations: square, circle, and user-defined arbitrary constellations. This section discusses each in turn.

Square Constellation

QASK with square constellation has a symmetric square constellation structure. The key variable in the square constellation is the M-ary number, M. In this toolbox, the square constellation is limited to be the result of power of 2, i.e. $M=2^K$, where K is a positive integer. Use the command

```
modmap('qask',M)
```

to plot a square constellation figure. The figure below is the constellation for M=64 generated by using the command `modmap`:



QASK Constellation

The result of the mapped data is in the interval [-x, x] for the in-phase component and [-y, y] for the quadrature component. The maximum value for the in-phase component x and the maximum value for the quadrature component y are listed in this table.

M-ary number (M)	Maximum In-phase (x)	Maximum Quadrature (y)
2	1	1
4	1	1
8	3	1
16	3	3
64	7	7

M-ary number (M)	Maximum In-phase (x)	Maximum Quadrature (y)
128	11	11
256	15	15

Circle Constellation

The Circle constellation is defined by specifying the in-phase and quadrature components of each point on each circle. The circle constellation uses three vectors to define the constellation: number of symbols in each circle (nic); radii for each circle (ric); and phase shift for each circle (pic). The three vectors have the same length. You can use either

```
dmod('qask/cir',nic,ric,pic)
```

or

```
modmap('qask/cir',nic,ric,pic)
```

to plot the constellation.

The three variables nic, ric, and pic define the constellation points on the (concentric) circles. nic defines the number of points on each of the circles, ric defines the radii of each of the circles, and pic defines the phase of every point on the circles. All three parameters are of equal length, and that length specifies the total number of circles in the constellation. On a given kth circle, there are nic(k) points, which distribute evenly on the circle. ric(k) defines the radius of the circle, and pic(k) specifies the phase value of each point on the circle in radians.

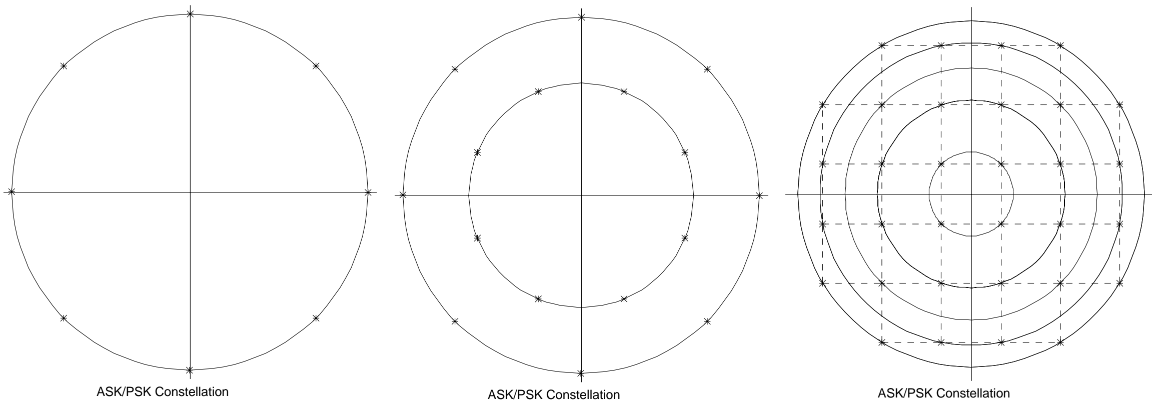
Circle Constellation Example. The following example uses the function modmap to generate constellation plots:

```
% Left plot - a one circle constellation with radius one.
modmap('qask/cir',8,1,0)
% Middle plot - a two circle constellation where
% each circle has 8 points.
modmap('qask/cir',[8 8],[.63 1],[pi/8 0])
```

A square constellation can be produced using the circle constellation format. The following lines generate a square constellation that is the same as the constellation given in the square constellation example:

```
% Right plot - a square constellation using the circle format.
ric = sqrt([2 10 10 18 26 26 34 34]);
modmap('qask/cir',4*ones(1,8),x,asin([1 1 3 3 1 5 3 5]./ric));
```

These plots depict the three constellations generated by the above code:



The left plot is simply a 8-PSK circle constellation. The middle plot is known as the optimal M-ary number 16 circle constellation. The right plot is a 32-square constellation. The *x*-axes in the plots are the in-phase components, and the *y*-axes are the quadrature components. The asterisks are the constellation points. The rectangular grid lines in the right plot have been added artificially to show the rectangular structure.

The elements in the vectors `nic` and `ric` must be positive. The mapping from the input integer to the components of the two axes must be one-to-one; the algorithm does not allow repeating points.

Arbitrary Constellation

The QASK with the user-defined arbitrary constellation has no limitation on the shape of the constellation structure, but the mapping must be one-to-one. You can define the points in any shape you like by specifying the in-phase and quadrature components of each point. You must define the constellation by specifying two vectors, the in-phase component vector (`inph`) and the

quadrature component vector (quad). The two vectors must be of equal vector length. The two elements inph(k) and quad(k) define a single point in the constellation. The elements in the two vectors are real numbers. In defining the arbitrary constellation, the in-phase and quadrature components mapping the input symbol k are inph(k+1) and quad(k+1) respectively.

You can use either

```
dmod('qask/arb',inph,quad)
```

or

```
modmap('qask/arb',inph,quad)
```

to plot the constellation.

The arbitrary constellation can represent both square and circle constellations. For example, you can generate the in-phase and quadrature component vectors used in defining the arbitrary constellation from the square constellation. The commands are:

```
y = modmap([0:M-1],1,1,'qask',M);
modmap('qask/arb',y(:,1),y(:,2));
```

When M=32, the figure generated is the same as the plot shown in the "Square Constellation" section.

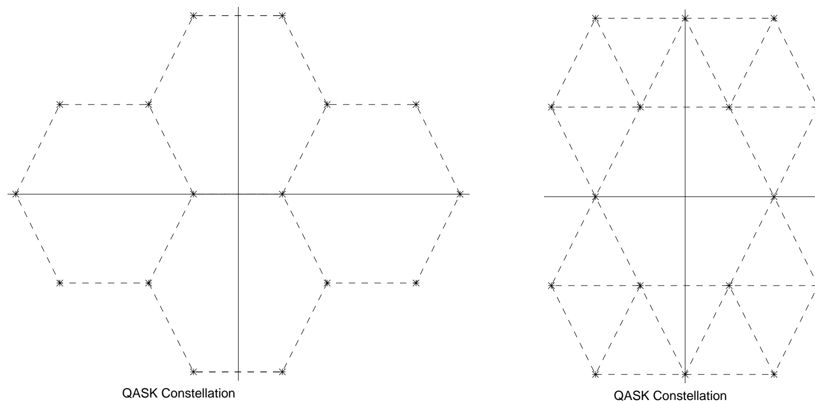
You can define any constellation structure by using a user-defined arbitrary constellation. For example, you can define a 16-ary hexagonal structure constellation by defining and plotting the constellation using these commands.

```
% Left-hand plot, a hexagonal constellation.
inph = [1/2 1 1 1/2 1/2 2 2 5/2];
quad = [0 1 -1 2 -2 1 -1 0];
inph = [inph;-inph]; inph = inph(:);
quad = [quad;quad]; quad = quad(:);
modmap('qask/arb',inph,quad)
```

Similarly, you can define a triangle structure by assigning the constellation in-phase and quadrature points as in the code below:

```
% Right-hand plot, a triangle constellation.
inph = [1/2 -1/2 1 0 3/2 -3/2 1 -1];
quad = [1 1 0 2 1 1 2 2];
inph = [inph;-inph]; inph = inph(:);
quad = [quad;-quad]; quad = quad(:);
modmap('qask/arb',inph,quad)
```

The figure below shows the plots of the hexagonal and triangle constellations:.



The left plot is the hexagonal constellation, and the right plot is the triangle constellation. The connection lines have been added artificially to show the hexagonal and triangle structures. You can create any constellation structure by using similar commands.

QASK Modulation/Demodulation Example. This example demonstrates the quadrature amplitude shift keying (QASK) modulation/demodulation method. Given the M-ary number is 16, a good choice for modulation and mapping is to use four-bit messages mapped into a square constellation.

Using the toolbox, you have the choice of doing the mapping and analog modulation separately using the MATLAB functions `modmap` and `amod` or

combining them together using amod alone. In this example, we'll do them separately:

```
% Number of data points
N = 20;
% Random integers
x = randint(N,1,16);
% 16-QASK mapping with Fd=1, Fs=100.
m = modmap(x,1,100,'qask',16);
% QAM modulation with Fc=10.
y = amod(m,10,100,'qam');
% demodulation with eye pattern diagram plot.
z = ddemod(y,10,1,100,'qask/eye',16);
```

The original signal is mapped into in-phase and quadrature components in the digital to analog mapping. The mapped signal is modulated using the QAM modulation method. You can then recover the original digital signal by using the QASK demodulation method.

Since $f_s > f_d$, in the digital demodulation process, the digital demapping process takes a sampling point from the analog demodulated signal. This sampling point is known as the decision point. As discussed in the "Signal Generators and Display Devices" section in this chapter, an eye-pattern diagram is commonly used to determine the decision point. This toolbox provides the MATLAB function eyescat and, in Simulink, the Eye Pattern Scatter block in the Source/Sink sublibrary to help to find the decision point, which should be the point where the "eye" is most widely open. In the digital demodulation and demapping blocks, the decision point is set in the parameter offset of the sample time variable. In the ddemod and demodmap functions, it is set in the offset in the digital frequency variable.

The demodulation in this example uses the eye-pattern diagram to find the decision point. The simulation process can be demonstrated by the data flow in the following figure.

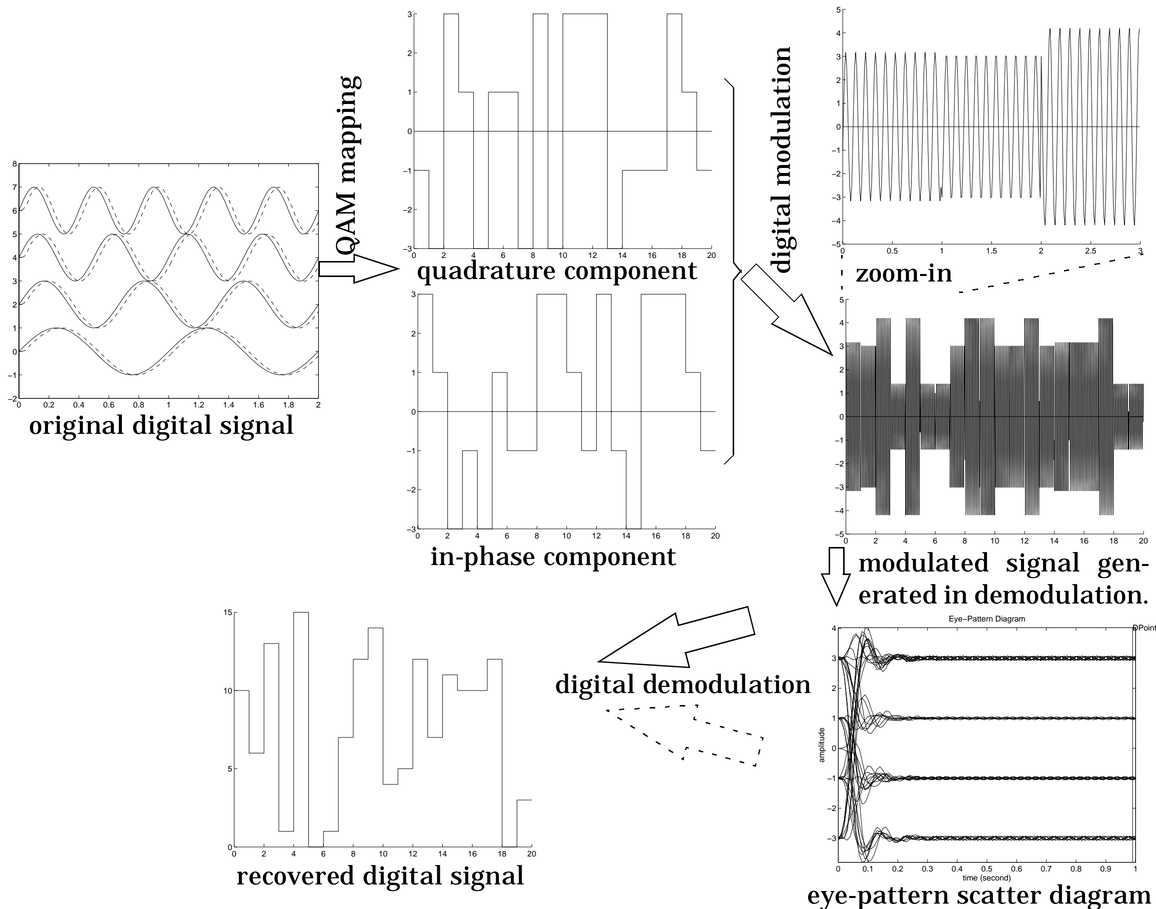


Figure 3-12: QAM Digital Signal Recovery Using an Eye-Pattern Scatter Diagram

In the figure, the *x*-axis for every plot is time (in seconds). The *y*-axis is the signal level. The arrows show the data flow in the simulation.

M-ary Frequency Shift-Keying Modulation

M-FSK modulation modulates a digital signal by changing the frequency of the output signal depending on the value of the input signal. Using M-FSK, you need to specify the *tone space* as an input parameter. The tone space is the

frequency separation between two consecutive frequencies in the modulated signal. The M-FSK modulation process divides into two parts: mapping and analog modulation. The mapping process maps the input symbol into the value of the frequency shift from the carrier frequency, and the analog modulation is analog FM. If the carrier frequency is `Fc`, M-ary number `M`, and the tone space `Δf`, then the frequency range of a modulated signal is in the range `[Fc, Fc+(M-1)*Δf]`. You can use the MATLAB function `modmap` for M-FSK mapping and the function `dmod` for the modulation. You can also use the Simulink blocks in the digital modulation/demodulation sublibrary for the modulation.

The demodulation for the M-FSK is different from the other digital demodulation methods discussed in this section. The demodulation process uses a length `M` vector signal where the frequency of the ith element in the vector signal matches the modulated signal when the input symbol is `i`. The demodulation process computes the correlation values between the signal array and the received signal and, after calculating the maximum correlation value, decides what symbol was most likely transmitted. Like other digital demodulation methods, M-FSK demodulation requires the specification of a decision point.

There are two different methods to compute the correlation value: the coherent and noncoherent methods. Using the coherent method, you must know the phase information of the modulated signal from the receiving side. The noncoherent method does not require phase information; it recovers the phase of the modulated signal during the demodulation. However, the noncoherent demodulation method is more computationally complex than the coherent demodulation method.

Coherent Demodulation

This figure depicts the coherent M-FSK demodulation method:

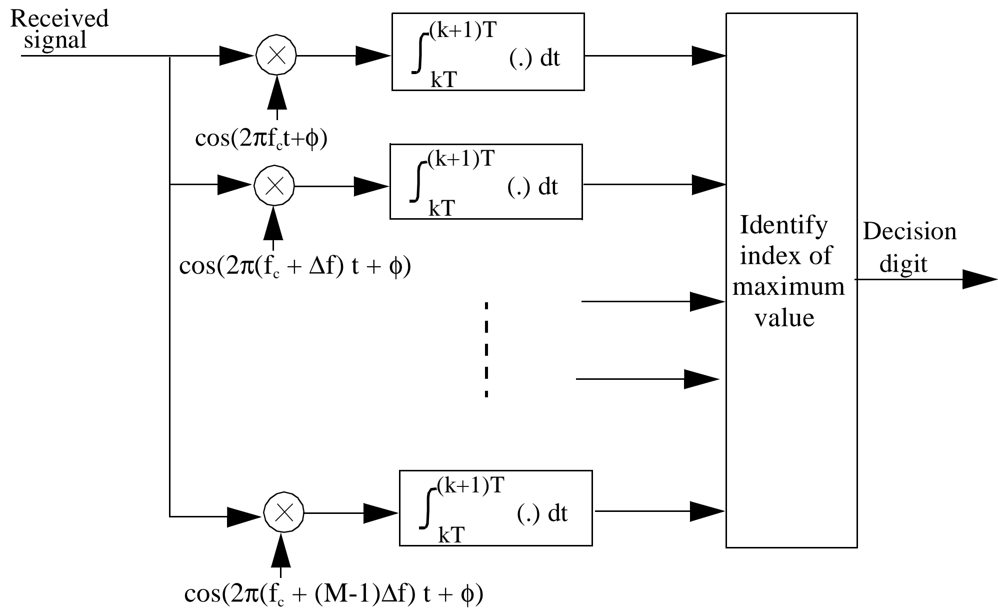


Figure 3-13: Coherent M-FSK Demodulation Block Diagram

In the figure, Fc is the carrier frequency, Δf is the tone space, and ϕ is the initial phase in the modulated signal. T is the symbol interval; alternatively, 1/T is the digital transmission frequency. M is the M-ary number in the modulation. The signal $\cos(2\pi(f_c+(I-1)\Delta f)t+\phi)$ is the modulated signal for symbol I. The integrations depicted in the boxes are the correlation computations. In the coherent correlation M-FSK demodulation, each integrator correlates the input signal with a specified frequency and phase. The output of the correlators is a scalar index. The index with the maximum correlation value is identified as the demodulation result.

Noncoherent Demodulation

Noncoherent M-FSK demodulation calculates the correlation value between the input signal and a vector of sinusoidal signals. Each sinusoidal signal has its frequency set to one of the possible messages in the signal set. Unlike the coherent correlation method, this method is insensitive to the initial phase of the modulation signal. The trade-off is that the calculation of the noncoherent correlation is more complicated since the noncoherent method must recover the

phase of the modulated signal. The figure below shows the noncoherent M-FSK demodulation method:

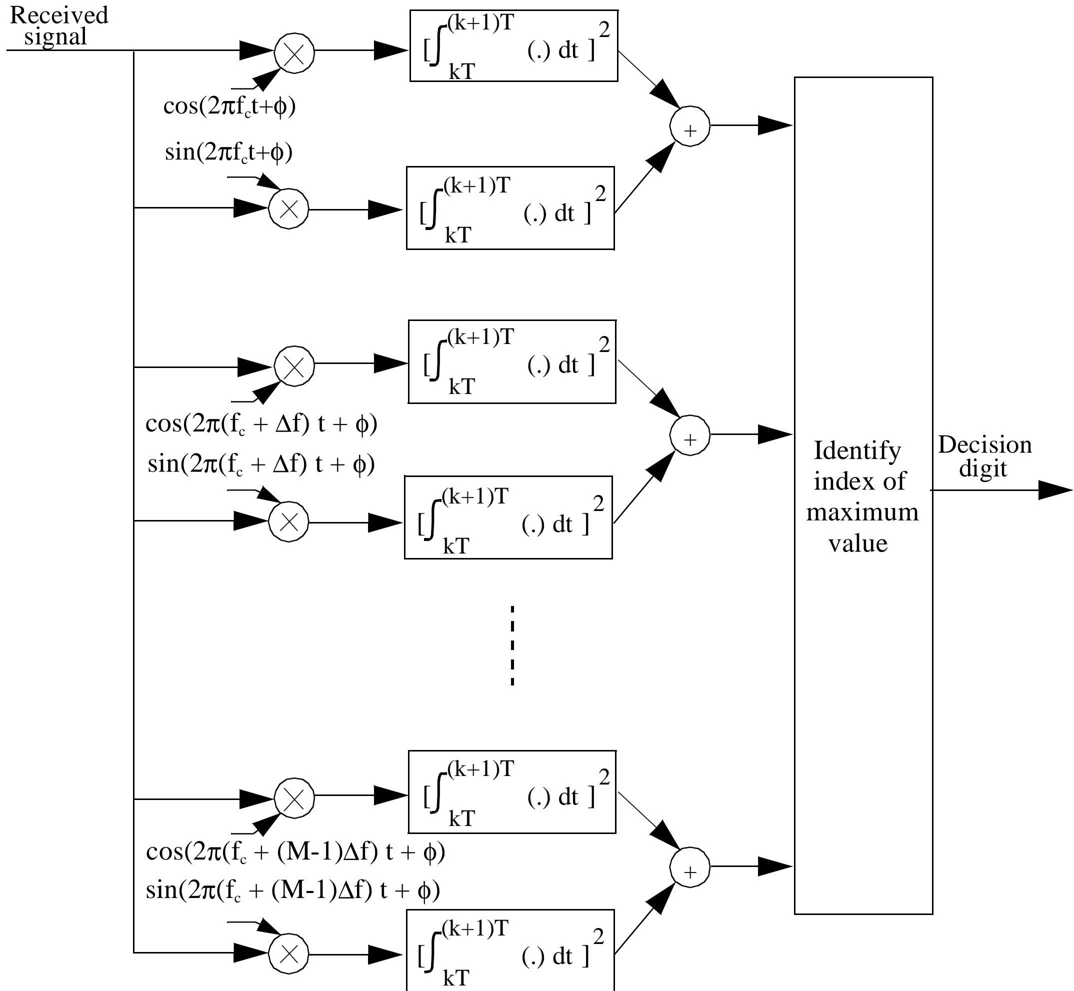


Figure 3-14: Noncoherent M-FSK Demodulation Block Diagram

The symbols used in the figure are the same as the symbols used in the figure for the coherent M-FSK method.

Simulink Example

The following Simulink block diagram gives an example of M-FSK modulation using both M-FSK coherent and noncoherent demodulation. Each of the methods is a combination of two blocks:

- M-FSK modulation is a combination of the M-FSK map block and the FM block.
- M-FSK coherent demodulation is a combination of the Coherent M-FSK Demodulation Correlation block and the Min/Max demap block.
- M-FSK noncoherent demodulation is a combination of the Noncoherent M-FSK demodulation correlation block and the Min/Max demap block.

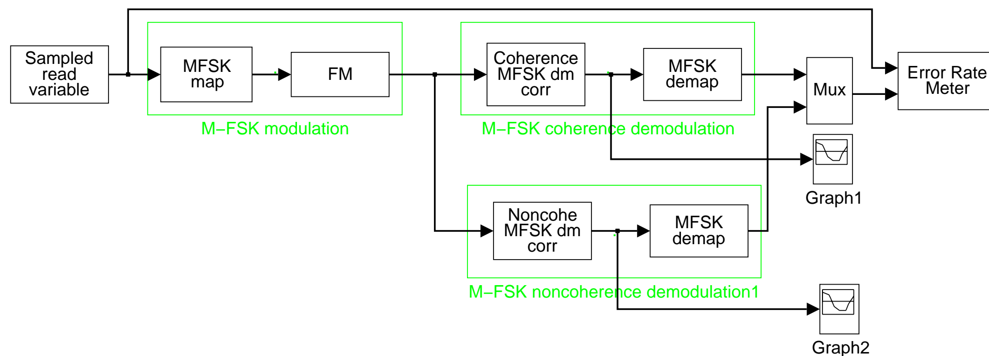
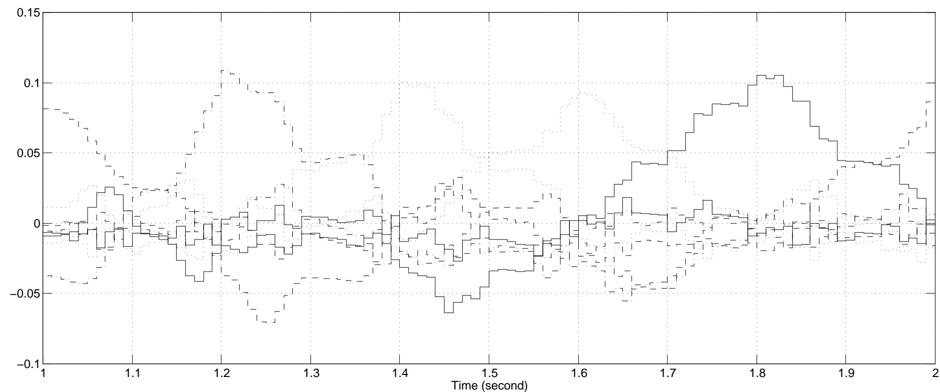


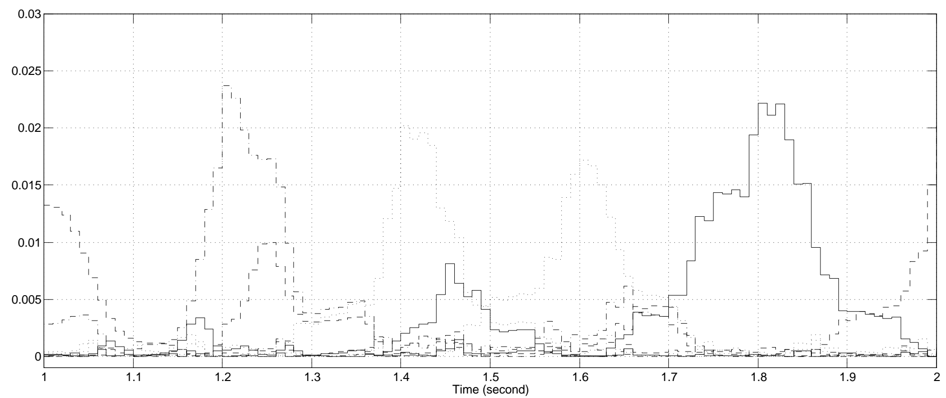
Figure 3-15: Simulink Block Diagram of M-FSK Modulation

The Sampled Read Variable block generates the input symbols. The Error Rate Meter block compares the original message signal and the signals recovered from the demodulation. The Graph1 scope block shows the values of the

correlation computation for the coherent correlation calculation. The result is shown in the plot below:



The Graph2 scope block shows the values of the correlation computation for the noncoherent correlation calculation. The next figure shows a snap shot from the scope block for the noncoherent M-FSK demodulation correlation computation:



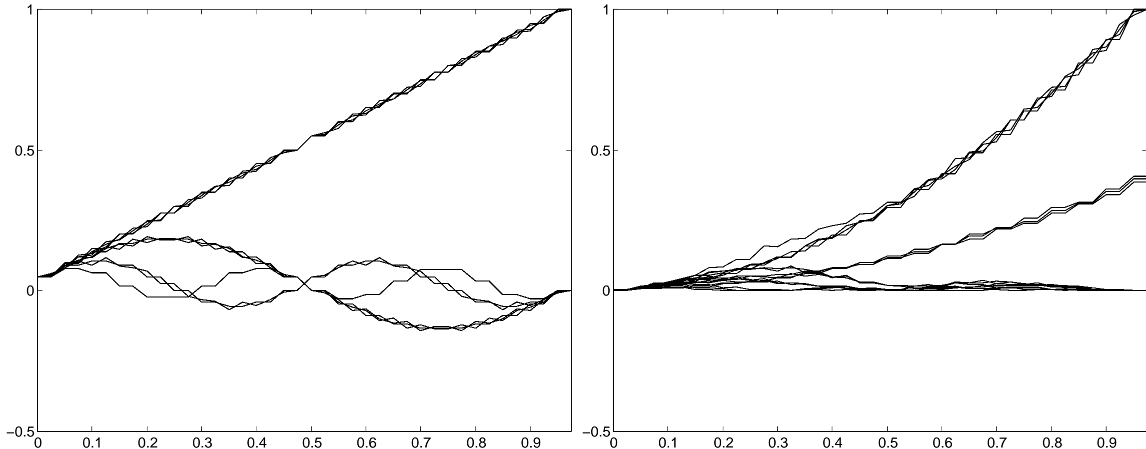
In the Min/Max Demap block, you can set the offset in the demapping. Usually, a communications engineer sets the offset the same way as the decision point in the M-QASK demodulation processing, which means that it is set to the time point where the single maximum value is most widely separated from the other values. This is similar to the method used in determining the decision point in the eye-pattern diagram.

MATLAB Example

You can do the same simulation described in the Simulink example using MATLAB functions. The command lines are:

```
% Define the frequency and tone space.
Fc = 10; Fs = 40; Fd = 1; df = 1;
% Assume M-ary number is 4.
M = 4;
% Generate the message signal.
x = randint(100,1,M);
% Modulation
y = dmod(x,Fc,Fd,Fs,'fsk',M,df);
% Coherent M-FSK demodulation with plot.
z1 = ddemod(y,Fc,Fd,Fs,'fsk/eye',M,df);
% Noncoherent M-FSK demodulation with plot.
z2 = ddemod(y,Fc,Fd,Fs,'fsk/eye/nocoh',M,df);
% The error rate
symerr(x,z1)
ans =
     0
symerr(x,z2)
ans =
     0
```

In the above code, the `ddemod` commands append the option `'/eye'` to the `method` string. This option generates the plot of correlation values. The figure below compares the plots of coherent and noncoherent M-FSK demodulation:



The plot generated from the coherent correlation M-FSK demodulation is shown on the left-hand side, and the plot generated from the noncoherent correlation M-FSK demodulation is shown on the right. Note that the curves in the eye pattern plots are normalized correlation values; the plots are not the conventional eye-pattern plots. You can use either of these plots to determine the offset value.

M-ary Phase Shift-Keying Modulation

M-PSK modulation modulates a digital signal by changing the phase values in the modulated output signal. M-PSK distinguishes between the digital messages by setting different initial phase shifts in the modulation. The digital input signals to a M-PSK modulator are in the range [0, M-1], where M is the M-ary number. The phase shift for input digit i is $2\pi i/M$.

The structure of the demodulator for M-PSK is similar to that of M-FSK. The demodulation process calculates the correlation value between the input signal and a vector of carrier frequency sinusoidal signal. Each sinusoidal signal in

the vector has its phase set to a possible result from the signal set. The figure below shows the M-PSK demodulation method:

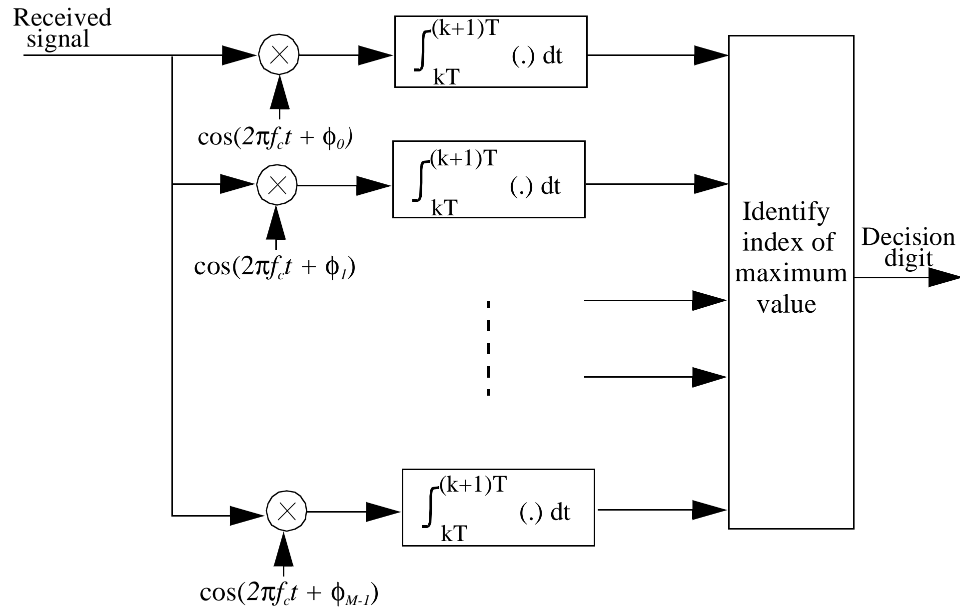


Figure 3-16: M-PSK Demodulation Block Diagram

In the figure, f_c is the carrier frequency, ϕ_i is the initial phase for digit i, and T is the symbol period, meaning that $1/T$ is the baud rate. Note that the M-PSK demodulation is a coherent correlation computation; you must provide the phase information.

Baseband Modulation and Demodulation

The Nyquist sampling theorem requires that a simulation sampling frequency must be at least twice the frequency f_c being modeled. In passband simulation, the simulation models the carrier frequency. Since the carrier frequency is usually a high frequency signal, modeling passband communication systems involves high computational loads. To alleviate this problem, baseband simulation techniques are used.

Baseband simulation, also known as the low-pass equivalent method, uses the complex envelope of a passband signal. As an example, consider analog quadrature amplitude modulation. Assume the carrier frequency is f_c and the initial phase is θ_0. The equation for modulating the in-phase signal $x_I(t)$ and the quadrature signal $x_Q(t)$ by the carrier frequency is

$$y(t) = x_I(t)\cos(2\pi f_c t + \theta_0) - x_Q(t)\sin(2\pi f_c t + \theta_0)$$

which is equivalent to

$$y(t) = \left\{Re(x_I(t) + jx_Q(t))e^{j\theta_0}e^{j2\pi f_c t}\right\} = Re\{Y(t)e^{j2\pi f_c t}\}$$

where

$$Y(t) = (x_I(t) + jx_Q(t))e^{j\theta_0}$$

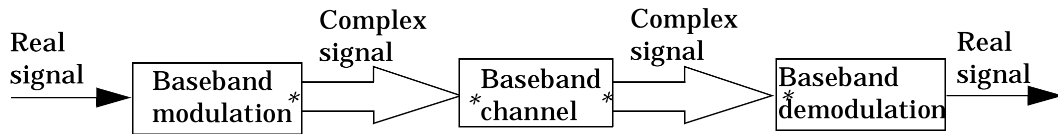
is called the *complex baseband waveform*. The high frequency component is:

$$e^{j\theta_0}$$

Baseband simulations model the complex baseband waveform $Y(t)$ only.

Let B be the bandwidth of the message signal. The baseband simulation requires the simulation sampling rate to be larger than $2B$. In the baseband simulation, the signal bandwidth is always assumed to be much smaller than the carrier frequency, or $B << f_c$.

The output of a baseband modulator and the input to a baseband demodulator are complex signals. The figure below shows the signal data type at each step of the baseband modulation/demodulation process:



To demodulate a baseband modulated signal, a baseband demodulator must be used. There is a difference between the way MATLAB and Simulink handle the data generated in a baseband modulation/demodulation simulation. In MATLAB functions, the output data type of a baseband modulated signal is a complex number. In Simulink blocks, all data is real, so complex data is represented by using a two-dimensional vector where each component is a real scalar. The first element in the vector represents the real part of the complex number, and the second element represents the imaginary part. In Simulink blocks, all output ports that output complex signals are marked with an asterisk. All input ports that expect complex input signals are also marked with an asterisk.

The following sections discuss in turn:

- Baseband analog modulation
- Baseband analog demodulation
- Baseband digital modulation
- Baseband digital demodulation

Baseband Analog Modulation

This section discusses baseband analog modulation techniques. This toolbox supports six baseband analog modulation techniques:

- Double sideband-suppressed carrier analog modulation (DSB-SC AM)
- Single sideband-suppressed carrier analog modulation (SSB-SC AM)
- Double sideband-carrier analog modulation (DSB-TC AM)
- Quadrature amplitude modulation (QAM)
- Frequency modulation (FM)
- Phase modulation (PM)

These methods are similar to their passband counterparts. Refer to the discussion in the “Passband Analog Modulation and Demodulation” section of this chapter for descriptions of the passband modulation techniques.

In the baseband analog modulation techniques provided in this toolbox, the output divides into in-phase and quadrature components. The input to the modulator is a real signal *u(t)*, except for quadrature amplitude modulation, where the input signal has two elements, the in-phase component and the quadrature component, $u_I(t)$ and $u_Q(t)$ respectively This table lists the real part of the modulation output $x_I(t)$ and the imaginary part $x_Q(t)$ for each modulation method:

Modulation Method	$x_I(t)$	$x_Q(t)$	Optional Parameters
DSB-SC AM	*u(t)*	*0*	
SSB-SC AM	*u(t)*	$\hat{u}(t)$, Hilbert transform of *u(t)*.	
DSB-TC AM	*u(t)-k*	*0*	*k* is modulation offset.
QAM	$u_I(t)$	$u_Q(t)$	
FM	$\cos\left(k\int u(\tau)d\tau\right)$	$\sin\left(k\int u(\tau)d\tau\right)$	*k* is modulation sensitivity
PM	$\cos(ku(t))$	$\sin(ku(t))$	*k* is modulation sensitivity

$x_I(t)$ and $x_Q(t)$ are the outputs when the modulation phase shift is zero. The modulated signal is represented in its complex envelope form:

$$Y(t) = (x_I(t) + jx_Q(t))e^{j\theta} = R(t)e^{j\psi(t)}$$

where θ is the carrier phase shift, *R(t)* is the real envelope, which is $\sqrt{x_I(t) + x_Q(t)}$; and ψ is the phase of the envelope:

$$\psi = \theta + \operatorname{atan}\frac{x_I(t)}{x_Q(t)}$$

The complex signal *Y(t)* is the output of both the MATLAB functions and the Simulink blocks in baseband simulation.

Baseband Analog Demodulation

This section discusses the baseband analog demodulation of the modulated complex signal *Y(t)* defined in the last section.

Y(t) may be distorted by noise, bandwidth limitations, or other unwanted effects in the transmission channel. This distorted signal is represented by *Z(t)*. The demodulation must recover the message signal *u(t)* ($u_I(t)$ and $u_Q(t)$ for QAM) from the received complex signal *Z(t)*. Demodulation can be divided into two methods: *coherent* and *noncoherent* demodulation. Coherent demodulation requires exact knowledge of the frequency and phase of the carrier frequency to recover the signal correctly. Noncoherent demodulation permits the recovery of the message signal without exact knowledge of the carrier signal's frequency and phase, but this method adds computational complexity and performance degradation. The noncoherent demodulation method is not available for every modulation scheme; the coherent demodulation method is always available.

Coherent Demodulation

This equation represents the received signal Z(t) by its in-phase and quadrature components:

$$z_I(t) + jz_Q(t) = Z(t)e^{j2\pi(f_c - \hat{f}_c) + j(\theta - \theta)}$$

In this equation, $\hat{f}_c$ is the received carrier signal frequency and θ is its phase. f_c and θ are the transmitted carrier signal frequency and phase. The signal *u(t)* ($u_I(t)$ and $u_Q(t)$ for QAM) can be recovered by referring to the table given in the "Baseband Analog Modulation" section of this chapter using the estimated signal $z_I(t)$ and $z_Q(t)$. The demodulation phase shift θ and the carrier frequency estimation error $f_c - f_c$ can be calculated by using the phase locked loop (PLL)

technique. The “Synchronization” section of this chapter provides a detailed discussion of the PLL technique.

Noncoherent Demodulation

The noncoherent demodulation scheme can be used with the DSB-SC AM, DSB-TC AM, FM, and PM methods to detect a modulated signal.

For an AM with carrier signal, an envelope detection method can be used for noncoherent demodulation. This method detects the signal in the form:

$$\hat{x}(t) = |Z(t)| - k$$

When the received signal is represented in polar form $Z(t) = R(t)e^{j\psi(t)}$, the detected PM signal is simply $\hat{x}(t) = \psi(t)$ and the FM signal is the derivative of the PM signal:

$$\hat{x}(t) = \frac{d}{dt}\psi(t)$$

The baseband waveform PLL simulation method can also be used for noncoherent demodulation. The method is discussed in the “Passband Analog Modulation and Demodulation” section of this chapter. For discussions of the FM and PM methods, refer to the “Passband Frequency Modulation and Demodulation” and “Passband Phase Modulation and Demodulation” sections of this chapter.

Baseband Digital Modulation

As discussed in the “Passband Digital Modulation and Demodulation” section of this chapter, the functionality of digital modulation can be divided into two parts: the mapping from digital signal to analog signal, and analog modulation. The “Passband Digital Modulation and Demodulation” section of this chapter discusses digital mapping methods.

In the rest of this section, the mapping and modulation processes are analyzed together as one method. Assume the input message has M-ary number M. The input is an integer in the range [0, M-1]. Most digital modulation methods can

use the baseband simulation formulas given in the “Baseband Analog Modulation” section:

$$y(t) = x_1(t)\cos(2\pi f_c t + \theta) - x_2(t)\sin(2\pi f_c t + \theta)$$

$$Y(t) = (x_1(t) + jx_2(t))e^{j\theta} = R(t)e^{j\psi(t)}$$

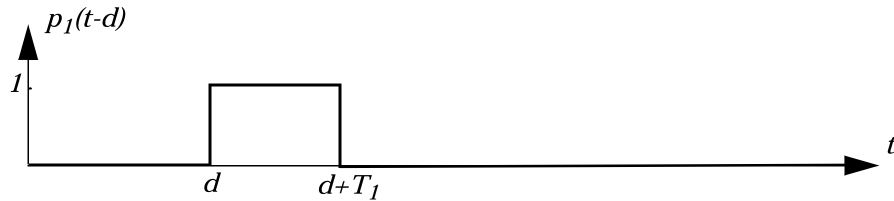
The sampling frequencies for $x_1(t)$ and $x_2(t)$ are $1/T_1$ and $1/T_2$ respectively.There may be delays in the transfer times for $x_1(t)$ and $x_2(t)$; these delays are represented by D_1 and D_2 respectively.

There is a mathematical representation for signals $x_1(t)$ and $x_2(t)$ that allows for a unified approach to several important digital modulation schemes. Let function $p_1(t)$ be a pulse function with pulse length T_1. The equation for $p_1(t)$ is

$$p_1(t) = u(t) - u(t - T_1)$$

where $u(t)$ is the unit step function.

In other words, $p_1(t)$ is nonzero in a T_1 length time frame and zero elsewhere:



Similarly, a pulse function $p_2(t)$ of length T_2 can be defined. The real and the imaginary components, $x_1(t)$ and $x_2(t)$, are:

$$x_1(t) = \sum_k A_k p_1(t - kT_1 - D_1)$$

$$x_2(t) = \sum_k B_k p_2(t - kT_2 - D_2)$$

This general formula allows us to specify M-ary ASK, M-ary QASK with square or arbitrary constellations, and M-ary PSK using only the parameters A_k, B_k,

T_1, T_2, D_1, and D_2. For more information about this mathematical representation, refer to Jeruchim, Balaban, and Shanmugan, *Simulation of Communication Systems,* pp. 328-9.

$X(t) = x_1(t) + j\,x_2(t)$ is the modulated signal with a zero initial phase shift in the modulation. This table lists mathematical formulae for A_k, B_k, $p_1(t)$, and $p_2(t)$:

Modulation Method	A_k	B_k	$P_1(t)$	$P_2(t)$	**Parameters**
M-ary ASK	$\left(n-\frac{M-1}{2}\right)\frac{2A_c}{M-1}$	*0*	$1;\ 0 \le t \le T_1$	*0*	A_c: peak amplitude.
M-ary QASK with square constellation	$s_1(n+1)$	$s_2(n+1)$	$1;\ 0 \le t \le T_1$	$p_1(t)$	s_1 and s_2 are output from command `[s1, s2] = qaskenco([0:M-1], M);`
M-ary QASK with arbitrary constellation	*Re(inph(n+1))*	*Im(quad(n+1))*	$1;\ 0 \le t \le T_1$	$p_1(t)$	*inph* and *quad* are length M vectors defining the in-phase component and quadrature component of the arbitrary constellation.
M-ary PSK	$\cos(2\pi n/M)$	$\sin(2\pi n/M)$	$1;\ 0 \le t \le T_1$	$p_1(t)$	$T_1=T_2$, $D_1=D_2$.

In the table, *n* is the integer information to be transferred in the time frame, which is in the range of [0, M-1]. The table doesn't list parameters for QASK with circle constellation. This is because circle constellations can be viewed as a special case of the arbitrary constellation method. Given the vectors: `nic` (number in circle), `ric` (radii in circle), and `pic` (phase in circle), you can obtain

the in-phase component inph and quadrature component quad by using the commands:

```
z = apkconst(nic, ric, pic);
inph = real(z);
quad = imag(z);
```

The modulated signal represented in terms of the modulation phase shift θ is:

$$Y(t) = X(t)e^{j\theta}$$

M-ary frequency shift-keying (M-FSK) and minimum shift-keying (MSK) do not fit the mathematical model described above. These two modulation methods are known as *continuous phase shift-keying* methods. Some digital modulation methods cannot be directly represented by the formula as presented above. Another general form modulated signal is given by:

$$y(t) = A\cos(2\pi f_c t + \theta_0 + \phi(t))$$

The complex envelope formula is given by:

$$Y(t) = Ae^{j\theta_0 + j\phi(t)}$$

For M-FSK, $\phi(t)$ is a variable changing with the value of the digital input. The value of $\phi(t)$ is

$$\phi(t) = \pi h\left(n_k\frac{(t-kT)}{T} + \sum_{i<k} n_i\right) \qquad kT \le t \le (k+1)T$$

where n_k is the input integer at computation step k. Variable h is the tone space, which is the minimum frequency difference between two modulated consecutive different symbols.

If M=2 and h=1/2, this is a special case of FSK known as MSK, which is a widely used method. In this case, the phase function $\phi(t)$ becomes.

$$\phi(t) = \frac{\pi}{2}\left(n_k\frac{(t-kT)}{T} + \sum_{i<k} n_i\right) \qquad kT \le t \le (k+1)T$$

Baseband Digital Demodulation

In general, the digital demodulation process can be divided into two parts: the analog demodulation and the digital demapping, except in the case of M-ary frequency shift-keying (M-FSK) and M-ary phase shift-keying (M-PSK), where a correlation computation method is used. The methods used in the digital demodulation for baseband simulation are discussed in previous sections.

Similar to the baseband analog demodulation methods, in the baseband simulation demodulation process, you need to specify the decision point for demapping. To match the formula described in the last section, you need to determine D_i, i = 1, 2, from the receiving side. The demodulation process is the reverse of the modulation process. Using the received signal $Z(t) = z_I(t) + j\,z_Q(t)$, which is the modulated signal Y(t) passed through a transmitting channel, the demodulation recovers the transmitted signals $s_1(t)$ and $s_2(t)$.

The baseband simulation M-FSK demodulation and M-PSK demodulation algorithms are the same as those used in passband simulation methods in which correlation computations are used. Please refer the discussion in the "Analog Modulation and Demodulation" section.

Using the Baseband Simulation Functionality in the Toolbox

This toolbox provides baseband simulation MATLAB functions as well as Simulink blocks in parallel with the passband simulation. In the Simulink Modulation/Demodulation sublibrary, there are baseband modulation/ demodulation sublibraries next to every passband modulation/demodulation

sublibrary. In the MATLAB function library, there is a baseband simulation function for every passband modulation/demodulation function as listed below:

Baseband Function	Passband Function	Functionality
amod	amodce	Analog modulation
ademod	ademod	Analog demodulation
dmod	ddemod	Digital modulation
ddemod	ddemod	Digital demodulation
modmap	modmap	Digital mapping
demodmap	demodmap	Digital demapping

The choices for the analog modulation/demodulation are amdsb-sc, amdsb-tc, amssb, qam, fm, and pm. The choices for the digital modulation/demodulation and digital mapping/demapping methods are ask, qask, qask/circle, qask/arb, fsk, psk, and sample. sample, which changes the sampling rate of the input signal, is a utility that supports modulation techniques.

In baseband simulation, there is no carrier frequency involved. Therefore, the computation sampling frequency for baseband simulation can be much lower than that in a passband simulation. The only frequency used in baseband analog modulation is the computation sampling frequency fs. By Nyquist sample theory, fs should be at least twice as large as the bandwidth of the input signal. In baseband digital simulation, there are two frequencies involved: the computation sampling frequency fs and the digital transferring frequency fd. fs must be larger than fd, and usually fs/fd is a positive integer.

The passband demodulation discussion emphasizes the importance of the lowpass filters. The lowpass filters also apply to baseband simulation. Because the computation sampling frequency is different between baseband and passband simulation, the effect of the lowpass filters on demodulation is also different. If the effect of lowpass filters isn't important in your application, you can set both the numerator and denominator of the lowpass filter to one.

Baseband Simulation Example

This example compares baseband simulation results by using Simulink block diagrams and MATLAB functions. The following Simulink block diagram includes the available baseband modulation/demodulation methods. The modulations methods included in the simulation are: DBS-SC AM, DBS-TC AM, FM, PM, SSB AM, and QAM. In the block diagram, two Signal Generator blocks are used as signal source. The first Signal Generator generates a 1 Hz sinusoidal signal. The second Signal Generator generates a 2 Hz sawtooth signal. The second signal source is used as the quadrature component for QAM only. The computation sample frequency set in the simulation is 100 Hz. This figure shows the Simulink blocks used to compare the available baseband modulation/demodulation methods:

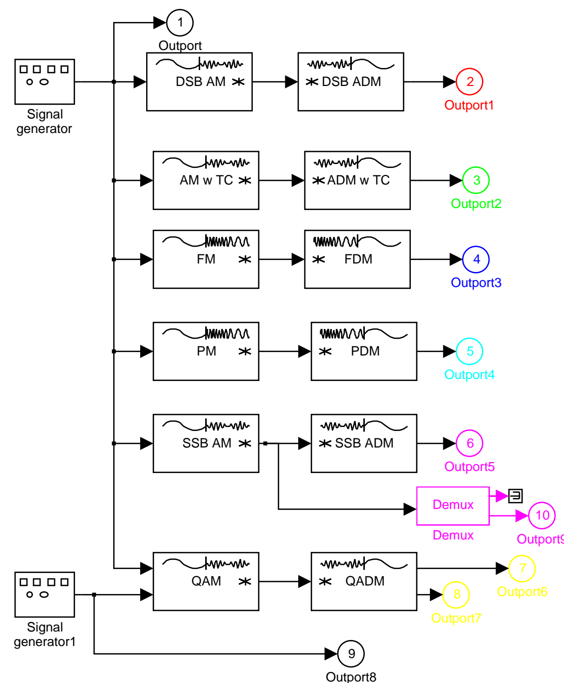


Figure 3-17: Simulink Block Diagram for Comparing Baseband Modulation/Demodulation Methods

Using the MATLAB commands

```
tutmodce
[t,x,yout] = euler('tutmodce',1)
```

invokes this Simulink block diagram for a one minute simulation. The simulation time is variable t. The simulation places each output in a column of the vector yout.

In the MATLAB simulation, the sample time and the lowpass filters are set to the same values as those used in the Simulink block diagram. The comments

in the code refer to the figure on the following page, which compares the MATLAB and Simulink baseband modulation/demodulation results:

```
% Sampling frequency
fs = 100;
% Lowpass filter
[num,den] = butter(2,100/2/pi/fs);
% DSB-SC AM simulation results - top left plot.
y = amodce(yout(:,1), 1/ts, 'amdsb-sc');
x = ademodce(y, 1/ts, 'amdsb-sc/costas', num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 2), 'y--')
% DSB-TC AM simulation results - top right plot.
y = amodce(yout(:,1), 1/ts, 'amdsb-tc', 1);
x = ademodce(y, 1/ts, 'amdsb-tc/costas', 1, num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 3), 'y--')
% FM simulation results - middle left plot.
y = amodce(yout(:,1), 1/ts, 'fm');
x = ademodce(y, 1/ts, 'fm', num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 4), 'y--')
% PM simulation results - middle right plot.
y = amodce(yout(:,1), 1/ts, 'pm');
x = ademodce(y, 1/ts, 'pm', num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 5), 'y--')
% SSB AM simulation results - bottom left plot.
[num_h, den_h] = hilbiir(ts, 10*ts, 10);
y = amodce(yout(:,1), 1/ts, 'amssb/time', num_h, den_h);
x = ademodce(y, 1/ts, 'amssb', num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 6), 'y--')
% QAM simulation results - bottom right plot.
% There are two input signals for the QAM modulation, the in-
% phase component and the quadrature component. The plot compares
% both components.
y = amodce([yout(:,1), yout(:,9)], 1/ts, 'qam');
x = ademodce(y, 1/ts, 'qam', num, den);
plot(t, yout(:,1), t, x, 'y--', t, yout(:, 4), 'y--')
```

The Simulink simulation results are plotted against the MATLAB simulation results together with the original input signal source. This figure includes plots for each modulation method:

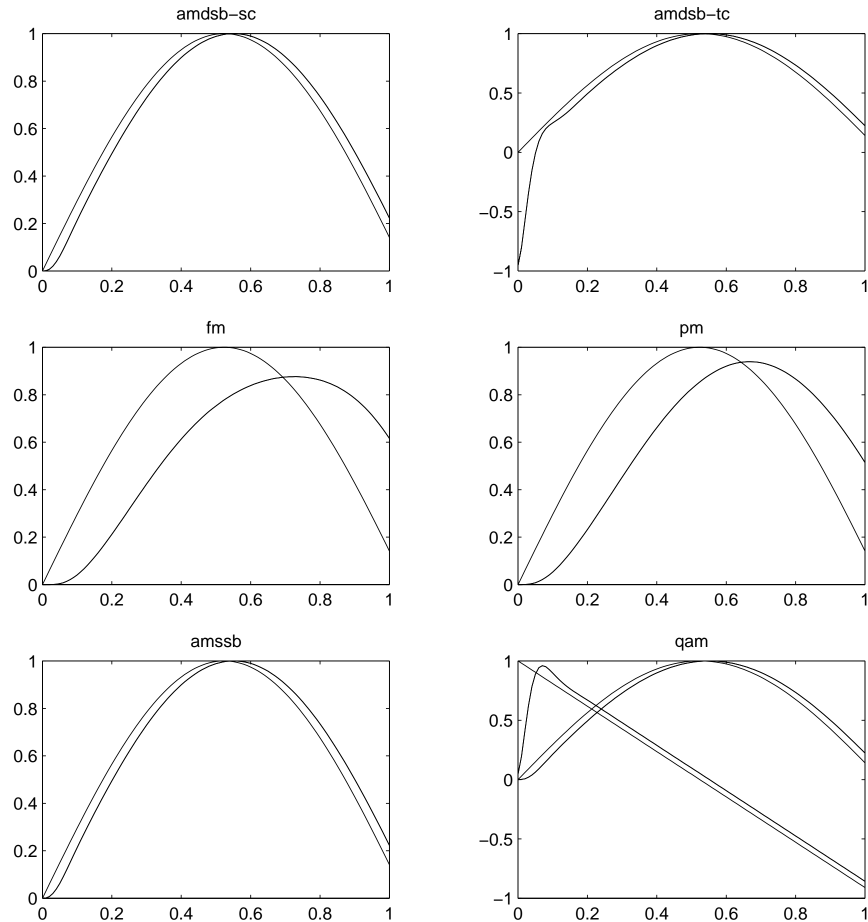


Figure 3-18: A Comparison of MATLAB and Simulink Simulation of Baseband Modulation/Demodulation Methods

The solid curves are the original input signals. The dotted curves are the Simulink results. The dashed curves are the MATLAB results. Because the

differences between the Simulink and MATLAB results are small (within numerical accuracy), the dotted and dashed curves overlap and cannot be distinguished. The recovered signal can be distinguished from the input signals by the time lags evident in each plot. The lowpass filters cause this delay between the original signal and the recovered signal. The lowpass filters also remove the high frequency part of the saw-tooth signal.

Signal Multiple Access

Signal *multiple access* is a technique that allows sharing of fixed communication resources by multiple users. Most modern communication transmitting resources are shared by multiple users. A typical multiple access communication system is shown in the following figure. A multiple access block on the transmitting side collects all information to be transferred in a specified sequence and sends the information through the transmitting channel. A multiple access block on the receiving side recovers the individual signals and divides the signals among the individual receivers in a specified sequence.

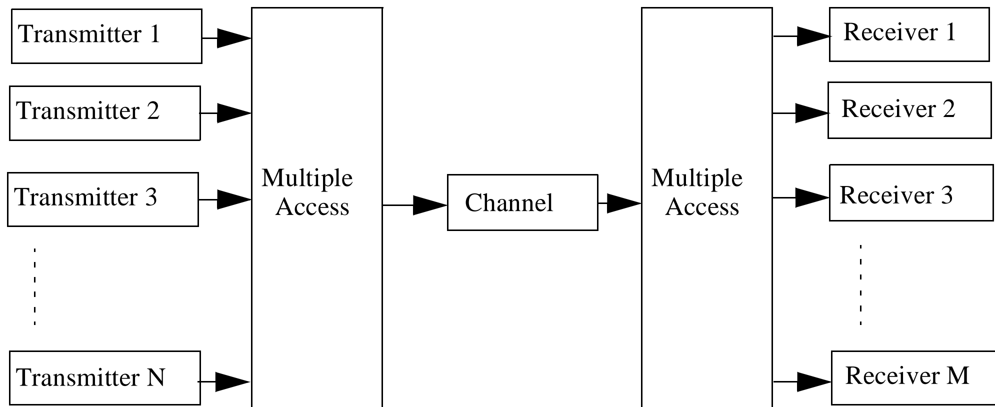


Figure 3-19: Multiple Access Block Diagram

The toolbox includes four different multiaccess techniques:

- Time-share multiplexing and demultiplexing
- TDMA (time division multiple access)
- FDMA (frequency division multiple access)
- CDMA (code division multiple access)

The following sections discuss each of these methods in turn.

Time-Share Multiplexing and Demultiplexing

The figure below shows an example of time-share multiplexing and demultiplexing. The input is a length 3 vector signal. The three input signals

are shown on the left side of the figure. The background shades of the plots are carried on to each stage of the time-share process. The middle plot is the result of the time-share process, which is a scalar signal. The plots on the right are the signals recovered from the time-share signal. In this example, the time interval [kT, (K+1)T] is divided into three sections. The time interval for each section is T/3. The first signal takes the first section; the second signal takes the second section; and the third signal takes the third section. The recovery side uses the same time-sharing division.

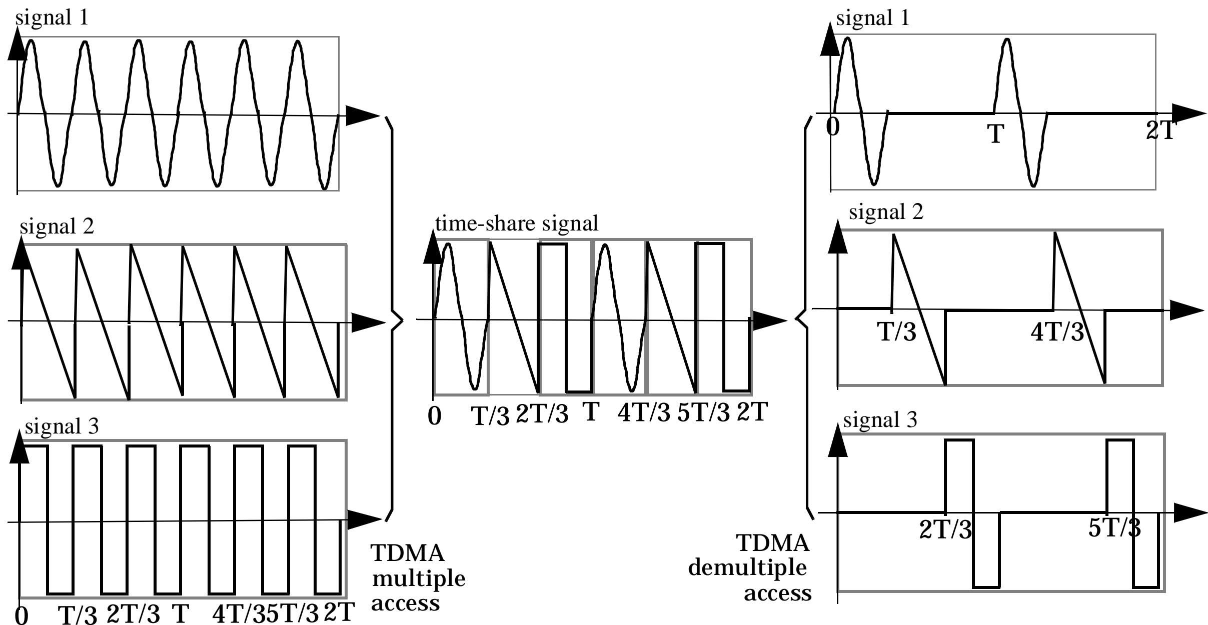


Figure 3-20: Time-Share Multiplex/Demultiplex Example

On the receiving side, the time-share process sends the time-share signal to whichever output signal has the allotted time slot. You have a choice about what the other output signals do during the times during which they have no access to the time-share signal. They can either output zero values or hold the last value they received when their allotted time occured in the transfer period. In the above example, the signals output zeros during the times in which they had no access to the time-share signal.

Vector Signal Redistribution

In time-share multiplexing, when an output signal does not have access to a time slot, it either is zero or holds its most recent value.In the time-share multiplexing example in the last section, sine waves went to the first output, triangle waveforms went to the second output, and square waves went to the third output. With the Vector Redistributor block, it is possible to combine and rearrange time-share signals in arbitrary sequences. This figure shows an example of the vector redistribution:

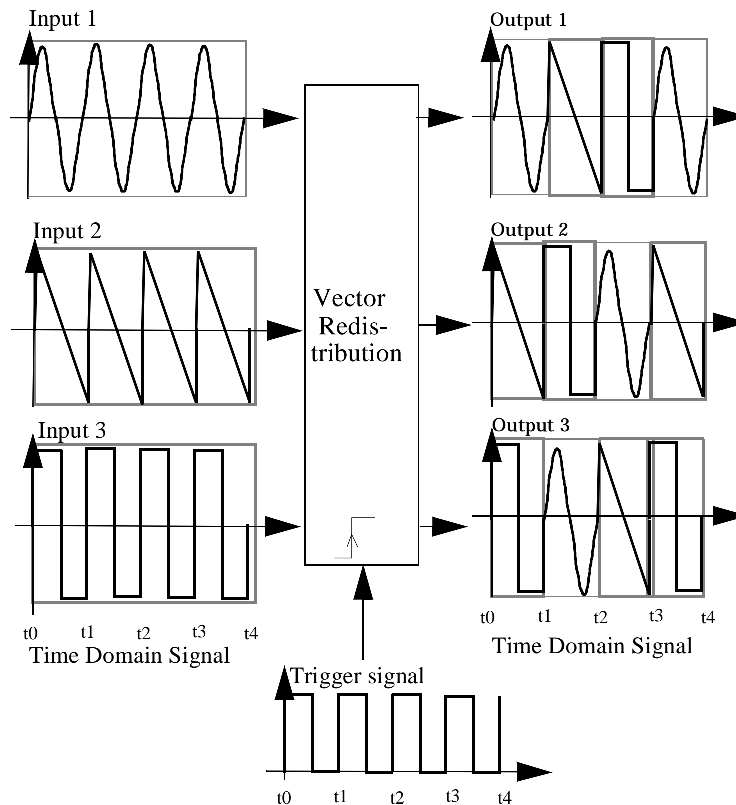


Figure 3-21: Vector Redistribution Example

In this example, there are three input and output signals. The output signals are a redistribution of the input signals. Reordering the input signals into the output signals is done each time a trigger signal occurs. To recover the original

signal, use the Vector Redistributor block again to rearrange the input signal into the inverse of the first redistribution.

Time Division Multiple Access (TDMA)

TDMA is a multiple access technique that divides a single transmitting time period into time sharing sections. Each input signal takes one time slot for each time period in the transmission. In this toolbox, it is assumed that there is some number, N, of inputs that must share the communication channel. Each input has exclusive access to the channel during its assigned time slot, but it cannot transmit at other times. The channel must have adequate bandwidth to accommodate the needs of all the inputs.

Digital TDMA

Digital TDMA is a multiple access method that gives access to the communication channel to each of a number of inputs for a limited time period. Only one input has access to the channel at any given time, and the method rotates through each input in a periodic fashion. Data is transmitted over the channel serially, and the channel data rate must be sufficiently high so that accumulated data at each input can be transmitted within its available time slot. Compression of the time scale of the inputs is equivalent to bandwidth expansion so that the data from each input can be transmitted within its allotted time frame.

In the example pictured below, there are 3 inputs. The background of the each input element indicates the time that each signal occupies in the TDMA signal. The time interval for each period is T. For N inputs, the sampling time for the TDMA signal is T/N. The recovered output signal from TDMA has a sampling time T and the ith element in the output signal has offset as (i-1)T/N. There is no special limitation for the input signal. The ith output element is equivalent to the sampling of the ith input element with sampling time T and offset (i-1)T/N. When the input signal has the exact sample time and offset, the signal output is exactly the same as the input signal.

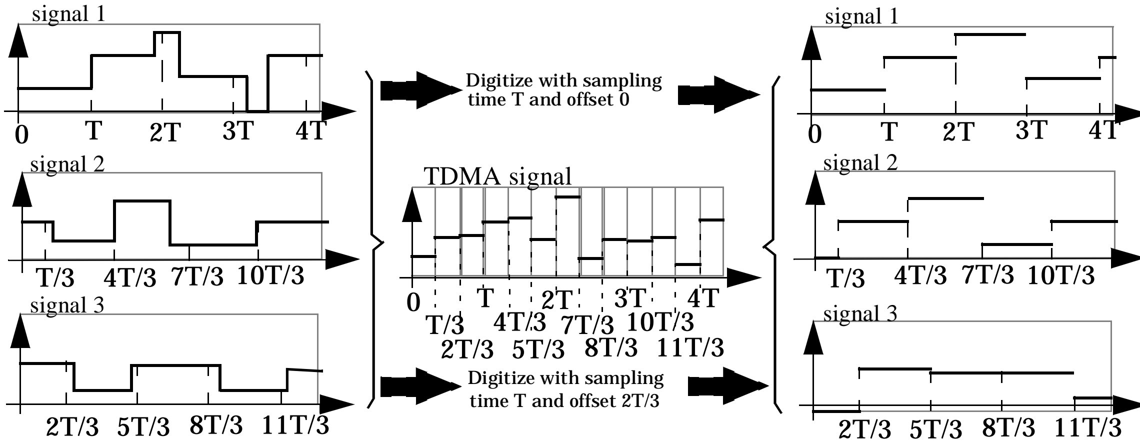


Figure 3-22: Digital TDMA Example

This toolbox provides both analog and digital TDMA blocks in the Simulink block library.

Frequency Division Multiple Access (FDMA)

FDMA is a technique widely used in radio frequency transmitting. A transmitting resource is shared by a number of transmitting sources. Each transmitting source takes a special assigned frequency interval. Usually, building a FDMA system involves modulation techniques. You can use the modulation method of your choice to modulate the message signal.

Modulation can be used to move the frequency range occupied by a signal to a different part of the frequency spectrum by selecting an appropriate carrier frequency. In this way a number of signals can be transmitted over a channel provided that the overall bandwidth of that channel is wide enough to accommodate the sum of the individual signal bandwidths. Usually, bandpass filtering is needed to reduce interference and crosstalk between the different signals.

An FDMA example is shown in the following figure. Three message signals are modulated into f_1, f_2, and f_3 respectively. Assume the lower bound for the ith modulated signal is f_i-B_i and the upper bound of the frequency is f_i+B_i. It is also assumed that $f_{i+1}+B_{i+1} < f_i-B_i$, for i=1,2,3.

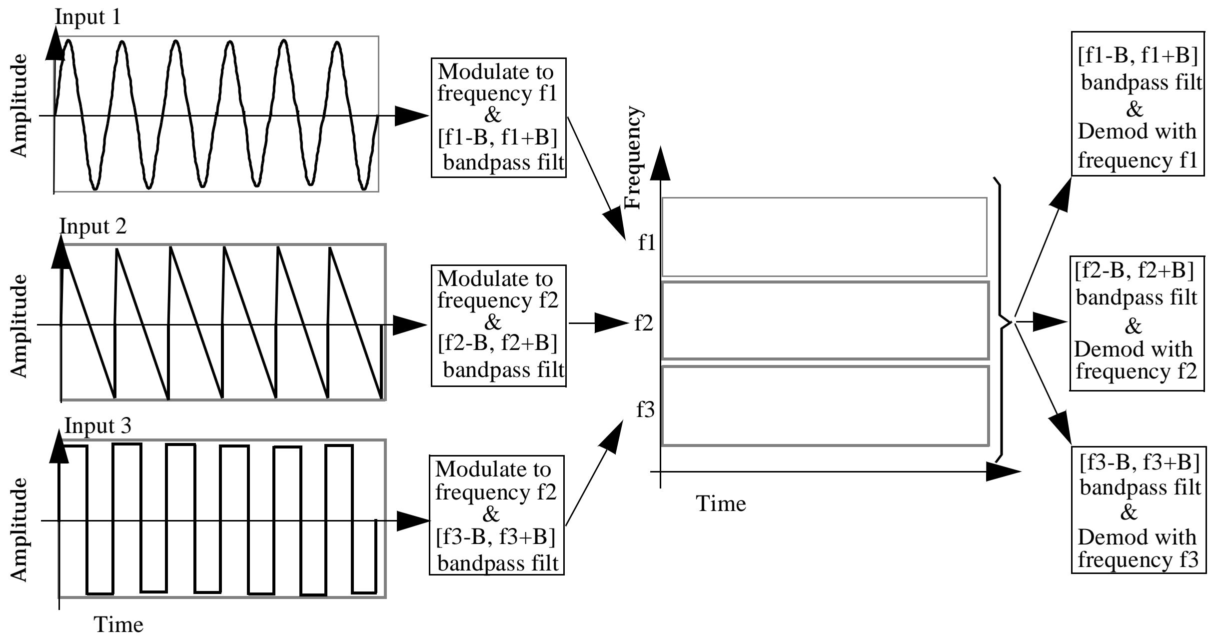


Figure 3-23: FDMA Example

The multiple accessed signal is shown in the middle of the figure. On the receiving side, the received signal goes through a bandpass filter. Then a demodulation process recovers the signal as shown in the plots of the left-hand side of the figure.

The block diagram of an example using the toolbox to build a FDMA system is shown in the following figure. This example uses the DSB-SC AM modulation method.

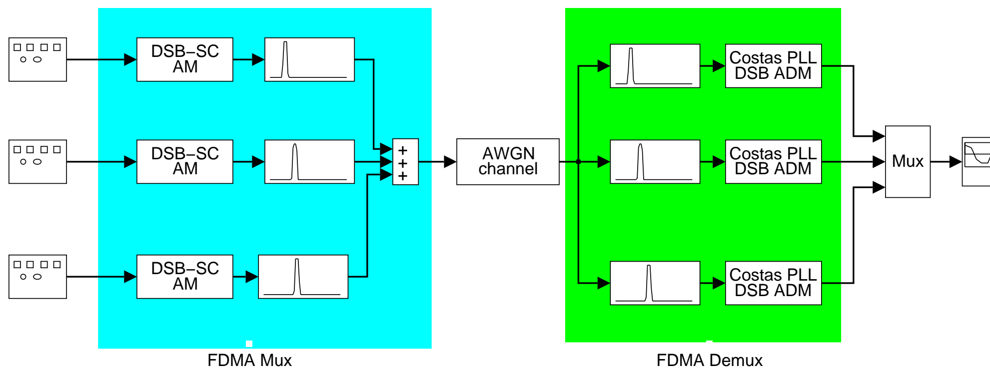


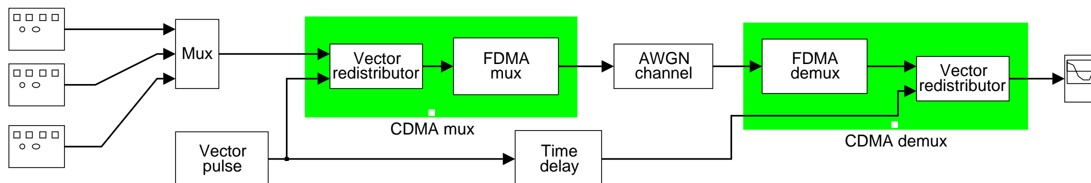
Figure 3-24: Simulink Block Diagram FDMA Example

The detailed discussion of the FDMA block diagram is provided in the "Simulink Block Reference" chapter.

Code Division Multiple Access (CDMA)

CDMA is widely used in modern communication systems, such as wireless communication systems. There are a number of CDMA methods used in applications. This toolbox provides a frequency hopping CDMA as an example of how to use the toolbox to implement a CDMA communication system. For more information about CDMA and frequency hopping methods, refer to Sklar, *Digital Communications, Fundamentals and Applications*, p. 491-3.

A frequency hopping CDMA uses the vector re-distributor discussed in the TDMA section. The following block diagram shows a simple frequency hopping CDMA system example.



The frequency hopping CDMA process rearranges the input vector signal, then, the rearranged signal vector goes through the FDMA process. The recovery of the CMDA signal is the inverse process of the CDMA process.

The time-frequency distribution map of the frequency hopping CDMA Mux for this example is shown in the following figure. Three signals are mixed with time, and then the mixed signals are modulated to different frequencies. The time-frequency sharing map related to the original signal is shown on the right-hand side of the figure. The CDMA Demux is the exact reverse of the flow with the consideration of the time delay.

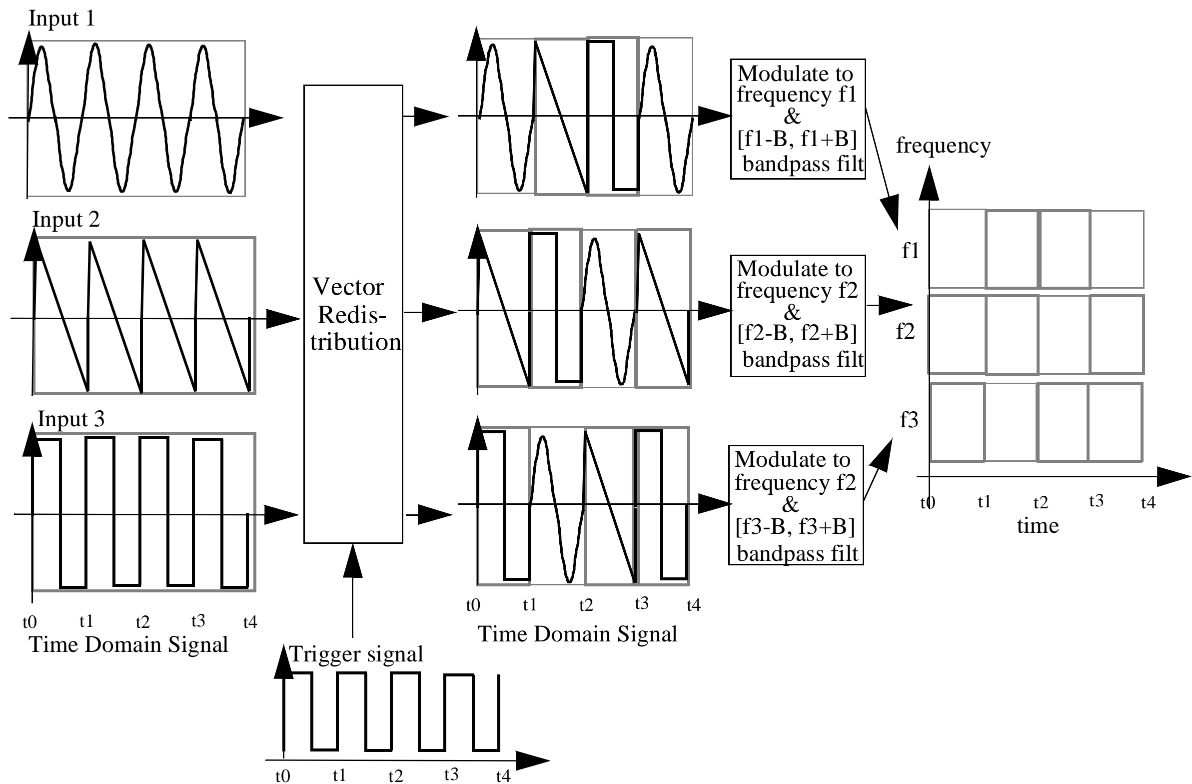


Figure 3-25: CDMA Example

Please read the Simulink reference for the Simulink simulation result by using the example.

Transmitting and Receiving Filters

There are a number of filters specially designed for communication systems. This section discusses these filters. The filters discussed in this section include the raised cosine filter, sinc filter, Hilbert transform filter, and complex filter. These filters are mostly used in the transmitting and receiving front. The commonly used digital filters such as the lowpass filters, highpass filters, bandpass filters, and bandstop filters can be found in the DSP Blockset, which is another product in the MATLAB product family.

Different design methods are available in the function blocks, such as Butterworth filter, Chebychev filter (type I and type II), and Elliptic filter, etc.

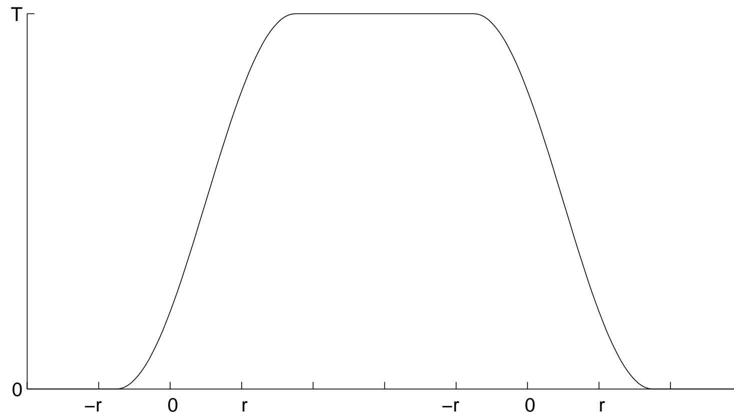
Raised Cosine Filter

The transfer function of the raised cosine filter is

$$H(f) = \begin{cases} T & \text{if } 0 \le |f| \le \dfrac{1-r}{2T} \\ \dfrac{T}{2}\left[1 + \cos\left(\dfrac{\pi T}{r}\left(|f| - \dfrac{1-r}{2T}\right)\right)\right] & \text{if } \dfrac{1-r}{2T} \le |f| \le \dfrac{1+r}{2T} \\ 0 & \text{if } \dfrac{1+r}{2T} \le |f| \end{cases}$$

where r is the rolloff factor and T is the time interval of the digital message. After a digital signal passes a R-C filter, the frequency of the signal is limited in the band [-W, W], where $W > \frac{1}{2T}$. The frequency spectrum of the raised

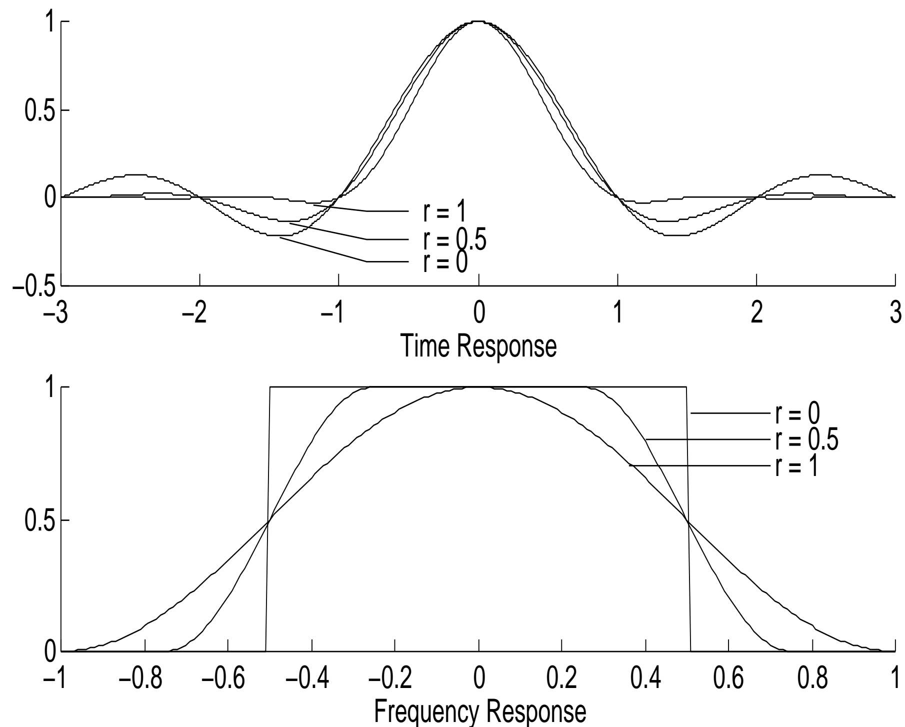
cosine filter is shown in the following plot.



The time response of the filter is:

$$h(t) = \frac{\operatorname{sinc}\left(\frac{t}{T}\right)\cos\left(\frac{\pi r t}{T}\right)}{1 - 4\frac{r^2 t^2}{T^2}}$$

This figure shows the time domain impulse and frequency responses of the R-C filter (for r=0, 0.5, 1) in the upper and lower plots respectively:.



Note that T has been normalized to 1 in these graphs. When the input is a step signal or a continuous signal, a sinc filter should be used before using the raised cosine filter. The sinc filter is discussed in the next section.

The raised cosine filter is a non-causal filter. The implementation of a non-casual filter is impossible because a non-causal filter depends on future input information. To solve this problem, this toolbox has artificially added a time delay to the filter in the implementation. The time delay is usually T multiplied by a positive integer.

In some applications, a raised cosine filter can be divided into two parts, one for the transmitting side and one for the receiving side. In this case, each side

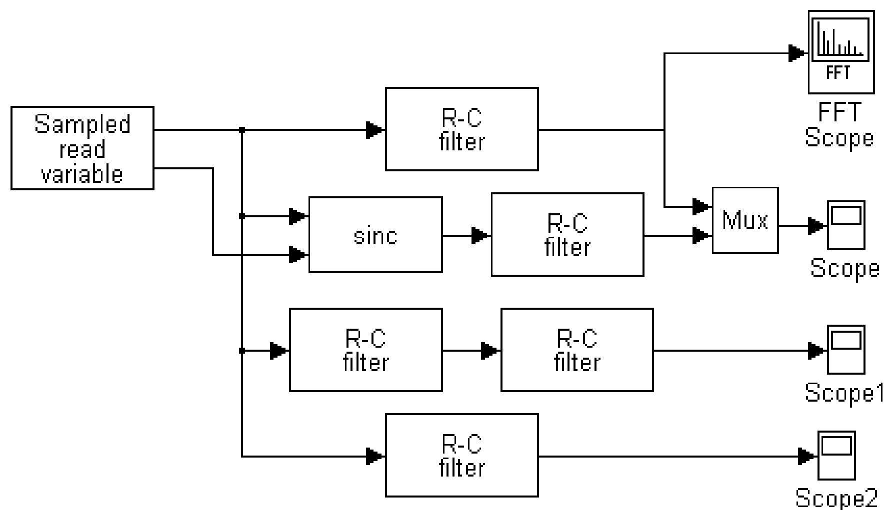
has the square root of the raised cosine filter. The impulse response of a square-root R-C filter is:

$$h(t) = \frac{4r}{\pi\sqrt{T}} \frac{\cos\left((1+r)\pi\frac{t}{T}\right) + \dfrac{\sin\left((1-r)\pi\frac{t}{T}\right)}{4r\frac{t}{T}}}{\left(1 - \left(4r\frac{t}{T}\right)^2\right)}$$

In using the square root raised cosine filter, the sinc filter should be used on the transmitting side but not on the receiving side. The combination of two square root raised cosine filters in this way is equivalent to a normal raised cosine filter if there is no channel noise and distortion.

Raised Cosine Filter Example

This example shows how to use the raised cosine filter. You can run this example by typing `tstrcos2` in the command window. The figure below shows the block diagram of the Simulink example:

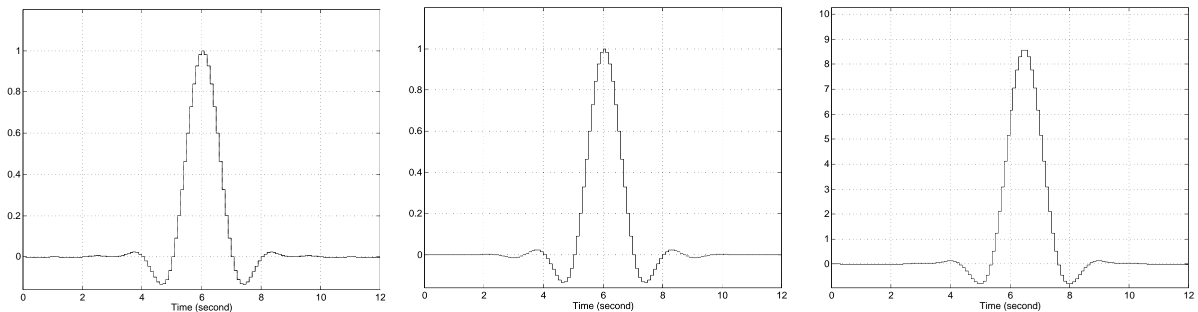


The sampling time is .1 sec. A pulse with width equal to 1 sec is generated from the Sampled Read Workspace block. The signal is sent to four different setups

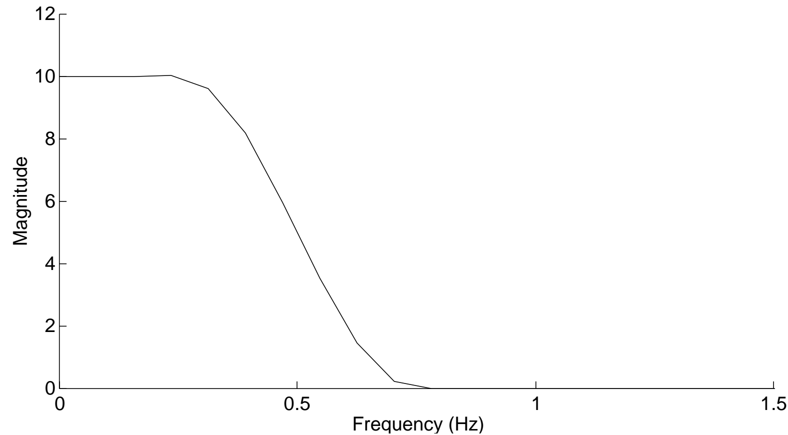
of the R-C filters. In all R-C filters, the digital transferring sampling time is set to be 1, computation sampling time is .1, filter rolloff factor is .5, and filter type is FIR. The first row of the R-C filter uses the sinc filter. The delay step is 6. The second row of the R-C filter does not use the sinc filter in the R-C filter parameter setup; it uses an external sinc filter. The left plot in the figure below shows the simulation result of the first and the second row. These two results overlap each other perfectly.

The third row uses two FIR/srqt filters. The first R-C filter uses a sinc filter and the second R-C filter does not use a sinc filter. The delay step for each block is 3. The middle plot in the figure below shows the simulation result. This simulation result is almost the same as when using a single normal R-C filter.

The fourth row uses no sinc filter. In the figure below, the right plot shows the simulation result. Note the amplitude of the simulation result is much larger compared to the other simulation results.



An FFT scope (this scope is provided in the DSP Blockset) is used to analyze the spectrum of the result in the first row. The simulation spectrum matches the theoretical spectrum.



Sinc Filter

The transfer function of the inverse sinc filter is a sinc function

$$\mathrm{sinc}(f) = \begin{cases} \dfrac{\sin(\pi f)}{\pi f} & t \neq 0 \\ 1 & t = 0 \end{cases}$$

The time domain impulse response of this filter is an sample-time-unit width impulse. This block is often used in conjunction with the raised cosine filter.

Hilbert Transform Filter

The Hilbert transform filter outputs a -90^{0} phase shifted signal over the input signal. The Hilbert transform filter has the transfer function

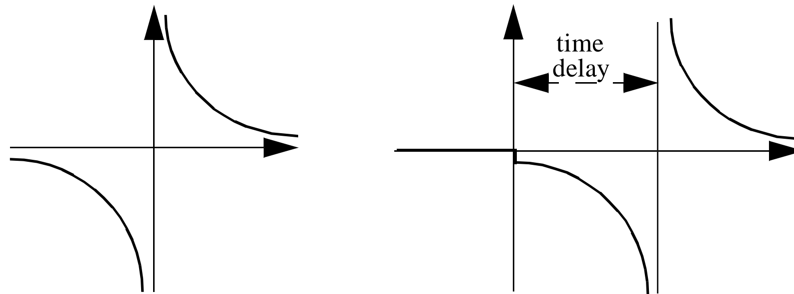
$$H(f) = -j\,\mathrm{sgn}(f)$$

which has the impulse response

$$h(t) = \frac{1}{\pi t}$$

The Hilbert transform filters are used in SSB AM modulations.

The Hilbert transform filter is a noncausal filter. This means that the current output depends on the future input of the system. The transfer function contains a purely imaginary component, which means the impulse response of the filter is odd symmetric. The impulse response of the system is depicted on the left-hand side plot of the figure.



(a) impulse response of HTF (b) impulse response of delayed HTF

The filter is not realizable since its impulse response is not zero prior to the time zero point. To overcome this difficulty, an artificial delay is added to the filter design. The delay is added such that the value of the impulse response before the time zero is not significant and is ignored in the implementation. The impulse response of a delayed Hilbert transform filter is shown on the right-hand side plot of the above figure.

Channels

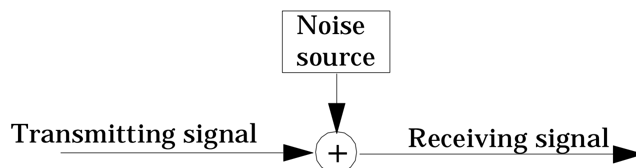
Communication channels introduce noise, fading, interference, etc., into the transmitting signals. In communication system simulation, the modeling of a channel is based on mathematical descriptions of the channel. The channel models differ from one transmitting media to another. This toolbox provides a limited number of transmitting channel models. You are encouraged to construct your own channel models. If the channel model is linear, you can use the functions in the System Identification Toolbox to identify the structure of the channel. If the channel model is non-linear, you can use the Optimization Toolbox to search for the parameters with measurement data and the given structure of the model.

The toolbox includes two different kinds of channel models: passband and baseband. The most commonly used model in channel simulation is the AWGN (additive white Gaussian noise) model.

In general, the channel model can be used directly in connecting the transmitting side and the receiving side of a communication system as shown in this figure:

Passband Channels

In passband channels, the transmitting signal is assumed to be a scalar signal. The passband channel sublibrary of this toolbox includes an AWGN channel model and a binary error channel model. This figure shows an additive noise channel model:



An AWGN model is a channel with white Gaussian noise as the noise source. Gaussian noise is discussed in the “Signal Generators and Display Devices” section of this chapter.

The channel fading model in the passband simulation is simply the transmitting signal multiply a gain value. The gain is less than or equal to one.

You can use the MATLAB function `rand` to generate uniform distribution noise and `randn` to generate Gaussian noise. The Statistics Toolbox supports more choices for noise generator functions as well.

Baseband Channels

Baseband channel models have the same channel model structure as discussed in the passband channel. The transmitting signal, receiving signal, and the noise source are complex. The real part of the signal is the in-phase component and the imaginary part of the signal is the quadrature component. In Simulink blocks, a complex signal is represented by a two element vector. The first element is the real (in-phase) component. The second element is the imaginary (quadrature) component.

Rayleigh Noise Channel

The noise input source for the Rayleigh noise channel is the Rayleigh noise. The noise source is discussed in the Signal Generators and Display Devices section. In the channel, two independent zero mean white Gaussian noises are added to the real part and the imaginary part of the transferred signal. These two independent white Gaussian noises share a same common variance.

Rician Noise Channel

The additive noise source for the Rician noise channel is a Rician noise. In the channel, two independent Gaussian noises are added to the real-part and imaginary part of the transferred signal. These two independent Gaussian noises share a same mean value and a same common variance. For a discussion of the Rician channel, see Webb and Hanzo, *Modern Quadrature Amplitude Modulation*, p.61.

Fading Channel

The fading channel in the baseband signal transmission has the following form

$$y(t) = u(t)\alpha(t)\exp(j\phi(t))$$

where $u(t)$ is the transferring signal; $y(t)$ is the output of the channel; $\alpha(t)$ is the fading envelope and $\phi(t)$ is the phase shift in radians. When both $\alpha(t)$ and $\phi(t)$ are time invariant, the fading channel has a fixed form.

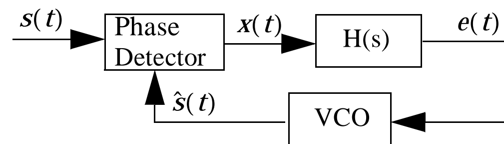
Synchronization

The communication receivers have to be synchronized with the transmitters in order to make a correct transmission. The simulation of the synchronization is a timing simulation. The synchronization functionality is provided in the Simulink block library only. The Communication Toolbox provides the Phase Locked Loop (PLL) synchronization simulation. A digital early-late gate simulation is provided as an example in the next chapter.

A PLL is an automatic control system to adjust the phase $\hat{\theta}(t)$ of the local signal $\hat{s}(t) = \sin(2\pi f_c t + \hat{\theta}(t))$ to match the phase the phase $\theta(t)$ of the received signal $s(t) = \sin(2\pi f_c t + \theta(t))$. In the PLL design, it is assumed that the phase $\theta(t)$ is slowly varying with respect to the frequency f_c.

Passband Signal PLL Simulation

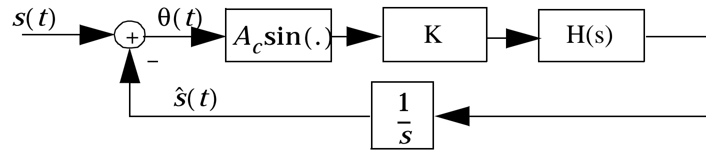
A typical PLL consists of three components: a phase detector, a filter H(s), and a VCO (voltage controlled oscillator). The phase detector in a simple PLL is a multiplier. The structure of a simple PLL is shown in the following figure.



In the figure, $s(t) = \sin(2\pi f_0 t + \theta(t))$ is the incoming signal and $\hat{s}(t) = \sin\left(2\pi f_0 t + \hat{\theta}(t) + \frac{\pi}{2}\right)$ is a locally generated signal. Note that $\pi/2$ has been added to the VCO as an initial value of the phase shift. In this toolbox, a multiplier phase detector is used. The output of the multiplier, *x(t)*, is $(t) = \frac{1}{2}\sin[\theta(t) - \hat{\theta}(t)] + \frac{1}{2}\sin[2\pi f_0 t + \theta(t) - \hat{\theta}(t)$. After passing a low-pass filter H(s), the high frequency term, the second term in the equation, is filtered. The signal *e(t)* is $e(t) = \frac{1}{2}\sin[\theta(t) - \hat{\theta}(t)]$. When $\theta(t) - \hat{\theta}(t)$ is small, we have $e(t) = \theta(t) - \hat{\theta}(t)$. Note that the PLL fails when the deviation of $\theta(t)$ is close to $\pi/2$.

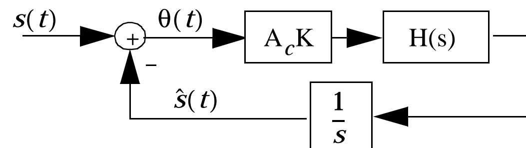
H(s) is a lowpass filter. You may use the MATLAB function `pllfilt` to design the filter.

The baseband model PLL method uses the multiplier phase detector and rearranging the variables, the PLL in the following figure is equivalent to the previous PLL described above.



The baseband model for the PLL is a nonlinear system model. The model depends on the information of the amplitude A_c of the incoming signal. The model is independent of the carrier frequency f_0.

The baseband model for the PLL is a nonlinear system. It is difficult to analyze a nonlinear system. Using the approximation $n[\theta(t) - \hat{\theta}(t)] \cong \theta(t) - \hat{\theta}($ when $\theta(t) - \hat{\theta}(t)$ is small, a linearized Baseband Model PLL block results, which is the simplest PLL model, as shown in the following figure.



The transfer function from the input phase $s(t)$ to phase error $\phi(t)$ is $G(s) = \frac{s}{s + A_c K H(s)}$. A PLL without filter is a first order PLL. A PLL with (n-1)th order H(s) is an nth order PLL. It is known that the steady state phase error $\theta(t)$ is zero for a rational H(s). You can use the MATLAB Control System Toolbox to design and to analyze the PLL systems.

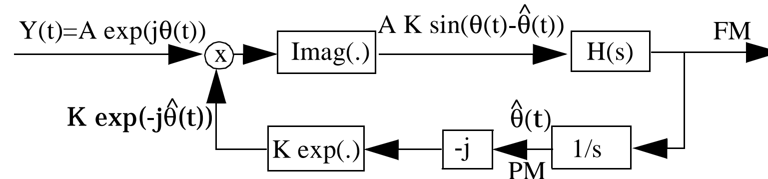
So far, all PLLs discussed above are analog. The charge pump PLL is a classical digital PLL. In the above discussion, a multiplier is used as phase detector. In practice, a number of other phase detectors can be used. One of them is the sequential logic phase detector. A PLL with a sequential logic phase detector (also named digital phase detector) is known as charge pump PLL.

A sequential logic phase detector operates on the zero crossings of the signal waveform. The equilibrium point of the phase difference between the input signal and the VCO signal equals π. The sequential logic detector can compensate for any frequency difference that might exist between a VCO and an incoming signal frequency. Hence, the sequential logic phase detector acts as a frequency detector. For this reason, a sequential logic phase detector is also called a phase/frequency detector (PFD).

A discussion of the charge pump PLL can be found in [40]. There are other references for the charge pump phase-locked loop, such as [17] and [18].

Baseband Signal PLL Simulation

Simulation using the baseband waveform has been discussed. PLL is one of the keys in the demodulation. The simulation of PLL can be on the baseband waveform. In this case, the simulation can be at a lower sample rate than that required to match the carrier frequency. The baseband waveform simulation for PLL can be done by using the structure shown in the following figure.



Please note that although the simulation of the baseband waveform PLL can have its simulation sample rate much lower than the carrier frequency f_c, to be able to find the accurate delay, the simulation step size should be small enough.

Utilities

This toolbox includes a number of basic functions and blocks to support the functions described in the previous sections. Please see the Simulink reference for detailed discussion on each of the functions. This section lists all functions. Selected functions are discussed in more detail while others are skipped because their functionality is obvious.

- Voltage Controlled Oscillator
- Modulo Integrator
- Windowed Integrator
- Scheduled Integrator
- Envelop Detector (not discussed)
- Edge Detector (not discussed)
- Register Shift (not discussed)
- Triggered Buffer Down (not discussed)
- Vector Redistribution (not discussed)
- Number Counter (not discussed)
- Error Counter (not discussed)
- Minimum/Maximum Index
- Array Functions
- Scalar to Vector Converter
- Vector to Scalar Converter
- Modulo Operation
- Mean and Variance
- Complex Filter (not discussed)
- K-Step Delay (not discussed)

Voltage Controlled Oscillator

The functionality of the voltage controlled oscillator (VCO) is the same as the frequency modulation. The voltage of the input signal *m(t)* controls the oscillation frequency of the output signal *y(t)*.

$$y(t) = m(t)\cos(2\pi f_c t + 2\pi\theta(t) + \phi_c)$$

where *y(t)* is the modulated signal. f_c is the carrier frequency. The unit for f_c is Hertz. ϕ_c is the initial phase. The unit for ϕ_c is radians. θ(t) is the modulation phase, which is the integration of the input signal *m(t)*.

$$\theta(t) = k_c\int_0^t m(t)dt$$

In the equation, k_c is the sensitivity factor.

Modulo Integrator

Simulink can be used for both continuous-time and discrete-time system simulation. In continuous-time system simulation, the step size reduces if the computation error is larger than the given tolerance. The computation error is determined by the relative error in the integration computation. In a simulation system, if one integration value is getting much larger than the others, that integration value will play a dominant role in the computation tolerance evaluation. In some applications, the modulo of the integration value is used instead of its original value.

One application is the computation of the FM modulation. The modulated signal y(t) is

$$y(t) = \cos(2\pi f_c t + 2\pi\theta(t) + \phi_c)$$

where θ(t) is the modulation phase, which changes with the amplitude of the input *m(t)* in the form

$$\theta(t) = k_c\int_0^t m(t)dt$$

If the input value *m(t)* keeps the same value, the value θ*(t)* can be very large. However, only θ*(t)* modulo one is used in the output computation. This integration outputs the value of the modulo integration.

Windowed Integrator

For an input signal *u(t)*, the output of the windowed integrator is

$$y(t) = \int_{t-T}^{t} u(\tau)\,d\tau$$

where T is a positive number, which is the integration window size of the integrator. In the computation, $u(t)=0$, for $t<0$ is assumed.

Scheduled Reset Integrator

Given an input signal *u(t)*, the output of the scheduled reset integrator in the time interval $nT < t \le (n+1)T$ is

$$y(t) = \int_{t-nT}^{t} u(\tau)\,d\tau$$

where T is a positive number, which is the reset schedule interval.

Minimum/Maximum Index

This block has been used in the M-ary FSK demodulation block. The functionality of this block is to output the index of the element with its value having the maximum value (or minimum value).

Array Functions

Some blocks in the current Simulink block library can be used only for scalar input or scalar output. An array function block has been added for multiple input/multiple output functions. This block is limited to calculate the functions with the input and output vectors having exactly the same size. Some functions that can be calculated using this block include: `sin`, `cos`, `tan`, `exp`, `log`, `ln`, etc.

Scalar to Vector Converter

This function converts scalar integer symbols to binary (base 2) vectors. The function `de2bi` is designed for this usage. A Scalar to vector converter block in Simulink has the same functionality.

Vector to Scalar Converter

This function converts binary (base 2) vectors to integer scalar numbers. The function `bi2de` is designed for the usage. A Vector to scalar converter block in Simulink has the same functionality.

Modulo Operation

The modulo operation in MALTAB is an internal function `rem`. In the Simulink block library, a similar functionality is implemented.

Mean and Variance

In MATLAB, you can use functions `mean` and `std` for the mean and variance calculations. In the Simulink block library, a block is implemented to calculate the mean and variance of all of the past input values of the signal.

Galois Field Calculations

This toolbox provides a set of self-contained Galois field functions to support error-control coding and decoding. This section discusses the definition of Galois fields and the arithmetic and algebraic computations in these fields.

Galois Field, GF(q)

A finite field, that is, a field containing a finite number of elements, is called a *Galois field*. The notation for a Galois field is GF(q) where $q=p^M$. The parameter p must be a prime number, and M must be a positive integer. For instance, there is no finite field containing 24 elements. If q=p (i.e., M=1), then GF(q) is known as a *prime field*. If M>1, then GF(q) is called an *extension field*. Addition and multiplication in a prime field are equivalent to the corresponding modulo-p operations on integers, but this is not true for extension fields. As an example of addition in a prime field, this table lists all the possible addition calculations in GF(5):

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

This table lists all the multiplications possible in GF(5):

*	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

You can perform addition in prime and extension fields by using the function gfadd. Multiplicative in prime and extension fields can be done by using function gfmul. You can construct the above addition and multiplication tables by using the following commands:

```
q = 5;
gf_q_add = []; % Assign a variable name for addition table.
gf_q_mul = []; % Assign a variable name for multiplication table.
for i = 0:q-1
   gf_q_add = [gf_q_add; gfadd(i*ones(1, q), [0:q-1], q)];
      % Construct the addition table.
   gf_q_mul = [gf_q_mul; gfmul(i*ones(1, q), [0:q-1], q)];
      % Construct the multiplication table.
end;
```

By changing the value of q, you can construct a table for any prime number.

Division and subtraction in prime and extension fields can be done by using the commands gfdiv and gfsub respectively.

GF(2) is a special Galois field, the *binary Galois field*. Most error-control coding algorithms are constructed using GF(2). In GF(2), there are only two elements: “0” and “1”.

Polynomials over GF(q)

A polynomial over GF(q) is a polynomial whose coefficients are elements from GF(q). This toolbox uses a vector to represent a polynomial. The ascending ordered format is used in representing a polynomial. For instance, a polynomial over GF(q)

$$f(X) = g_0 + g_1 X + g_2 X^2 + g_3 X^3 + \ldots + g_M X^M$$

is represented by a vector $[g_0\ g_1\ g_2\ g_3\ \ldots\ g_M]$. Function gfpretty is designed to improve the visual presentation of algebraic equations. For example,

```
x = [1 2 3 4 5 4 3 2 1];
gfpretty(x)
ans =
                2      3       4     5      6      7    8
1 + 2 X + 3 X  + 4 X  + 5 X  + 4 X  + 3 X  + 2 X  + X
```

The degree of a polynomial is defined to be the highest exponent value of *X* in *f(X)*. The above example *f(x)* has degree 8. You can add two polynomials by directly using gfadd. The multiplication of two polynomials (convolution) can be done by using gfconv. The division of one polynomial by another (deconvolution) in GF(q) can be done by using gfdeconv. For example, to multiply polynomial *x* = [1 1 1 1 1] by polynomial *y* = [1 1 1 1 1] over GF(7), use the command

```
z = gfconv(x, y, 7)
```

which results in *z* = [1 2 3 4 5 4 3 2 1] in GF(7). The following functions are specially designed for polynomial computation in GF(q) or GF(q^M):

gfadd, gfconv, gfdeconv, gfprimck, gfprimdf, gfprimfd

The last three functions on the list are designed for primitive polynomial computation. This concept is discussed in the next section, "GF(q^M), the GF(q) Field Extension."

Polynomials over GF(2) are the most commonly used in error-control coding. The polynomial functions listed above use GF(2) as the default if the prime number q is not specified.

GF(q^M), the GF(q) Field Extension

The elements of an extension field, GF(q^M), of a prime field GF(q) can be conveniently represented by polynomials of degree M-1 and coefficients in GF(q). All elements of GF(q^M) can be generated by constructing polynomials of degree M-1 with all q^M possible combinations of coefficients in GF(q). Addition and subtraction of two polynomials over GF(q) are performed by simply performing those operations on the corresponding coefficients.

Determining the rules for multiplication is more complicated and requires some additional definitions. A primitive element, α, of GF(q) has the property that all nonzero elements of the field can be generated by forming successive powers of α. Every Galois field has at least one primitive element. A degree-M polynomial with coefficient in GF(q) is said to be *irreducible* if it is not divisible by any polynomial with coefficients in GF(q) and of degree less than M and greater than zero. An irreducible polynomial having a primitive element is known as a *primitive polynomial*.

Multiplication of polynomials is done in the customary manner except for the fact that powers that become too high for the extended Galois field are reduced by means of a primitive polynomial set equal to zero. If $p(X)$ is a primitive polynomial for the extension field, then we can use the relationship

$$p(X) = a_0 + a_1 X + a_2 X^2 + \ldots + X^M$$

or

$$X^N = -a_0 - a_1 X - a_2 X^2 - \ldots - a_{M-1} X^{M-1}$$

Note that for an element a in the binary field

$$a = -a$$

Multiplication tables for elements of an extended Galois field can be generated by the use of a primitive polynomial. Note that a specific primitive polynomial is associated with the table.

A list of all elements in a GF(q^M) can be generated by using function `gftuple`. For example, a list of GF(3^3) can be generated by the following commands:

```
% specify the exponential value M and prime number q
M = 3; q = 3;
% generate a list of GF(q^M) tuple form in tuple
% and exponential form in pow.
[tuple, pow] = gftuple([-1:q^M-2]',M,q)
```

There are three different kinds of representations for the elements in $GF(q^M)$, the exponential representation, the M-tuple representation, and the polynomial representation. This is a list of all the elements in $GF(3^3)$:

Exponential	3-Tuple	Polynomial
$\alpha^{-\infty} = 0$	[0 0 0]	0
$\chi^{0} = 1$	[1 0 0]	1
α	[0 1 0]	X
α^{2}	[0 0 1]	X^2
α^{3}	[2 1 0]	$2 + X$
α^{4}	[0 2 1]	$2X + X^2$
α^{5}	[2 1 2]	$2 + X + 2X^2$
α^{6}	[1 1 1]	$1 + X + X^2$
α^{7}	[2 2 1]	$2 + 2X + X^2$
α^{8}	[2 0 2]	$2 + 2X^2$
α^{9}	[1 1 0]	$1 + X$
α^{10}	[0 1 1]	$X + X^2$
α^{11}	[2 1 1]	$2 + X + X^2$
α^{12}	[2 0 1]	$2 + X^2$
α^{13}	[2 0 0]	2
α^{14}	[0 2 0]	2X
α^{15}	[0 0 2]	$2X^2$
α^{16}	[1 2 0]	$1 + 2X$
α^{17}	[0 1 2]	$X + 2X^2$
α^{18}	[1 2 1]	$1 + 2X + X^2$

Exponential	3-Tuple	Polynomial
α^{19}	[2 2 2]	$2 + 2X + 2X^2$
α^{20}	[1 1 2]	$1 + X + 2X^2$
α^{21}	[1 0 1]	$1 + X^2$
α^{22}	[2 2 0]	$2 + 2X$
α^{23}	[0 2 2]	$2X + 2X^2$
α^{24}	[1 2 2]	$1 + 2X + 2X^2$
α^{25}	[1 0 2]	$1 + 2X^2$

For convenience, this toolbox uses the exponent of α to represent the element in GF(q^M). For example, the table uses x=9 to represent α^9. By convention, $\alpha^{-\infty}$ is represented by any negative integer. You can use function `gftuple` to convert a vector of exponential form to M-tuple form.

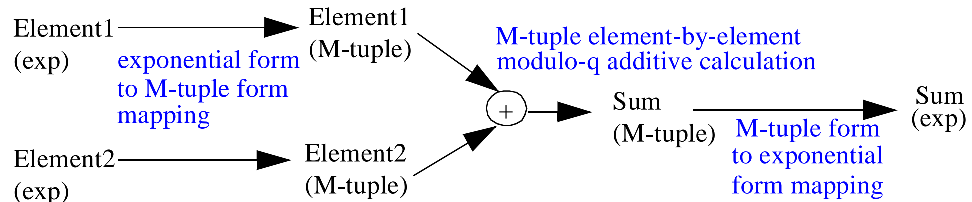
The extension field GF(q^M) possesses many interesting mathematical properties. Calculations are not as straightforward in GF(q^M) as in GF(q). The Communications Toolbox supports a number of calculations on GF(q^M), including:

- Addition in GF(q^M)
- Multiplication in GF(q^M)
- Cyclotomic coset calculation in GF(q^M)
- Minimal Polynomials over GF(q^M)
- The Roots of Polynomials over GF(q^M)
- Various utilities, such as polynomial truncation

The following sections of this chapter discuss each of these topics in turn.

Addition in GF(q^M)

Addition in GF(q^M) can be done as an element-by-element modulo-q calculation on M-tuple vectors. Starting from the exponential representation, the calculation is shown in this flow chart:



You can use `gfadd` or `gfplus` for the GF(q^M) calculation. A special variable `tp` should be created as the third parameter in the function call. `tp` must contain all the elements in GF(q^M). Addition is defined between two elements in the same extension field, GF(q^M).

This is an example of addition using vectors $a = \begin{bmatrix}\alpha & \alpha^3 & \alpha^5 & \alpha^7\end{bmatrix}$ and $b = \begin{bmatrix}\alpha^2 & \alpha^4 & \alpha^6 & \alpha^8\end{bmatrix}$ in GF(3^3):

```
% exponential form of vector a and b
a = [1 3 5 7];
b = [2 4 6 8];
% generate GF(3^3) tuple list
tp = gftuple([-1:3^3-2]', 3, 3);
% The additive calculation
c = gfadd(a,b,tp)
c =
  10 12 14 16
```

You can validate the result by using the GF(3^3) element list table. To verify the addition of α^7 plus α^8 calculated above, find from the GF(3^3) table the 3-tuple representation of α^7 and α^8. The representation for α^7 is [2 2 1], and the representation for α^8 is [2 0 2]. The addition of these two elements is [1 2 0], which is the 3-tuple representation of α^{16}. This result agrees with what was found in the above calculation using `gfadd`.

Multiplication in GF(q^M)

Using the exponential form, multiplication is relatively simple compared to addition. You need only add exponents of α modulo-q^M-1. For example, you can

multiply vectors $a = \begin{bmatrix}\alpha^{-Inf} & \alpha^{5} & \alpha^{15} & \alpha^{20}\end{bmatrix}$ and $b = \begin{bmatrix}\alpha^{1} & \alpha^{7} & \alpha^{14} & \alpha^{21}\end{bmatrix}$ using these commands:

```
a = [-Inf 5 15 20];
b = [1 7 14 21];
tp = gftuple([-1:3^3-2]', 3, 3);
c = gfmul(a, b, tp);
c =
  -Inf 12 24 10
```

The Cyclotomic Cosets of GF(q^M)

Let β be an element in GF(q^M). The element β^{2t} is called a *conjugate* of β, where *t* is a positive integer. It can be shown that for any polynomial *f(X)* with its element in GF(q), if $f(\beta) = 0$, then $f(\beta^{2t}) = 0$.

For any β in GF(q^M), there exists a subgroup in a GF(q^M), called the *cyclotomic coset* associated with β, such that $coset = \begin{bmatrix}\beta & \beta^2 & \beta^4 & \dots & \beta^{2S}\end{bmatrix}$ is closed, i.e., $\beta^{2(S+1)}$ equals one of the elements in the coset as listed above. The element with the smallest power number is called the *leader* of the coset. In the above example, β is the leader of the coset. The cyclotomic cosets can be calculated by using `gfcosets`.

For example, the cosets for GF(2^4) can be calculated by the command

```
cs = gfcosets(4, 2)
cs =
             0   NaN   NaN   NaN
             1     2     4     8
             3     6    12     9
             5    10   NaN   NaN
             7    14    13    11
```

The notation `NaN` is used to fill the space in the matrix. This toolbox uses cyclotomic cosets in the computation of minimal polynomials and for the computation of roots in GF(q^M).

Minimal Polynomials over GF(q^M)

The minimal polynomial of β, where β is on GF(q^M), is defined to be the polynomial of the smallest degree over GF(q) such that f(β)=0. In this toolbox,

function gfminpol is designed to find the minimum polynomial. For example, to find the minimum polynomial of $\beta = \alpha^3$, where β is in GF(2^4), you can simply use the command

```
pmin = gfminpoly(3, 4, 2)
pmin =
1  1  1  1  1
```

Using the function gfpretty, you can obtain the “pretty” form of pmin:

```
gfpretty(gfminpoly(3, 4, 2))
1 + X + X^2 + X^3 + X^4
```

The Roots of Polynomials over GF(q^M)

The roots of a polynomial *f(X)* are defined to be the solutions of *X* such that *f(X)=0*. A polynomial with its coefficients in GF(q) may have no root in GF(q). However, you can find its roots in an extended field GF(q^M) by using the function gfroots. For example, consider the polynomial $1 + X + X^2 + X^3 + X^4$ over GF(2). Neither 0 nor 1 is a solution for this polynomial, so it doesn’t have any solutions in GF(2). You can find its roots in GF(2^4) by using function gfroots:

```
a = [1  1  1  1  1];
rts = gfroots(a, 4, 2)';
rts =
      3  6  12  9
```

Other GF(q) Related Functions

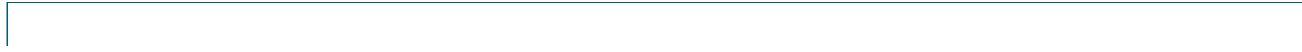
There are a number of other functions implemented in this toolbox to support the GF(q) computation. These functions are

- gftrunc, which truncates an input polynomial to eliminate useless terms
- gflineq, which solves solve a linear equation in GF(q) field

Note that by using MATLAB “/” or “\”, you may not be able to obtain a linear equation solution on GF(q). For example:

```
% Using \ in MATLAB finds solutions that are not in GF(3).
a = [0 2 1; 0 2 2; 2 1 0];
b = [0; 1; 2];
x1 = a \ b
x1 =
1.25
-0.5
1

% Use gflineq to obtain correct solutions in GF(3).
x2 = gflineq(a,b)
x2 =
2
1
1
```

In the toolbox, two functions, `primes` and `isprime`, are dedicated to finding prime numbers and checking whether a number is prime. Please see Chapter 5, the *MATLAB Function Reference*, for examples of using these two functions.



4

Examples

Bit-Error Rate Computation 4-3
Background 4-3
Open the COMMGUI Window 4-3

Digital Subscriber Telephone Lines (DSL) 4-7
Channel and Noise Models 4-8
The Transceiver Architecture 4-9
Performance Simulation 4-10
References 4-14

Early-Late Gate Synchronization 4-15
Background 4-15
Communication System Simulation 4-18

V.34 Modem Data Transmission 4-29
Background 4-29
Simulation Block Diagram 4-30
Parameters and Limitations of the Simulation Design . . . 4-32
Call Modem Simulation Implementation 4-35
Answer Modem Simulation Implementation 4-43

This chapter provides four application examples. The four examples are:

- Bit-error rate computation
  This example is a simple tool for computing bit-error rate with different modulation methods, error control coding methods, and channel noises.
- Digital subscriber telephone lines
  This example simulates digital transmission through a telephone line. It includes noise cancelation and equalization models.
- Early-later gate synchronization
  Synchronization plays a vital important role in the data transmission. This example implements a complete synchronization system for receivers.
- V.34 modem data transmission
  V.34 modem standard is the newest standard for computer data transmission through telephone lines. This example simulates the data transmission between a call modem and an answer modem.

MATLAB and Simulink are widely used in industrial applications. The author of this toolbox is planing to collect application examples in the communication area and have them published in the form similar to the examples provided in this chapter. Please send your examples to the author at the following address if you are interested in having them published:

Wes Wang
The MathWorks, Inc.
24 Prime Park Way
Natick, MA 01760
`wang@mathworks.com`

Bit-Error Rate Computation

Background

This section provides an example of how to use the toolbox to calculate bit error rates. It provides a flexible tool for the bit error rate computation using different methods and parameters. You can choose to use techniques provided in the toolbox or your own method to compute the bit-error rate.

Open the COMMGUI Window

Open the graphical user interface (GUI) window by typing the command

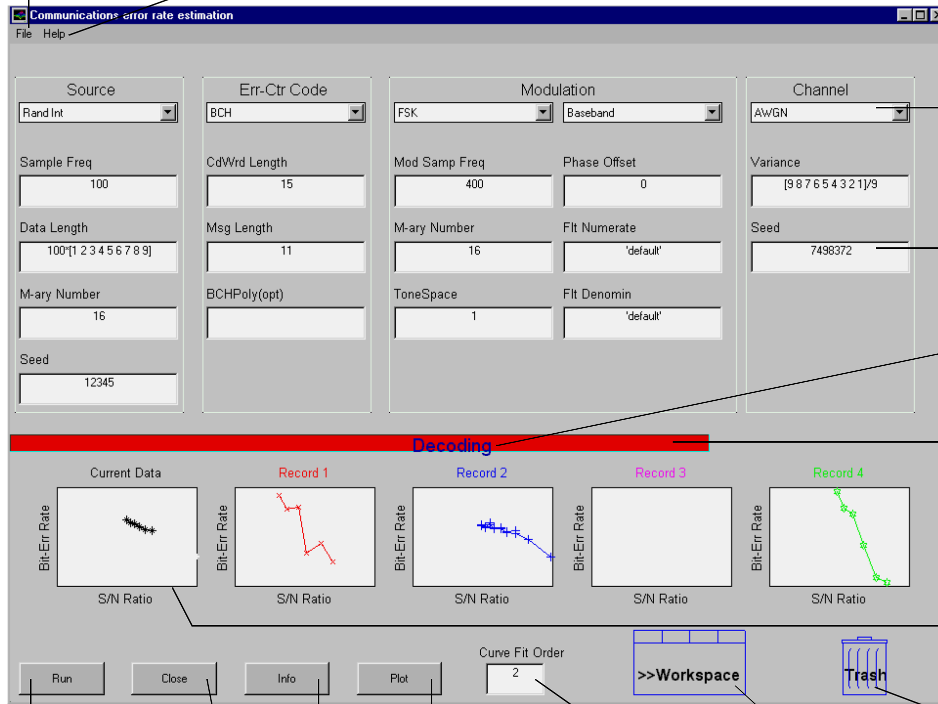
```
commgui
```

in the MATLAB workspace. You can also double-click on the Bit-Error Rate Computation block in the `demo` sublibrary in the Communications Toolbox block library.

This figure shows the GUI window:

Load or save the setting data.

Help for this GUI example.



Processing method selections (editable).

Parameter entry (editable).

Computation stage.

Calculation percentage.

Record of past computation results.

Record of current computation results.

Start computation. The processing stage color indicates the computation status.

Close this GUI figure.

Help for how to use the GUI.

Plot the computation result. The plot is enabled only if computation data is available. The plot is on a log-log scale.

The order of the curve fitting in plot (editable).

Copy workspace variable to record data. Or, copy the record data to workspace.

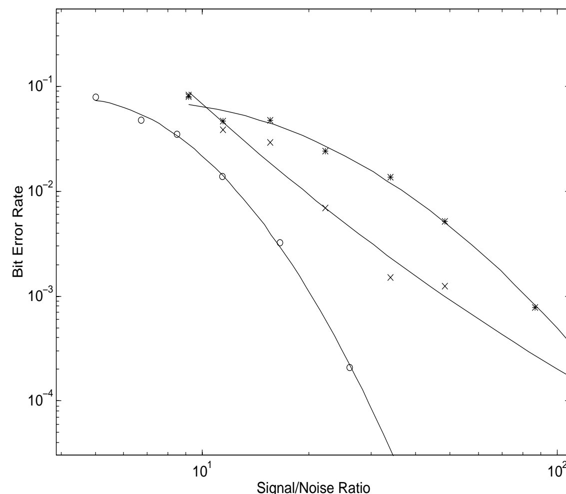
Sending record data to trash deletes the record data.

This window includes four different functional categories:

- Signal Source
- Error-Control Coding
- Modulation/Demodulation
- Channel

You can choose computation methods in each of the four categories. The methods can be found in the pull-down menus in each category. Once you have selected a method in a category, you can specify the parameters associated with that particular method in the data entry fields located beneath the pull-down menus. For example, the first parameter in the Channel category can be a vector signal, which determines the computation length.

Buttons to control the execution are located at the bottom of the window. After setting the methods and parameters, click the **Run** button to start the computation process. The execution function automatically fills the computation results into the current data record. The computed data includes the signal to noise ratio and the bit-error ratio. The computation stage, which is located directly beneath the group of functional categories, indicates which stage the calculations have reached in the processing. The calculation percentage indicates what percent the total calculation has occured. After the computation is done, clicking the **Plot** button opens a figure that plots the bit-error rate.



The plot is on a log-log scale. The first, second, third and current record data are indicated by the '*', 'x', '+', and 'o' symbols respectively. No curve is drawn if the record is empty. The curves are the result of curve fitting with the given data. The curve fitting polynomial order can be specified by changing the Curve Fit Order parameter.

You can rearrange the data by dragging and dropping the data record between records and the workspace area. When you drop data into the trash can, which is located in the lower right-hand corner of the GUI window, the data will be deleted. You can zoom in and out of the plot window. Ordinarily, MATLAB plots the default to zoom off, but in this demo the default is zoom on. Refer to the *MATLAB Reference Guide* or type `help zoom` on the MATLAB command line for a discussion of the `zoom` function.

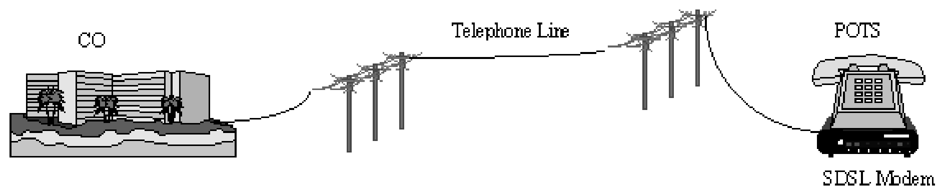
Please note that the chosen parameters must fit the requirements of the computation. If you don't know a good value for a parameter, leave it empty. The function will choose the default computation parameter for you.

You can save the window parameter setting into a data file by choosing the **File** pull-down menu at the top left corner of the window. You may load saved data after you have saved a window setting. The **Help** pull-down menu provides online help for how to use the function.

Digital Subscriber Telephone Lines (DSL)

This example is provided by Walter Y. Chen of Texas Instruments, Inc. (13510 North Central Expressway, Mail Stop 446, Dallas, TX 75265). The author of this toolbox has revised the design of the example.

Digital Subscriber Lines (DSLs)[1][2][3] are used to connect telephone subscribers to local central offices (COs) over the traditional twisted pair telephone subscriber lines. These digital connections can provide symmetrical or asymmetrical transmission with a throughput between a few 100 kbps to a few Mbps.



The Integrated Services Digital Network (ISDN) basic rate access channel available now through telephone companies is based on the Digital Subscriber Line (DSL) technology. That DSL has a net (echo cancellation based) duplex transmission throughput of 160 kbps. The transmission is carried out with a 2B1Q line code with a baud rate of 80 kHz. The base transmission spectrum is from DC to 40 kHz.

The repeaterless T1 service is based on the High bit-rate Digital Subscriber Lines (HDSL) technology. HDSL employs the same 2B1Q line code as does the DSL but with a higher baud rate of 400 kHz. This results in a base transmission bandwidth of about 200 kHz. The higher transmission throughput is achieved by using higher speed signal processing hardware in conjunction with a smaller serving distance.

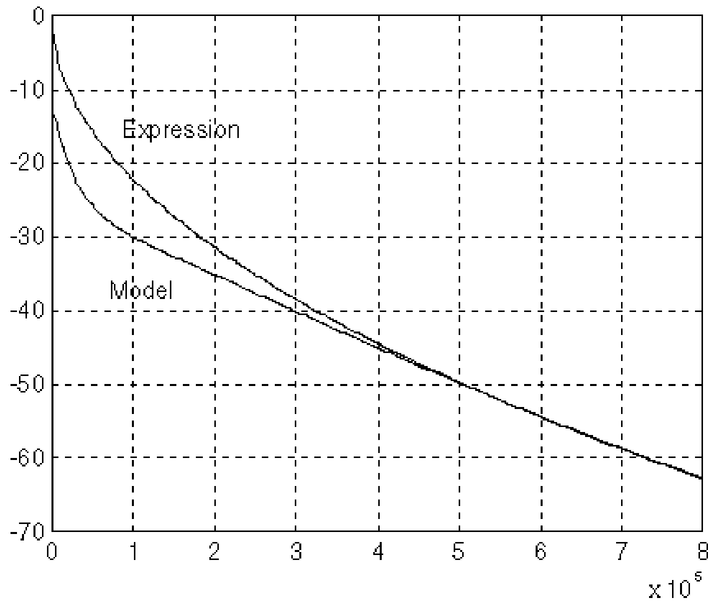
The maximum transmission throughput of a DSL system is limited by the subscriber loop channel loss and the corresponding noise environment. For DSL and HDSL the noise environment is dominated by self Near End Crosstalk (NEXT). The self NEXT is defined by the interference from adjacent transmitters with similar DSL systems at the same central office or residence.

Channel and Noise Models

A typical telephone subscriber line has a length of nine kft (26 AWG). The squared channel transfer function can be approximated as

$$|H_e(f)|^2 \approx e^{-2d\zeta\sqrt{f}}$$

for $\zeta = 9 \times 10^{-7}$, and d=9000 ft.



Besides the channel loss, the transmission performance is also limited by channel noise. It consists of Near End Crosstalk (NEXT), Far End Crosstalk (FEXT), and white background noises. The NEXT and FEXT noise power is dependent on the coupling function and the disturbing source power density. The squared NEXT coupling transfer function is

$$|H_{\text{NEXT}}(f)|^2 = \chi f^{3/2}$$

where $\chi = 8.8 \times 10^{-14}$ for 49 disturber 1% worst case. The value of χ should be modified according to

$$\chi = \chi_{49} \frac{n^{\frac{6}{10}}}{10}$$

for different number, n, of disturbers.

The squared FEXT coupling transfer function is

$$\left|H_{\mathrm{FEXT}}(f)\right|^2 = \psi f^2 d \left|H_e(f)\right|^2$$

where $\psi = 8 \times 10^{-20}$ for 49 disturber 1% worst case and the same scaling method also applies to ψ.

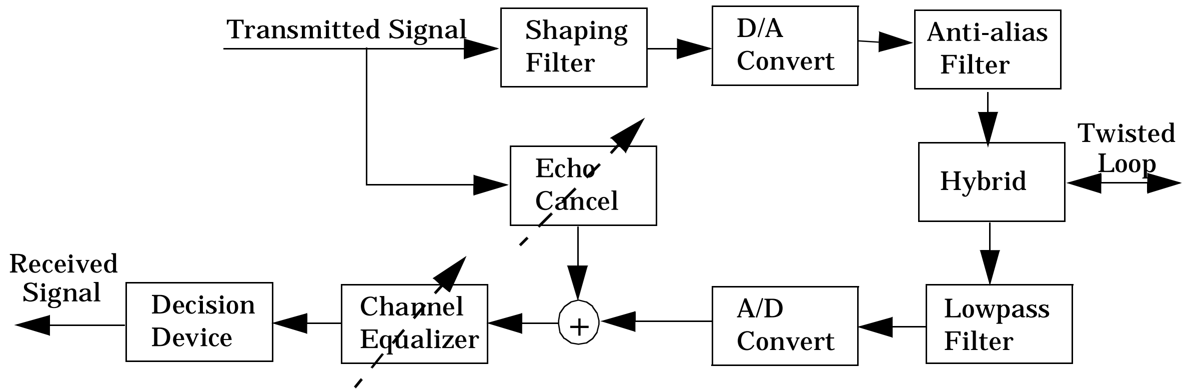
The power spectrum density of the ISDN basic access rate service can be expressed as

$$\mathrm{PSD}_{\mathrm{ISDN}} = 3.017 \frac{f_s}{f_0} \frac{\sin^2 \frac{\pi f}{f_0}}{\left(\frac{\pi f}{f_0}\right)^2}$$

for $f_0 = 80$ kHz.

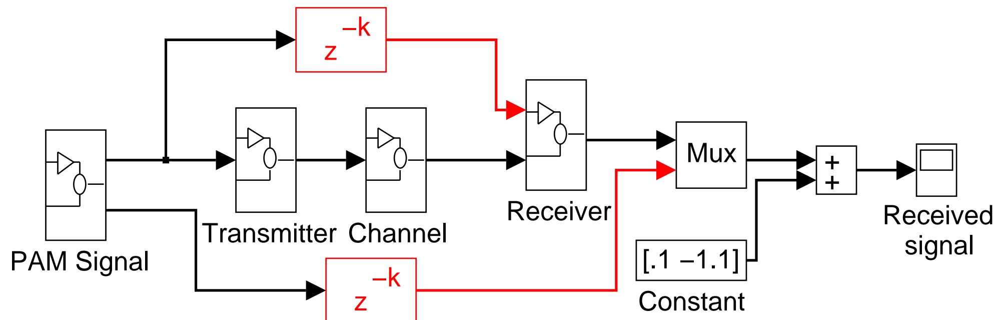
The Transceiver Architecture

The echo cancellation based DSL transceiver can be realized with the multi-level pulse amplitude modulation technique. The transceiver structures at each end of a DSL system are identical. The transmitter part of a DSL consists of a digital shaping filter, a D to A converter, and an anti-aliasing filter. The receiver part consists of an analog lowpass filter, an A to D converter, and a baud rate channel equalizers. An echo canceller is used to separate the received signal from the transmit signal.



Performance Simulation

The DSL simulation can be carried out with transmitter, channel, noise, echo path, and receiver models.



The transmitter is simply a combination of a raised-cosine filter and a PAM signal source. The channel consists of a loop model and a noise model, both of which are presented as FIR filters, in conjunction with a noise generator. The channel model is generated by first calculating the channel transfer function and then the channel impulse response for the FIR filter via Discrete Fourier transform. The noise model is calculated by designing an FIR filter to fit the spectrum of the noise environment which is the cascading of the HDSL

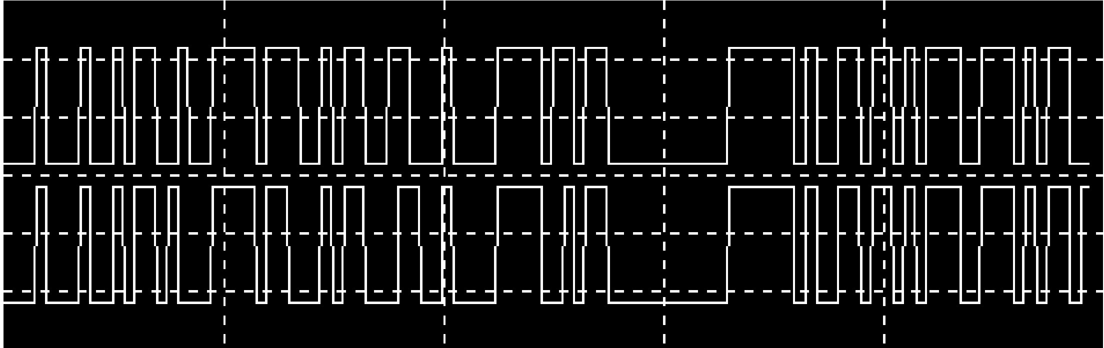
transmit power spectrum and the squared NEXT coupling transfer function. The effect of noise is then simulated by driving this FIR filter with a white Gaussian noise whose power is adjusted to 0 dBm/Hz. A simplified echo path model consists of an over damped second order system. The echo model is also represented by an FIR filter. The transfer function of the echo path is

$$H(s) = \frac{1}{2} - \frac{R_{12}}{Z_0 + R_{12}} \frac{s\left(s + \frac{1}{C_1 R_2}\right)}{s^2 + \left(\frac{1}{C_1(R_{Z1} + R_2)} + \frac{R_{Z12}}{L_p}\right)s + \frac{R_{Z1}}{C_1(R_{Z1} + R_2)L_p}}$$

where $Z_0 = 135\Omega$, $R_{12} = 99\Omega$, $R_2 = 124\Omega$, $R_{Z1} = 106\Omega$, $R_{Z12} = 57\Omega$, $C_1 = 0.033\mu F$, $L_p = 1.5\,mH$.

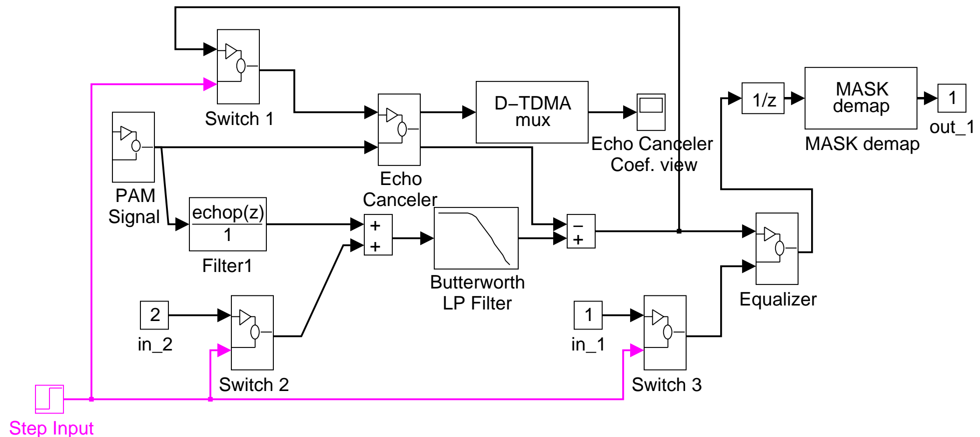
The receiver is a combination of a Bessel lowpass filter and an adaptive channel equalizer. The receiver also has an echo canceller that cancels echos generated by the combination of the echo path model and another PAM signal source. The transmitted signal is relayed to the receiver for the initial equalizer training. With the absence of the far end signal, the echo canceller is first trained to reduce the echo level by about 60 dB. After the initial convergence of the echo canceller, the adaptation step size of the echo canceller is reduced to a much smaller value to keep tracking the possible variations of the echo path transfer function with the presence of the received signal.

Simulation results can be examined by displaying transmitted and received signal, equalizer output error, equalizer coefficients, and echo canceller coefficients. A snap shot of the transmitted and recovered signal is shown in the next figure. An artificial shift has been added to distinguish the two different signals.

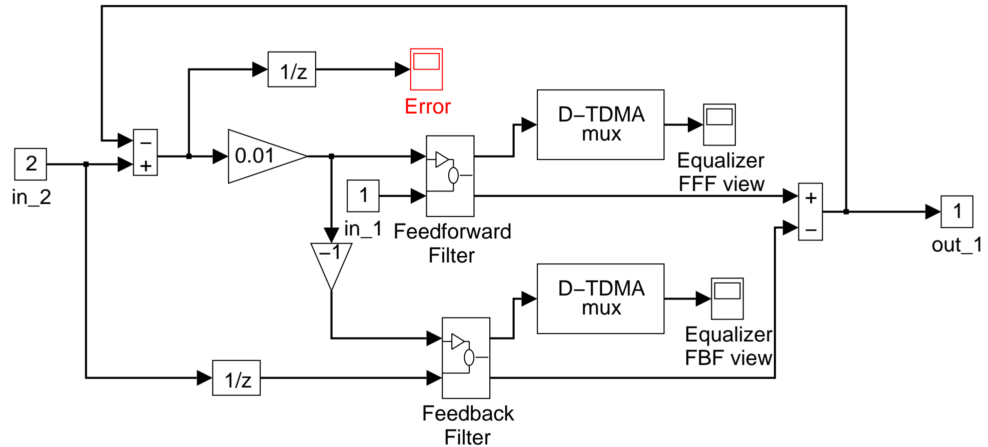


The Receiver

The block diagram of the receiver is shown below. The receiver contains echo cancellation, an equalizer, and a low pass filter.

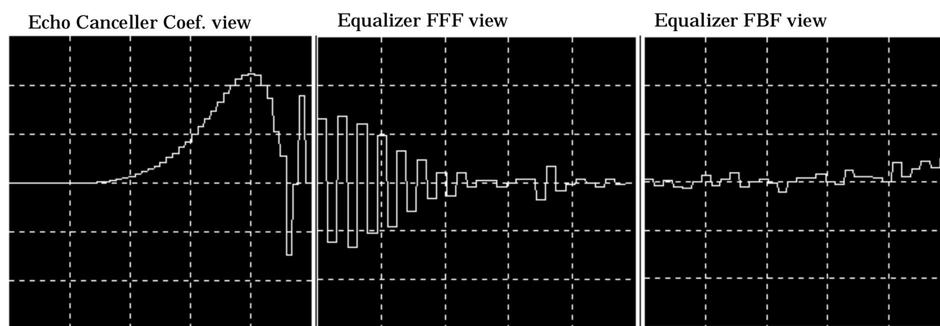


You can open the equalizer block to look into the block diagram structure of the equalizer. The equalizer is mainly constructed by using a feedforward filter and a feedback filter. The structure of the equalizer is shown in the following figure.



The Step Input block in the Receiver subsystem provides a switch signal. Before the step signal becomes one, the system is in a training period. The equalizer and echo cancellation parameters are trained in the training period. This example has placed the trained parameters as initial conditions at the beginning of the simulation, which enable you to see the simulation result without waiting to the end of the training period. If you prefer to see the training period, you can set the parameters `X1_ini`, `X2_ini`, and `X3_ini` to be zero vectors in the MATLAB workspace. You should set the Step Time in the Step Input block to be 50 seconds instead of 0 seconds. Before the 50 seconds, it is the training period. You can open the Echo Canceler Coef. View scope (in the Receiver subsystem), the Equalizer FFF View scope (in the Equalizer subsystem), and the Equalizer FBF View scope (in the Equalizer subsystem) to view the parameter training process. A snap shot of the parameters during the

training is shown in the figure below. Note that the figure displays the values of each element in the vector:



References

[1] *IEEE Journal on Selected Areas in Communications*, Special Issue on High Speed Digital Subscriber Lines (HDSL), Vol. 9, No. 6, August 1991.

[2] *IEEE Journal on Selected Areas in Communications*, Special Issue on Copper Wire Access Technologies for High Performance Networks, Vol. 13, No. 9, December 1995.

[3] Chen, W.Y, “Architecture and performance simulation of a single pair HDSL,” *IEEE ICC ‘96 Proc*.

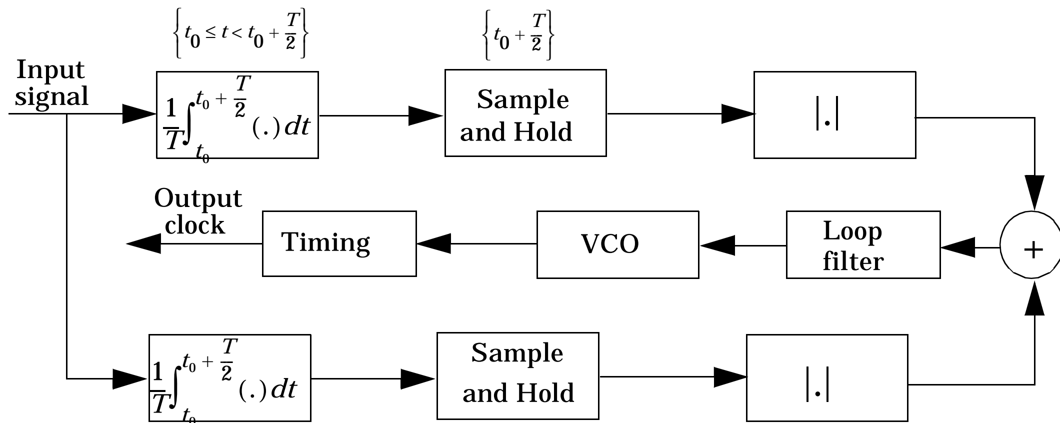
Early-Late Gate Synchronization

This example is provided by Darrel Judd of Harman Motive (1201 South Ohio Street, Martinsville, IN 46151).

When a communications signal is received, it must be synchronized to the system that receives it. Depending on the type of demodulation scheme used, it may be important to know the correct carrier frequency and phase. This is called carrier synchronization. If the signal is a data signal, it is also important to synchronize to the symbol periods of the data. This is called data synchronization. There are many ways that the synchronization problem has been solved: pilot tones, preambles, phase locked loops, etc. This example assumes that the signal of interest is a two tone FSK signal. It is also assumed that the baud rate is known, but the starting point of each bit is not known. This tutorial uses an early-late gate technique to baud synchronization to the instantaneous frequency output of an FM detection process. The Simulink function is called `elg.m`.

Background

The early-late gate synchronizer is a very popular technique. A block diagram of the algorithm is shown below. This version works for rectangular pulses.

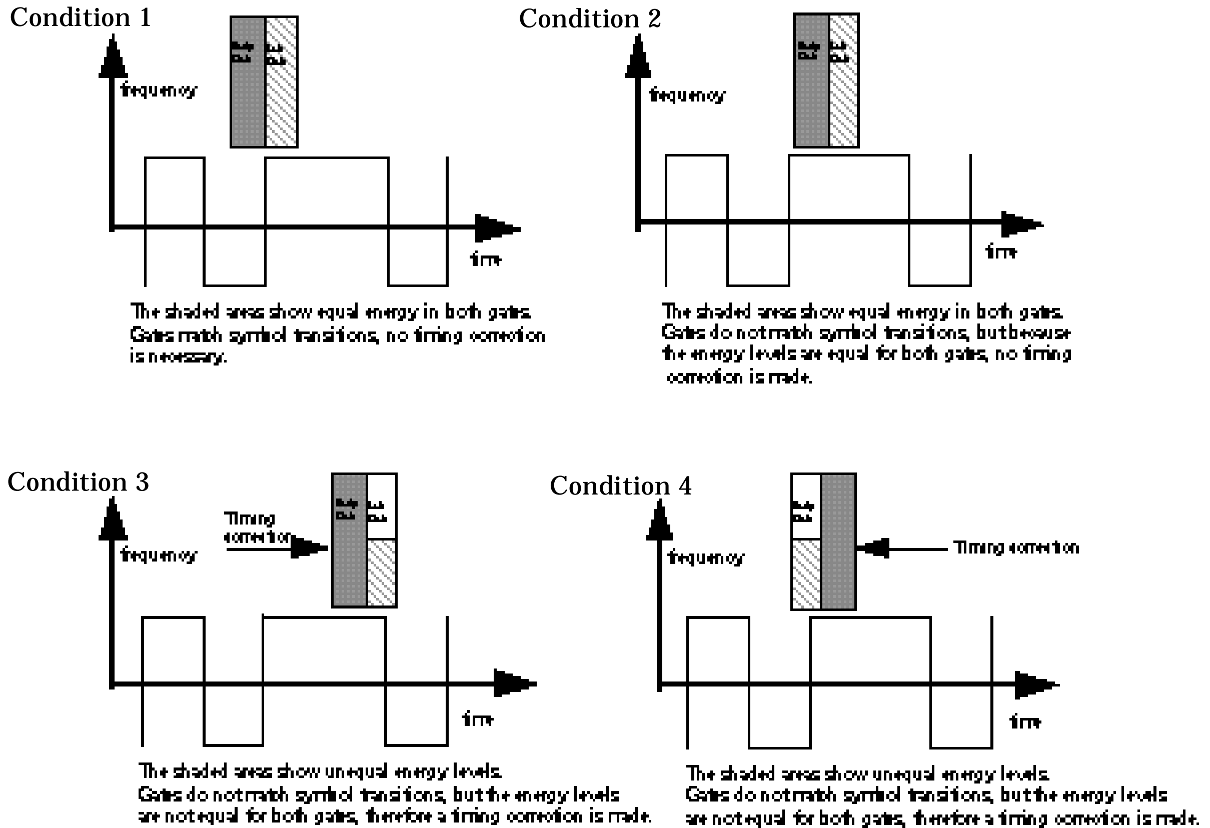


The basic principle for this data synching technique is to compare the energy in the first half of a symbol period to the energy in the last half of a symbol

period. (“Early” and “Late” gate, respectively). There are four conditions that can occur:

- Condition 1. The loop has locked to the symbol period. The energy levels are equal. This means that a symbol transition was not found in this symbol period. No timing correction is made for this condition.
- Condition 2. Two identical symbols have been transmitted. The energy levels are equal. This means that a symbol transition was not found in this symbol period. No timing correction is made for this condition.
- Condition 3. The energy in the early gate is greater than the energy in the late gate. This means that there was a symbol transition in the last half of a symbol period. A timing correction can now be made to synchronize up to the symbol boundary.
- Condition 4. The energy in the late gate is greater than the energy in the early gate. This means that there was a symbol transition in the early gate. A timing correction can now be made to synchronize up to the symbol boundary.

These four cases are shown graphically in the following figure.



The basic steps of the algorithm are listed below:

1 Given a data signal s(t), choose an arbitrary point in time, t_0. This point lies somewhere within a symbol boundary. Let this point be $s(t_0)$.

2 Let the symbol period be T, where 1/T is the symbol rate.

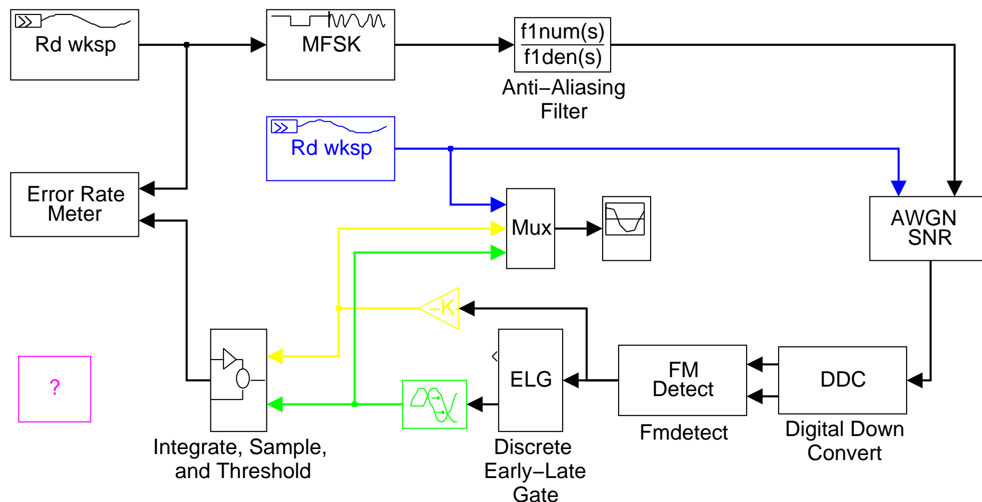
3 Now define a window in time as $t_0 \le t \le t_0 + \frac{T}{2}$. This window is called the early gate.

4 Now define a window in time as $t_0 + \frac{T}{2} \le t \le t_0 + T$. This window is called the late gate.

We now have two adjacent windows that cover one symbol period, but the starting point of the symbol is not known. If we look at the amount of energy in each gate, we can deduce if it is at a symbol boundary. If the energy in the early gate is greater than the energy in the late gate this means that the signal transitioned during the late gate. Our point of reference, could now be decreased and we would be one step closer to knowing the symbol boundary.

Communication System Simulation

To run this simulation as it is described, open the Demo window in the Communications Toolbox Main Window. When the Demo window is opened you should see a window labeled **Early Late Demo**. This figure shows the window's contents:



The simulated system can be broken down into four basic areas:

- The generation and modulation of a digital signal. This is shown in the first three blocks.
- The transmission of the signal through a communications channel.
- The demodulation of the received signal. This process is represented by the blocks in the lower half of the figure.
- The monitoring of the results. Results are displayed in a graph window and with an Error Rate Meter window.

The basic parameters of the simulation; sample period (`tss`), symbol period (`tdd`), carrier frequency (`Fc`), tone spacing (`Fshift`), and number of tones (`MM`), can be modified from the MATLAB workspace after the `elg` window has been opened.

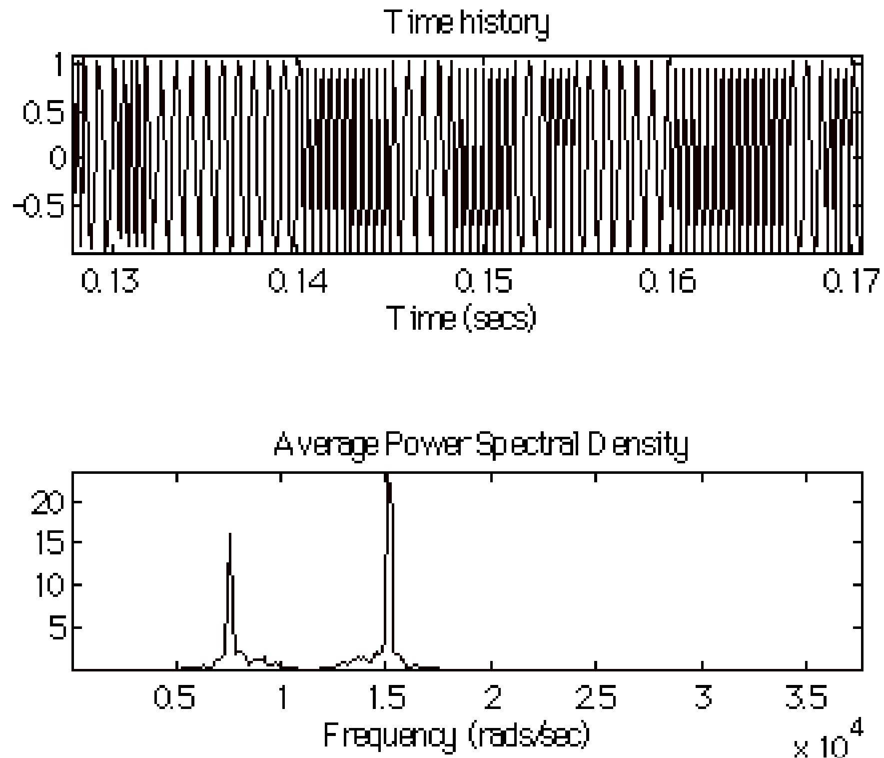
The first step in the simulation is to generate a digital communications signal. A two-tone FSK signal is generated. Next, noise is added to the signal with the block, AWGN SNR. On the receiving side, the signal is first down converted to remove a carrier and separated into quadrature components. This is done with the block DDC. The instantaneous frequency of the baseband signal is then calculated using the FM Detect block. Now the processed signal is ready to be input into the Discrete Early-Late Gate block. The outputs of this block are two timing signals. These signals are synchronized to the symbol boundaries of the detected FSK signal. These timing signals are then used to extract the original transmitted symbols from the instantaneous frequency. The results are then compared to a delayed version of the original digital message with the Error Rate Meter block. The rest of the blocks are used to display the results of the simulation. The Transport Delay block is needed because there is an inherent phase delay caused by the Phase Locked Loop used in the Discrete Early-Late Gate block.

Signal Simulation

The two-tone FSK signal is generated by the top left three blocks in the system block diagram. The Rdwksp block is set up to give continuous random numbers ranging between 0 and 1. The output of this block is sent to the MFSK Mod block. The binary data is also sent to the Error Rate Meter block so that the original data can be compared with the demodulated data later in the simulation.

The MFSK Mod block modulates the digital data stream according to the parameters listed earlier.

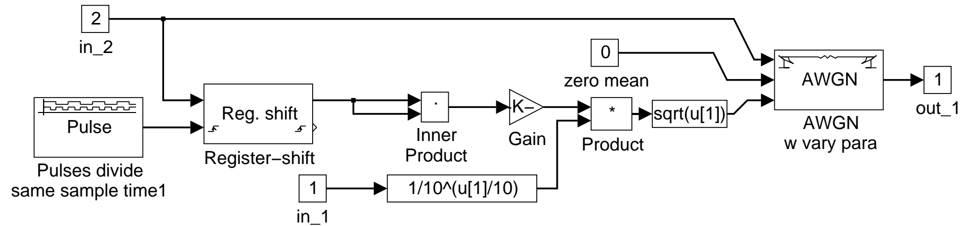
Because rectangular pulse shaping is used for this simulation, and because a rectangular pulse is time limited, the resulting signal will have infinite bandwidth in the frequency domain. In order to solve this problem, an anti-aliasing filter is added to limit the bandwidth of the signal. The bandwidth is chosen to be 4000 Hz. It should be noted that if `tss` is changed for this simulation that the Anti-Aliasing filter roll-off frequency will need to be changed in the Mask for the FSK2 block. The output signal of this block, and its power spectral density are shown below.



Communications Channel

The block diagram of the AWGN SNR block is shown in the next figure. This block adds additive white Gaussian noise (AWGN) to the signal at input port 1. The signal to noise ratio (SNR), in dB, is determined by the value present at

the second input port. During the system simulation, the SNR ranges from -5 dB to 10 dB. The SNR level is incremented by one dB every 20 symbol periods. By changing the SNR range, an error free operating range for the system can be determined:



Signal Demodulation

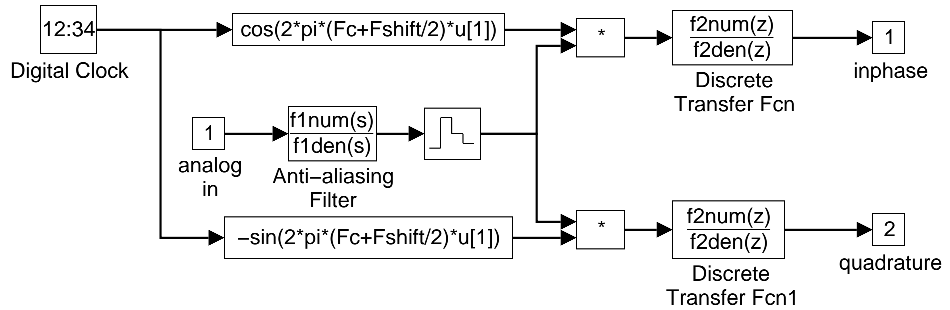
The signal demodulation process is represented by the blocks in the lower half of the el g block diagram.The demodulation is accomplished by Digital Down Convert, FM Detect, Discrete Early-Late Gate, and Intergrate Sample and Threshold blocks.

Digital Down Converter. Once the signal has passed through the communications channel, it must be sampled and down converted to baseband. This is done with the DDC block. The analog input is first filtered to ensure that it is band limited. Then it is sampled with a Zero-Order Hold block.

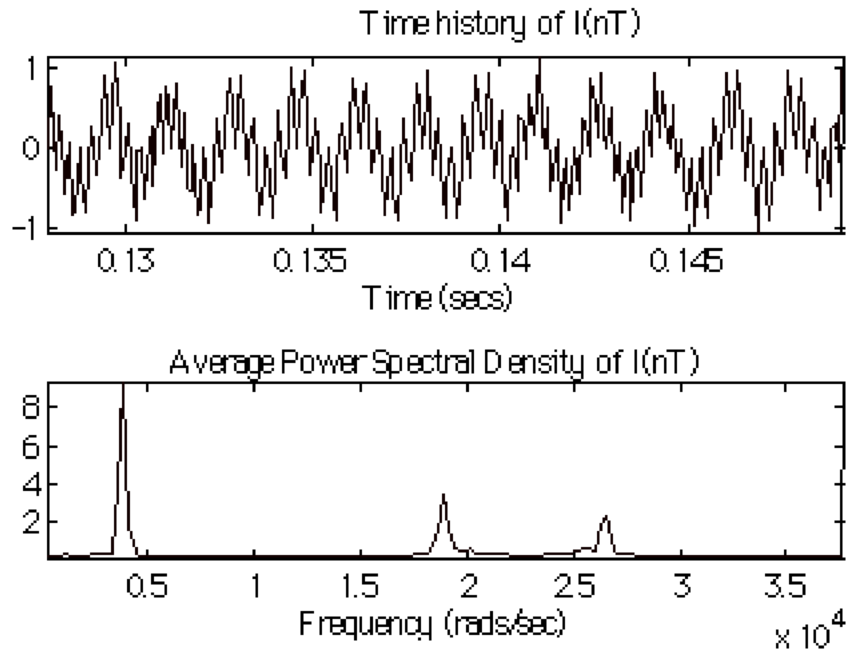
At this point, the signal processing is best described by the following equations:

$$s(t) = s(nT)$$
$$I(nT) = \frac{A}{2}\cos(\pm\omega_c nT) + \frac{A}{2}\cos((\omega_c \pm \omega_s)nT)$$
$$Q(nT) = \frac{A}{2}\sin(\pm\omega_c nT) + \frac{A}{2}\sin((\omega_c \pm \omega_s)nT)$$
$$s(nT) = A\cos((\omega_c \pm \omega_s)nT)$$
$$I(nT) = s(nT)\cos(\omega_c nT)$$
$$Q(nT) = -s(nT)\cos(\omega_c nT)$$

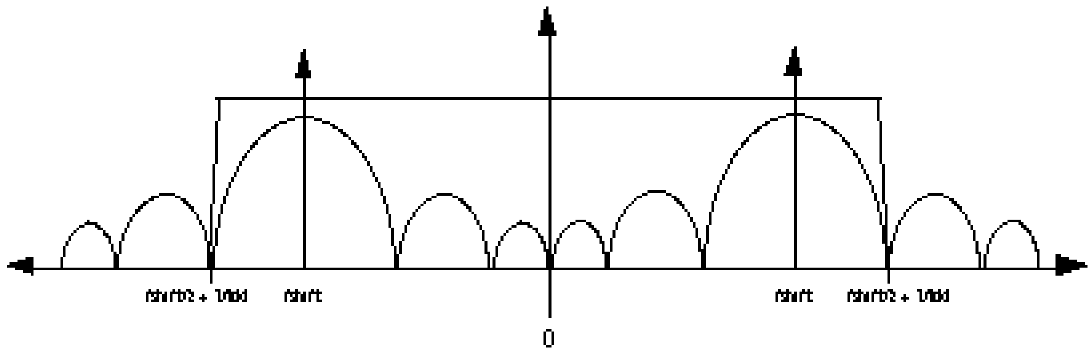
The overall process for the DDC is shown in the following figure.



A time plot and power spectral density plot of I(nT) are shown in the figure below, note the twice carrier peaks, $2\omega_c \pm \omega_s$, in the power spectral density.

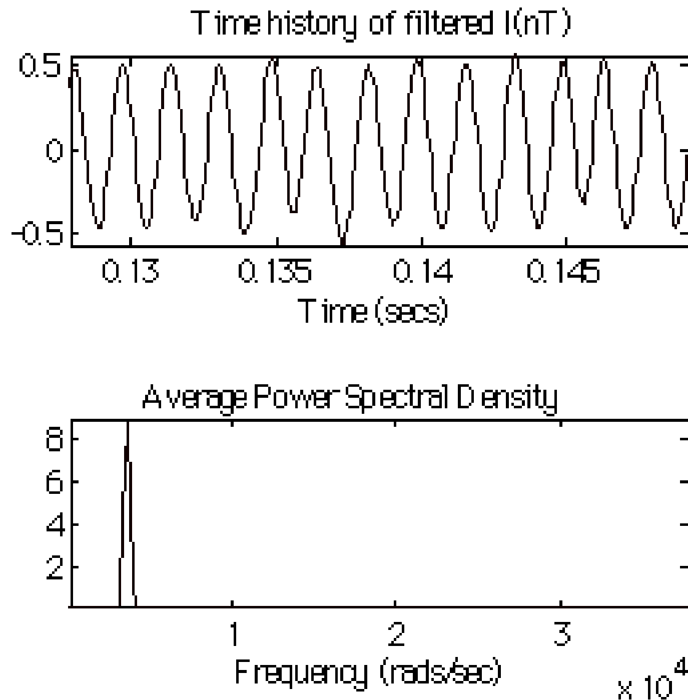


The $2\omega_c + \omega$ terms are filtered out of I and Q with identical low pass filters. The cut off frequency for these filters is determined by the symbol rate used. An example of how this is done is shown in the next figure.



The cutoff frequency is chosen such that the main lobes of the spectrum and tone spaces are included in the pass band. If the `fshift` frequency between the two tones is large enough, a band pass filter can be placed around each main lobe. This helps to eliminate intersymbol interference. Another solution to the symbol interference problem is to use a pulse shaping filter. The results of the filtering process are shown in the below figure. Note that the $2\omega_c$ terms have

been filtered out. The quality of this filtering determines the spur-free dynamic range of the DDC:.

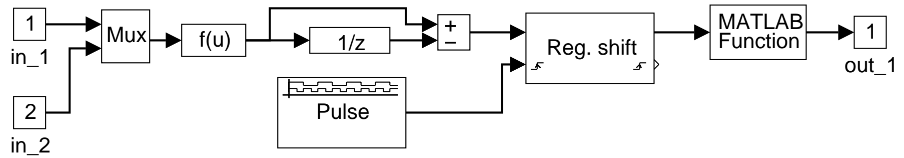


The FM Detect Block. The instantaneous frequency is calculated from the complex signal according to the following equation:

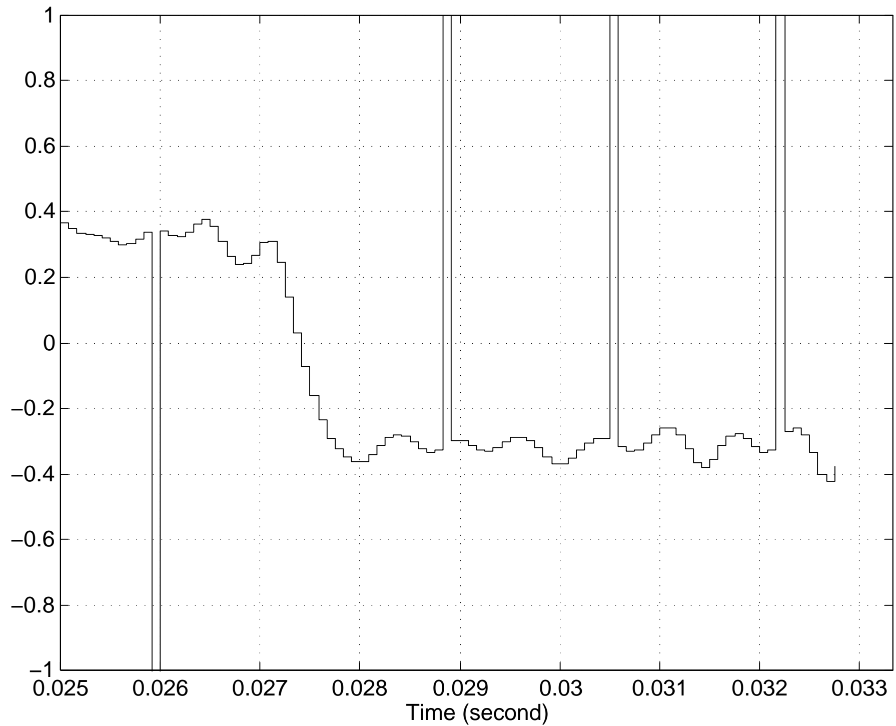
$$freq(nT) = \frac{d\left(a\tan\left(\frac{Q(nT)}{I(nT)}\right)\right)}{dt}$$

This method works well for high signal-to-noise ratio conditions.

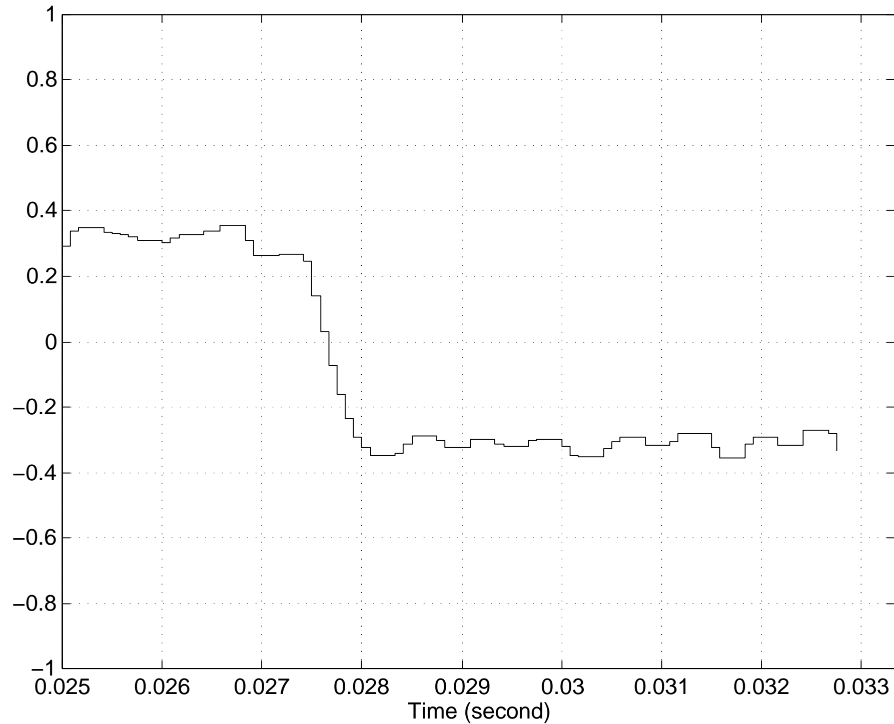
The FM Detect block is shown in the following figure. The derivative is simulated with a first order difference consisting of a unit delay and subtract.



A problem with this technique is that when the phase of the complex signal moves from quadrant 3 to quadrant 2 or 4, large noise spikes are generated in freq(nT). This is shown in the figure below:



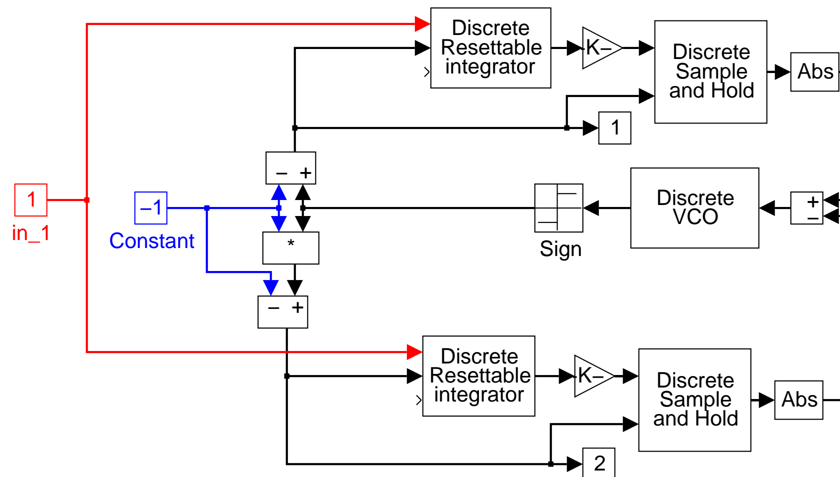
These spikes are easily removed with a five tap median filter. The median filter is implemented with the Register Shift block and the MATLAB Function block. The figure shows the results of the filtering:



The Discrete Early-Late Gate Block. The diagram for the Early-Late Gate block is shown in the next figure. There are two outputs for the block:

- A square wave representing the early gate
- A square wave for the late gate

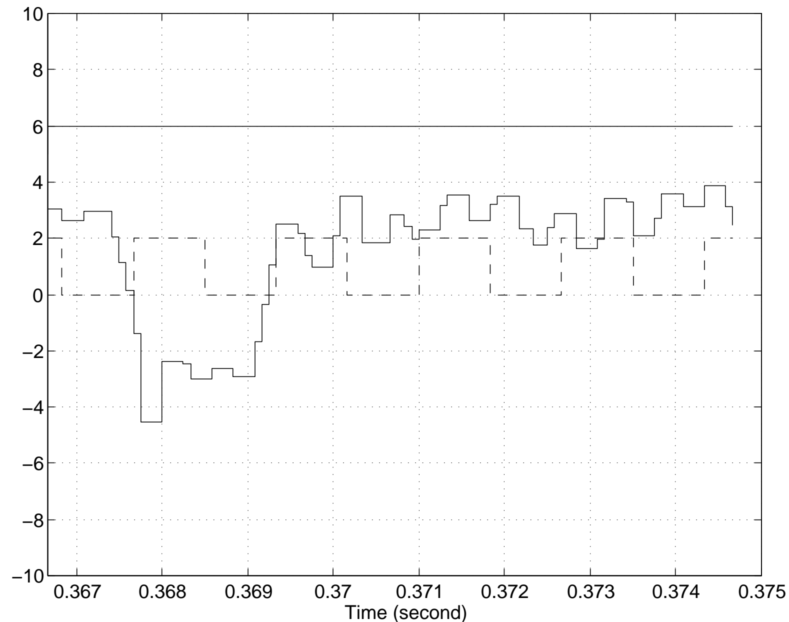
The symbol boundary is at the falling edge of the early gate and at the rising of the late gate:



The output of the Discrete Early Late Gate block, and the instantaneous frequency output from the FM Detect block are shown in the next figure. The early and late gate wave forms are aligned with the symbol boundaries. The other solid line is the SNR level. For this plot the SNR was 6 dB.

The Early Late Gate technique of synchronization can be used for other kinds of demodulation techniques. The FM Detection process could be replaced with

a matched filter. The energy output from each filter could then be used to generate the error function.



The symbol boundaries are determined by the Early Late Gate block. The outputs from the Early Late Gate block can now be used to trigger an “integrate and dump” over each symbol period. The output of the integrator can then be compared to a threshold to determine each symbol.

Each demodulated symbol is now compared to the transmitted symbol. An error count and rate is tabulated with the Error Rate Meter block.

Note: Please refer to F.M. Gardner’s *Phase-Locked Loops* for a more detailed discussion on early-late synchronization.

V.34 Modem Data Transmission

Background

Modems are designed to transfer data between computers through telephone lines. To ensure the correct transmission of data between two different modems, the International Telegraph and Telephone Consultative Committee (CCITT) has issued a number of standards for data communication over telephone networks. The standards for modems are specified with the initial character V, known as the V series. V series are useful for data protocol, data translation, data compression, etc. The standards are as follows:

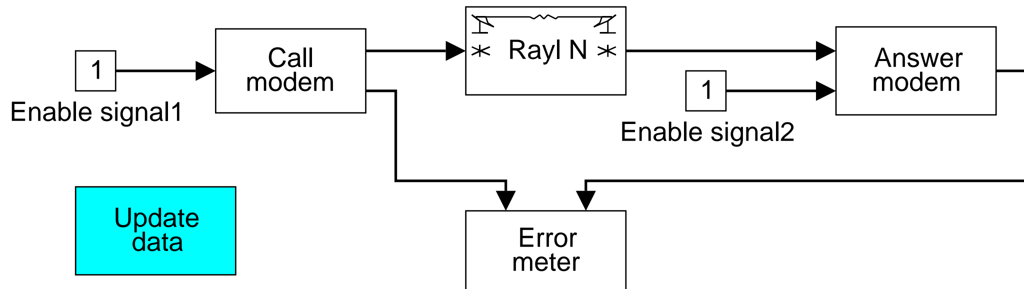
V. Series	Functionality
V.1-V.8	General description of coding, symbol rate, and power levels for modem.
V.10-V.34	The signal conventions standard for modems using voice frequency telephone lines.
V.36-V.38	The signal conventions standard for wide band modems.
V.41-V.42	Error-control convention.
V.50-V.57	Testing method, noise measurement specification.
V.100, V.110, V.120, and V.230	Internetworking specifications with packet and ISDN networks.

The V.34 modem is a standard from CCITT (now, referred to as the International Telecommunication Union-Telecommunication (ITU-T) Standardization Sector) for telephone line data transmission published on September, 1994. It is the latest modem standard available. The maximum bit transmission rate is 28,800 bits per second. The following example shows an implementation of the V.34 modem. The example is designed to clearly follow the structure described in the standard. Time efficiency and speed are not the key considerations in the implementation. You can use this example to understand the detailed structures and data flow of a V.34 modem. You can also modify this example to construct your own implementation.

Refer to U. Black's *The V Series Recommendations* for a detailed discussion of the CCITT standard before V.34.

Simulation Block Diagram

This example simulates the data translation process only. The modem start-up training before the data translation is not included. The example assumes that the parameters between the Call Modem and the Answer Modem are the same.



In this example, the Call Modem transmits data to the Answer Modem. The data to be transmitted are random binary numbers generated inside the Call Modem. The first output port of the Call Modem outputs the baseband QAM modulated signal. The transmission channel is assumed to be a Rayleigh noise channel. The Answer Modem uses the received signal to recover the signal transmitted over the telephone line. It is assumed that the Answer Modem shares the same parameters as those used in the Call Modem. The enable signals are placed to indicate the data translation period.

The second output port of the Call Modem contains integers that are sent to the Answer Modem. The sample rate of the data is:

```
.28/v34_rate_p/v34_rate_j/4
```

The integer contains `v34_rate_b/4` bits of the message. (The next section details the definitions of the parameters `v34_rate_p`, `v34_rate_j` and `v34_rate_b`.) The output of the Answer Modem has the same format as the second output of the Call Modem.

The simulated data are compared in the Error Rate Meter block. Note that there is a time delay between the Call Modem data and the Answer Modem recovered data. You may need to tune the parameter Delay between the input and output in the Error Rate Meter block.

The Call Modem and the Answer Modem are designed as ready-to-use blocks. You can input parameters to the dialog box for different simulation parameters. The dialog box is as follows:

Data transfer for calling modem (Mask)

Block name: Call modem

Block type: Data transfer for calling m

The input signal enable the block to transfer data.

OK

Cancel

Help

Symbol rate, S (2400, 2800, 3000, 3200):

3200

Data rate, R (2400, 4800, 9600, 14400, 28800):

28800

Convolution encoder (1:16 state; 2:32 state; 3:64 state):

1

Precode filter coefficient (1x3 vector):

[1 .8 .4]+sqrt(-1)*[1 .5 .2]

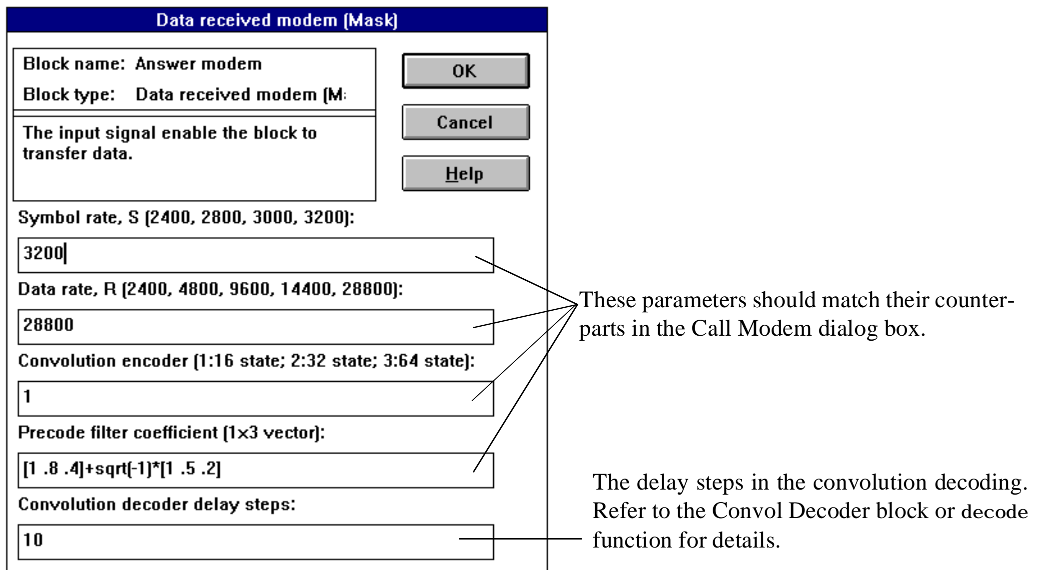
Date transfer symbol rate. The valid entry numbers are 2400, 2800, 3000, and 3200 in this simulation.

Date transfer bit rate. The valid entry numbers are 2400, 4800, 7200, 9600, 12000, 14400, 16800, 19200, 21600, 24000, 26400, and 288000 in this simulation. Read the “Parameters and Limitations of the Simulation Design” subsection for the limitations of the rate assignment.

Use of different convolution encoders. The valid numbers are 1, 2, and 3, which indicate the 16, 32, and 63 state convolution encoder is used.

A 1×3 complex vector, which contains the coefficients of the precoder filter.

The dialog box for the answer modem is:



You can use the block directly for the simulation.

Like most other blocks in the toolbox, the modem simulation blocks are constructed by using the combination of Simulink built-in blocks and S-functions. The construction of the blocks takes advantage of the ready-to-use blocks in the Communications Toolbox. To see the detailed design of the blocks, unmask the blocks and double-click the block.

Parameters and Limitations of the Simulation Design

You should read the ITU-T recommendations to understand fully the system design, parameter settings, and the techniques shown in the modem simulation. You can order the standard from ITU-Sales Department, Place des Nations, CH-1211 Geneve 20 Suisse, Geneva, Switzerland (4122) 730-5285.

In the following discussion, tables and figures using the reference V.34 mean that the table or figure can be found in the V.34 standard.

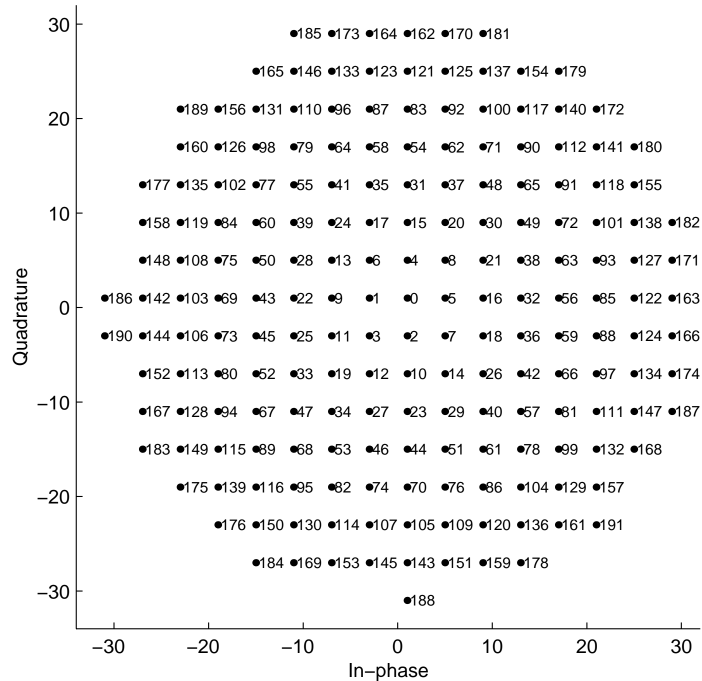
The table below lists the default values used in the simulation You can change a variable by assigning a new variable name in the MATLAB workspace, or by directly changing the variables in parameter fields in the dialog box. The variables with empty variable names in the table indicate that a scalar or vector is directly assigned in the dialog box in the Simulink block in this example.

Description	Variable Name	Value
Symbol rate, S	SRATE	3200
Bit rate, R	BRATE	28800
Convolution encoder	CON_CODE	1
Precode filter coefficients		$[1\ .8\ .4] + \sqrt{-1} \times [1\ .5\ .2]$
Convolution decoder delay steps		10
Number of bits per map frame, b	v34_rate_b	72
Number of data frames per superframe, J	v34_rate_j	7
Number of map frame per dataframe, P	v34_rate_p	16

The last three variables are calculated by using function v34prep. In this example, you can calculate the other parameters by assigning the three variables in the table and double-clicking the Update Data block. The Update Data block executes the v34prep function and generates a MATLAB figure. On the top of the MATLAB figure is a list of calculated parameters. At the bottom of the figure is a plot of the modulation constellation used in matching the data listed in the first part of the figure. The plot includes only one quarter of the constellation points. The complete constellation is the combination of the listed constellation and the 90, 180, and 270 degree rotations of the constellation.

Symbol Rate S:	3200
Bit Rate R:	28800
# of Dataframe per Superframe J:	7
# of Mapframe per Dateframe P:	16
# of bit per Dataframe N:	1152
# of bit per Mapframe b:	72
# of dividing bit in parser q:	4
# of shell mapping bit K:	28
# of rings in shell mapping M:	12
Extended M:	14
# of points in 2D constellation L:	768
Extended L:	896

One quarter of the points in the constellation:



You can use the function `v34const` to calculate or plot the constellation points with the given requirements.

The simulation is designed for the cases when all bits in SWP, the frame switching pattern, are all ones. Please check Table 8/V.34 for limitations. In the design, you can calculate the ratio of

```
Symbol_rate_S/Bit_rate_R
```

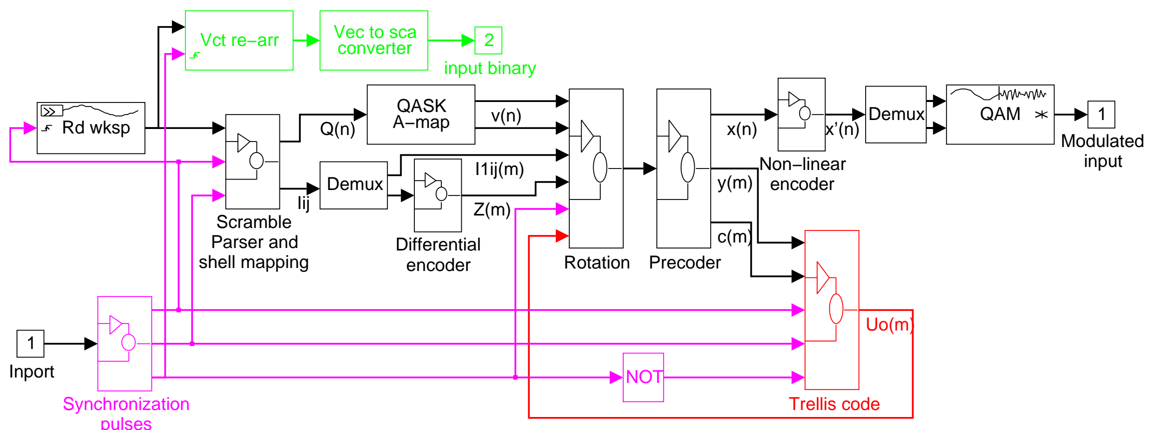
The blocks in this example work when the ratio between these two rates is an integer.

The simulation design is suitable when the number of map frame bits b is larger than 12.

When this example cannot simulate the data specified in SRATE and BRATE, the execution of a double Update Block will show a warning line in the figure. The error means the designed Call Modem and Answer Modem cannot simulate your given rate.

Call Modem Simulation Implementation

You can unmask the Call Modem block to see the details of the implementation of the Call Modem. The top level of the Call Modem is:

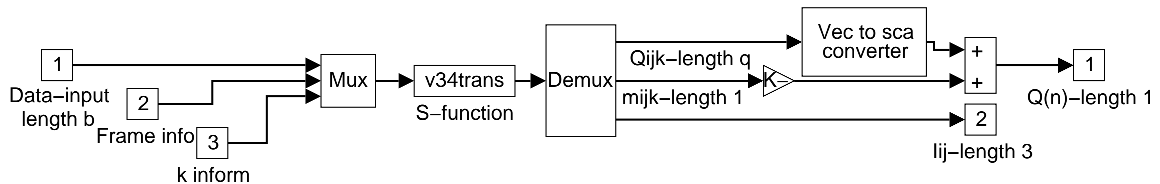


When the Call Modem is enabled to translate data, the enable signal enables the synchronization pulses subsystem to generate pulses to start the data

translation process. The data to be transferred in this example is a random binary integer, generated from the Read from Workspace block. The length `v34_rate_b` binary vector is divided into four sections. Each section takes a 4-D symbol interval to convert the `v34_rate_b/4` bit binary vector to an integer and outputs to the output port `2`.

Scramble, Parser, and Shell Mapping Subsystem

The scramble, parser, and shell mapping computation is accomplished in the scramble, parser and shell mapping subsystem. The figure below shows the block diagram of this subsystem:

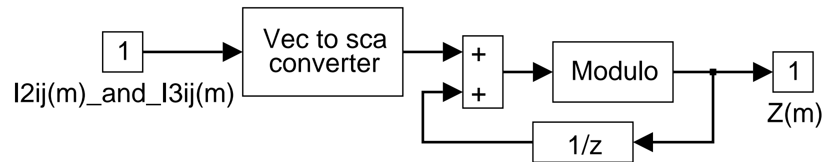


The length `v34_rate_b` binary vector is inputted into an S-function `v34trans`. The function `v34trans` is a C code S-function; read the file `v34trans.c` for the implementation details. The execution of the function `v34trans` is triggered by the raising edge of frame information signal. The block outputs the length `v34_rate_q` Q_{ijk} signal, length one m_{ijk} signal and length three I_{ij} signal at the raising edge of the `k` information signal. The signals Q_{ijk} and m_{ijk} generate signal `Q(n)`.

Differential Encoder Subsystem

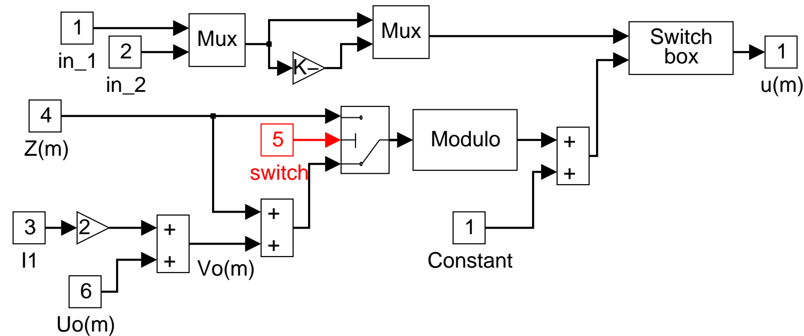
The differential encoder subsystem maps signal `Q(n)` into a two-dimensional signal `v(n)`, which includes in-phase and quadrature components. The constellation used in the mapping is provided in the constellation figure in the last section. The I_{ij} signal is divided into $I1_{ij}$ and $I2_{ij}$, $I3_{ij}$ signal. The $I2_{ij}$

and $I3_{ij}$ signal is used to generate Z(m) signal through different encode. This figure shows the block diagram of the differential encode calculation:



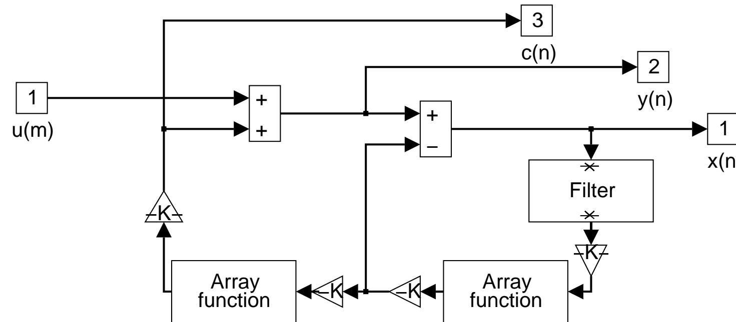
Rotation Subsystem

The in-phase and quadrature signal in v(m) is then rotated Z(m)*90 degrees when k=0, or $(Z(m)+2I1_{ij}+U_0(m))*90$ degrees when k=1. The implementation of the rotation is shown in the following block diagram, which produces signal u(m).



Precoder Subsystem

The precoder uses u(m) signal to generate signal c(n), y(n), and x(n). The implementation of the precoder is shown below, in which the filter is a complex filter. The complex filter takes a complex signal and complex filter coefficients. The filter outputs a filtered complex signal.



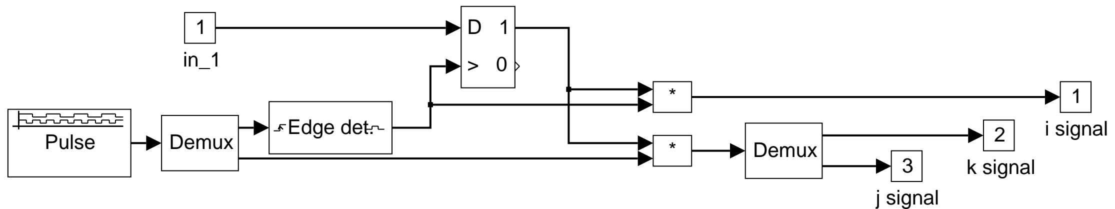
Nonlinear Encoder Subsystem

A nonlinear encoder takes the signal x(n) and outputs signal x'(n). In this example the nonlinear encode assumes s=0, so, x'(n) = x(n). The Call Modem outputs the QAM modulated x'(n) signal.

There are two subsystems used in the implementation. They are the synchronization pulses signal generator and the trellis code.

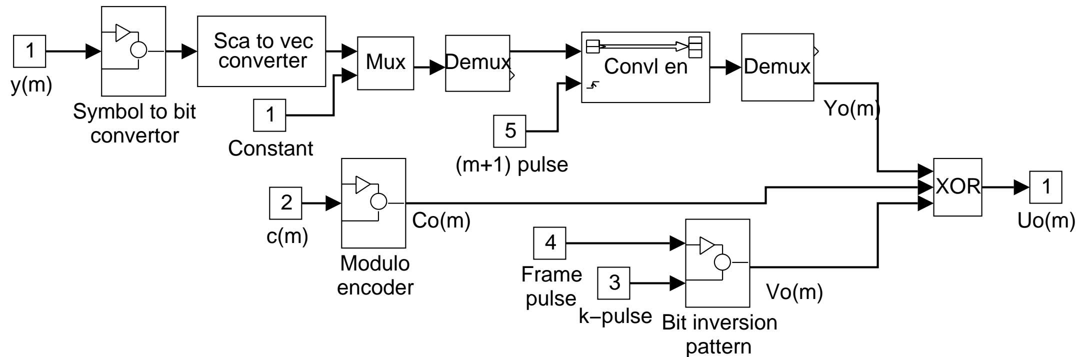
Synchronization Pulses Subsystem

The pulses are generated in the vector generator in the following block diagram. The generated pulses are output only if the enable signal is one signal. This subsystem produced frame signal i, 4-D frame signal j, and 2-D signal k. These signals are used to synchronize the simulation.



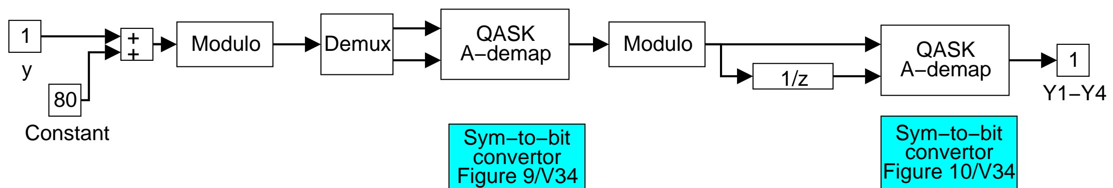
Trellis Code Subsystem

The most complicated part in the V.34 modem is the trellis modulation implementation. This figure shows the top level block diagram of the trellis coder:

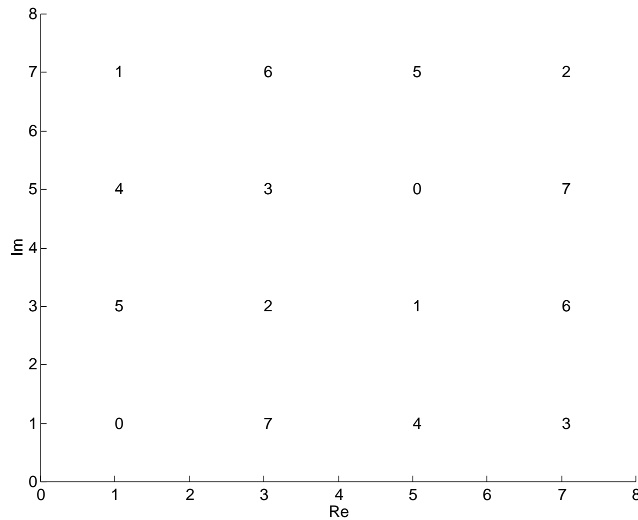


The $y(m)$ signal and $c(m)$ signal from the precoder are used to generate signal $Y_0(m)$ and signal $C_0(m)$ respectively. Based on the combination of j and k in the transmission, a $V_0(m)$ signal is produced as a fixed pattern in each frame. $U_0(m)$ signal, the trellis coder output, is the exclusive OR computation of the three signals, $Y_0(m)$, $C_0(m)$, and $V_0(m)$.

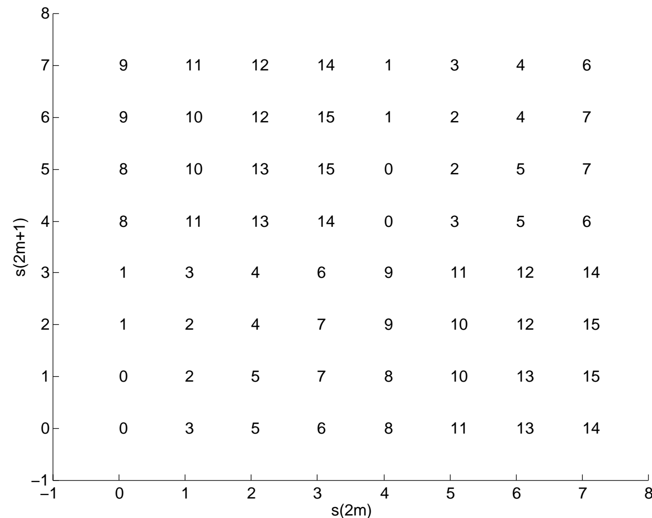
Symbol to Bit Conversion Subsystem. The symbol to bit conversion subsystem in the trellis coder subsystem maps the in-phase and quadrature signal in $y(2m)$ and $y(2m+1)$ into the four bits signal Y1, Y2, Y3, and Y4. This block diagram shows the implementation of the mapping:



The mapping takes place in two stages. The first stage is the mapping from the in-phase and quadrature signal y(n) into s(n). The in-phase and quadrature values are shifted to the value where both in-phase and quadrature are positive. The next plot shows the conversion map.



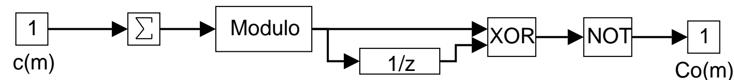
The second stage is the mapping from s(2m) and s(2m+1) to Y1, Y2, Y3, and Y4. This plot shows this stage of the mapping:



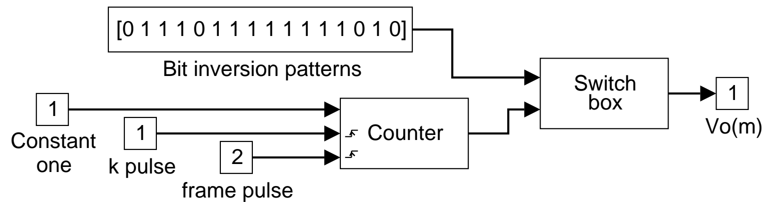
You can generate these two plots by double-clicking the cyan colored block in the symbol to bit convertor subsystem. Note that to ensure a correct plot, you have to close all other current opened figures.

The signals Y1, Y2, Y3, and Y4 n produce Y_0 signal, which is used in the convolution encoder.

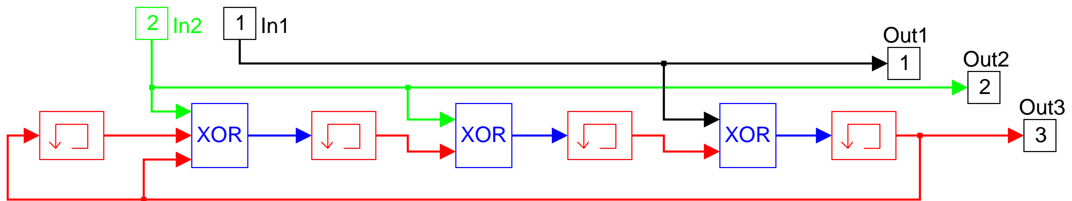
Modulo Encoder Subsystem. This is a diagram of the modulo encode subsystem:



The Bit Inversion Pattern Subsystem. The bit inversion pattern subsystem uses the frame signal and the k pulse signal to select the bit to be used in as $V_0(m)$. This figure shows the block diagram:

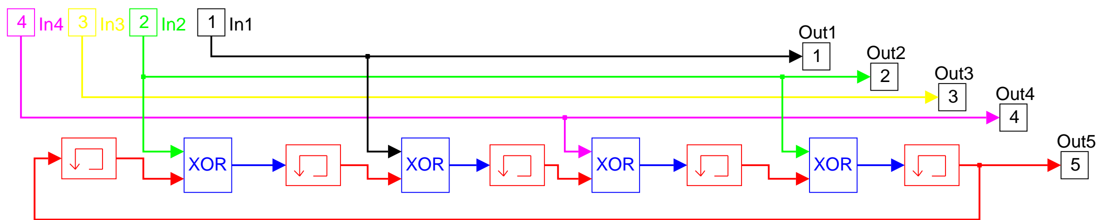


The Structure of Convolution Encoders. You can choose the convolution encoder in one of the three encoders, which are V.34 16 convolution encoders:



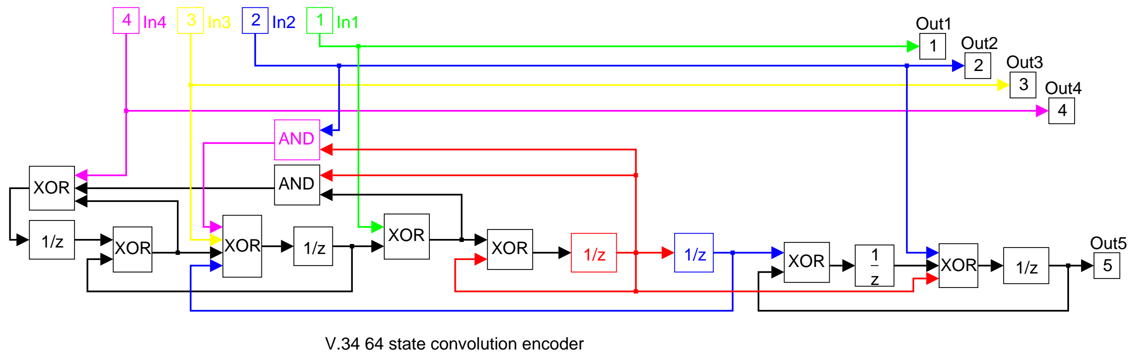
V.34 16 state convolution encoder

This is the V.34 32 state convolution encoder:



V.34 32 state convolution encoder

This is the V.34 64-state convolution encoder.

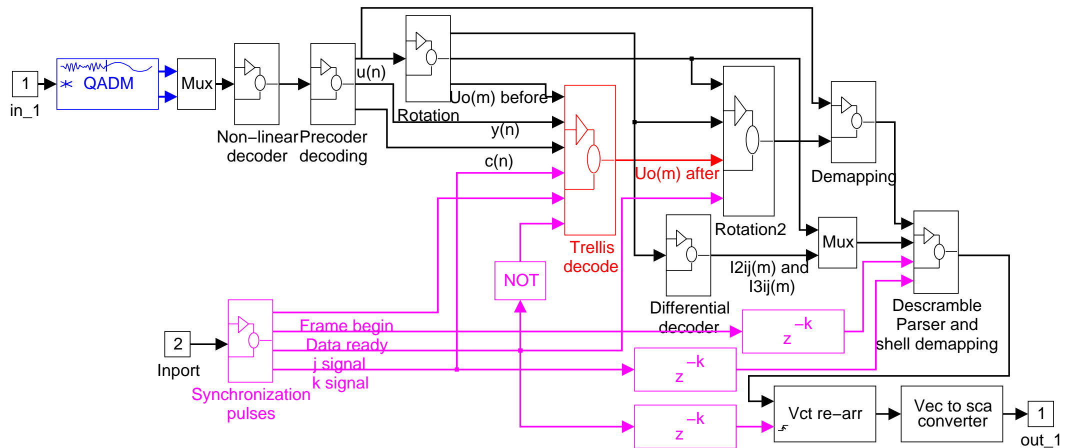


V.34 64 state convolution encoder

You can assign the `CON_CODE = 1, 2, or 3`, to choose the 16-state, 32-state and 64-state convolution encode. Note that the simulation will be very slow if you choose to use the 64-state encoder because of the high order states.

Answer Modem Simulation Implementation

The implementation of the Answer Modem is the inverse of the implementation of the Call Modem. You can unmask the Answer Modem to see the structure of the implemented Answer Modem. The top level of the Answer Modem is shown below. You can go through the details of the implementation by reading the block diagram and the parameter setting in the blocks.



The first input signal of the Answer Modem is the received modulated signal sent from the Call Modem. The second input signal to the Answer Modem is the enable signal to enable the synchronization pulses. The output of this block is the received data with the same format as the second output as the Call Modem.

You can use the output signal to compare the transmitted data. You can use a Triggered To Workspace block or Triggered To File block to save the data in comparing the transmitted data. You can connect the data input of the Triggered To Workspace block to the output port of the descramble, parser, and shell mapping subsystem. The trigger signal should be the signal “data ready” output from the synchronization pulses subsystem.

5

MATLAB Function Reference

Formats and Conventions 5-3

Reference Tables 5-4

The MATLAB function library in the Communications Toolbox is a collection of ready-to-use MATLAB commands for communication system analysis, design, and simulation. This chapter provides descriptions for all functions in this toolbox. If applicable, the functionality of the MATLAB function is the same as the functionality of the corresponding Simulink blocks. Because of different implementations, the numerical results of a MATLAB M-file function may not exactly match with the numerical results of the corresponding Simulink block.

The MATLAB functions are divided into these categories:

- Source, Sink, and Error Analysis
- Source Coding
- Error-Control Coding
- Error-Control Coding, Lower Level Functions
- Modulation and Demodulation
- Galois Field Computation Functions
- Special Filters
- Utilities

Note that there are fewer MATLAB functions than Simulink blocks in this toolbox. Some of the MATLAB functions combine several types of functions into one function. For example, the function `encode` supports linear block code, Hamming code, cyclic code, BCH code, Reed-Solomon code, and convolution code.

Formats and Conventions

This chapter uses a different format for the function descriptions. Note that not all of theses headings are used in each function:

Purpose	Provides short concise descriptions.
Synopsis	Shows format of the command or function.
Description	Describes what the command/function does and any rules/restrictions that apply.
Examples	Proves examples of how the command/function can be used.
Algorithm	Associated algorithms and routines.
See Also	Refers you to other related commands/functions.
Reference	Additional resources.

and the following conventions:

`Monospace`	Commands, functions names, and screen displays; for example, `conv`.
Italics	Book titles, names of sections, mathematical notation, and for introduction of new terms; for example, *Communications Fundamentals*.
Bold Initial Caps	Key names, menu names, and items that are selected from menus; for example, the **Enter** key.

Reference Tables

The Communications Toolbox provides eight main categories of functions. The MATLAB command `help comcmds` displays an online table of the main categories and their functions.

Source, Sink, and Error Analysis	
biterr	Bit error number and bit error rate computation.
eyescat	Eye-pattern diagram or scatter plot.
randbit	Place ones randomly in zero vectors.
randint	Uniform distribution random integer generator.
symerr	Symbol error number and symbol error rate computation.

Source Coding	
compand	μ-law or A-law compressor and expander calculation.
dpcmdeco	Differential pulse code modulation decode computation.
dpcmenco	Differential pulse code modulation encode computation.
dpcmopt	Differential pulse code modulation parameter estimation using training data set.
lloyds	Scalar quantization optimization using training data and using Lloyd algorithm.
quantiz	Produce quantization index and quantized output value.

Error-Control Coding	
bchgen	Produce generator matrix and parity-check matrix for BCH code.
bchpoly	Produce generator polynomial for BCH code.
cyclgen	Produce generator matrix and parity-check matrix for cyclic code.
cyclpoly	Produce cyclic generator polynomial for cyclic code.

Error-Control Coding	
decode	Decoding computation for a number of error-control coding techniques, which include linear block coding, Hamming coding, cyclic coding, BCH coding, Reed-Solomon coding, and convolution coding.
encode	Encoding computation for a number of error-control coding techniques, which include linear block coding, Hamming coding, cyclic coding, BCH coding, Reed-Solomon coding, and convolution coding.
gen2abcd	Convert a convolution code transfer function to [A, B, C, D] form of transfer function.
gen2par	Convert between generator matrix and parity-check matrix.
hammgen	Produce generator matrix and parity-check matrix for Hamming code.
htruthtb	Generate truth table for Hamming decode.
oct2gen	Convert convolution code transfer function between an octal form and a binary form.
rsdecof	Use Reed-Solomon code to decode an encoded text file.
rsencof	Use Reed-Solomon code to encode a text file.
rspoly	Produce Reed-Solomon code generator polynomial.
sim2gen	Generate a convolutional code transfer function from a Simulink block diagram.
sim2logi	Generate a nonlinear convolution code transfer function from a Simulink block diagram.
sim2trans	Generate a convolution code transfer function from a Simulink block diagram. This function applies to both linear and nonlinear systems.

Error-Control Coding, Lower Level Functions	
bchcore	BCH decode kernel processing function.
bchdeco	BCH decode computation.

Error-Control Coding, Lower Level Functions	
bchenco	BCH encode computation.
bchpoly	Produce BCH code generator polynomial.
convenco	Convolution code encoding calculation.
errlocp	Calculate error-location polynomial for BCH and Reed-Solomon code.
rscore	Reed-Solomon decode core computation part.
rsdeco	A user-interfaced Reed-Solomon decode calculation.
rsdecode	Reed-Solomon code decode calculation.
rsenco	A user-interfaced Reed-Solomon encode calculation.
rsencode	Reed-Solomon code encode calculation.
viterbi	Viterbi convolution decode.

Modulation and Demodulation	
ademod	Analog demodulation, passband simulation. This function is the inverse process of function amod.
ademodce	Analog demodulation, baseband simulation. This function is the reverse of function amodce.
amod	Analog modulation, passband simulation. The modulation methods can be chosen from AM-DSB-SC, AM-SSB, AM-TC, QAM, FM, and PM.
amodce	Analog modulation, baseband simulation. The modulation methods can be chosen from AM-DSB-SC, AM-SSB, AM-TC, QAM, FM, and PM.
apkconst	Compute/plot QASK circle constellation.
ddemod	Digital demodulation, passband simulation. This function is the reverse of function dmod.
ddemodce	Digital demodulation, baseband simulation. This function is the reverse of function dmod.
demodmap	Digital demodulation demapping. This function is the reverse of function modmap.

Modulation and Demodulation	
dmod	Digital modulation, passband simulation. The modulation methods can be chosen from ASK, QASK, FSK, MSK, and PSK.
dmodce	Baseband digital modulation. The modulation methods can be chosen from ASK, QASK, FSK, MSK, and PSK.
modmap	Digital modulation mapping for ASK, QASK, FSK, MSK, and PSK.
qaskdeco	Decode square constellation QASK code. This function is reverse computation of qaskenco.
qaskenco	Compute/plot QASK square constellation.

Galois Field Computation Functions	
flxor	Floating point number exclusive OR calculation.
gfadd	Add two GF(2) polynomials or add two GF(2^m) elements.
gfdiv	Compute the quotient and remainder of GF(2) polynomial division.
gfconv	Convolution calculation for two GF(2) polynomials.
gfcoset	GF(2^m) cosets calculation.
gfdeconv	GF(2) polynomials deconvolution calculation.
gffilter	GF(2) filtering computation.
gflineq	Compute x in $A\ x = b$ in GF(2).
gfminpol	Find minimum polynomials in GF(2^m) .
gfmul	Polynomial multiplication in GF(2) or element multiplication in GF(2^m).
gfplus	GF(2^m) additive computation.
gfpretty	GF(2) polynomial presentation.
gfprimck	Test GF(2) polynomial irreducible and primitive properties.
gfprimdf	Output default GF(2) primitive polynomial at a given degree.

Galois Field Computation Functions	
gfprimfd	Find GF(2) primitive polynomials.
gfrank	Matrix rank in GF(q).
gfrepcov	Galois field polynomial representation conversion.
gfroots	Find roots of a polynomial in extended Galois field.
gfsub	Subtract a GF(2^m) polynomial from another.
gftrunc	GF(2) polynomial truncation process.
gftuple	GF(2^m) m-tuple representation.
isprime	Check whether a number is a prime number.
primes	Generate prime numbers.

Special Filters	
hilbana	Design an analog Hilbert transform filter.
hilbatf	Design an analog Hilbert transform filter with compensator for the known input signal frequency.
hilbiir	Design a digital Hilbert IIR filter.
hank2sys	Convert Hankel matrix to linear system.
imp2sys	FIR filter to IIR filter conversion.
rcosfir	Raised cosine FIR filter design (lower level function).
rcosflt	Filtering signal using raised cosine filter.
rcosiir	Raised cosine IIR filter design (lower level function).
rcosine	Design a raised cosine filter.

Utilities	
bi2de	Binary to positive decimal integer conversion.
blkdiag	Matrix block diagonal matrix.
callhelp	Display help text sections.

Utilities	
checkinp	Check the function input and set it to a default value if the input is empty or missing.
de2bi	Positive decimal integer to binary conversion.
vec2mat	Convert vector to matrix by a given column number.

ademod

Purpose

Analog demodulation, passband simulation

Syntax

```
ademod(method);
y = ademod(rec_sig, Fc, Fs, method, num, den);
```

Description

Function `ademod` demodulates an analog modulated signal in passband. This function is the inverse process of the `amod` function. The string variable method specifies the types of demodulation used. The table below lists the types of analog demodulation available:

Method	Demodulation Scheme
'amdsb-sc'	Amplitude demodulation, double sideband suppressed carrier
'amdsb-tc'	Amplitude demodulation, double sideband with transmission carrier
'amssb'	Amplitude demodulation, single sideband suppressed carrier
'qam'	Quadrature amplitude demodulation
'fm'	Frequency demodulation
'pm'	Phase demodulation

`ademod(`*`method`*`)` provides more detailed help for using this function with a specific demodulation method.

`y = ademod(rec_sig, Fc, Fs, `*`method`*`, num, den)` demodulates the received signal `rec_sig`. The carrier frequency is `Fc` (Hertz), and the sample frequency is `Fs` (Hertz). The time interval between two consecutive points of `y` is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default value is zero. The sample frequency must match the parameters used in the modulation, but the initial phase may differ.

This function uses a lowpass filter in the demodulation. The numerator and the denominator of the lowpass filter are specified by the input parameters `num` and

den respectively. The sample time of the lowpass filter equals 1/Fs. When num is zero, empty or not given, this function uses a default Butterworth lowpass filter generated by:

```
[num,den] = butter(5,Fc*2/Fs);
```

method = 'amdsb-sc'

y = ademod(rec_sig,Fc,Fs,'amdsb-sc',num,den) demodulates the input signal rec_sig using the double sideband suppressed carrier method.

y = ademod(rec_sig,Fc,Fs,'amdsb-sc/costas',num,den) uses a Costas loop in the demodulation computation.

method = 'amdsb-tc'

y = ademod(y,Fc,Fs,'amdsb-tc',num,den,offset) demodulates the input signal rec_sig using the double sideband with transmission carrier method. Note that this method has an optional parameter offset, which is the value subtracted from rec_sig after the demodulation.When offset is omitted, the function sets offset = mean(rec_sig).

method = 'amssb'

y = ademod(rec_sig,Fc,Fs,'amssb',num,den) demodulates the input signal rec_sig using the single sideband suppressed carrier method.

method = 'qam'

y = ademod(rec_sig,Fc,Fs,'qam',num,den) demodulates the input signal rec_sig using the quadrature amplitude demodulation method. The input signal rec_sig must be a vector. The output signal y is a two-column matrix. The first column is the in-phase signal and the second column the quadrature signal.

method = 'fm'

y = ademod(rec_sig,Fc,Fs,'fm',num,den) demodulates the input signal rec_sig using the quadrature frequency demodulation method.

method = 'pm'

y = ademod(rec_sig,Fc,Fs,'pm',num,den) demodulates the input signal rec_sig using the quadrature phase demodulation method.

See Also

amod, dmod, ddemod, amodce, ademodce

Purpose

Analog demodulation, baseband simulation with complex envelope

Syntax

```
ademodce(method);
y = ademodce(rec_sig, Fs, method, num, den);
```

Description

Function `ademodce` performs analog demodulation in baseband. Use this function to demodulate the `amodce` modulated signal. The string variable method specifies the type of demodulation used. The table below lists the types of analog demodulation available:

Method	Demodulation Scheme
'amdsb-sc'	Amplitude demodulation, double sideband suppressed carrier
'amdsb-tc'	Amplitude demodulation, double sideband with transmission carrier
'amssb'	Amplitude demodulation, single sideband suppressed carrier
'qam'	Quadrature amplitude demodulation
'fm'	Frequency demodulation
'pm'	Phase demodulation

`ademodce(method)` provides more detailed help for using this function with a specific demodulation method.

`y = ademodce(rec_sig, Fs, method, num, den)` demodulates the received signal `rec_sig`. The simulation sample frequency is `Fs` (Hertz), and the time interval between two consecutive points in y is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default value is zero. The sample frequency must match the one used in the modulation, but the initial phase may differ.

This function uses a lowpass filter in the demodulation. The numerator and the denominator of the lowpass filter are specified by the input parameters `num` and `den` respectively. The sample time of the lowpass filter equals `1/Fs`. When `num`

is zero, empty or not given, this function uses a default lowpass filter, which is generated by using the command:

```
[num,den] = butter(5,Fc*2/Fs);
```

method = 'amdsb-sc'

`y = ademodce(rec_sig,Fs,'amdsb-sc',num,den)` demodulates the input signal `y` using double sideband suppressed carrier method.

`y = ademodce(rec_sig,Fs,'amdsb-sc/costas',num,den)` uses Costas loop in the demodulation computation.

method = 'amdsb-tc'

`y = ademodce(rec_sig,Fs,'amdsb-tc',num,den,offset)` demodulates the input signal `rec_sig` using the double sideband with transmission carrier method. Note that this method has an optional parameter `offset`, which is the value subtracted from `y` after the demodulation. When `offset` is omitted, the function sets `offset = mean(y)`.

method ='amssb'

`y = ademodce(rec_sig,Fs,'amssb',num,den)` demodulates the input signal `rec_sig` using the single sideband suppressed carrier method.

method = 'qam'

`y = ademodce(rec_sig,Fs,'qam',num,den)` demodulates the input signal `rec_sig` using the quadrature amplitude modulation method. The input signal `y` is a vector. The output signal `rec_sig` is a two-column matrix. The first column is the in-phase signal and the second column the quadrature signal.

method = 'fm'

`rec_sig = ademodce(rec_sig,Fs,'fm',num,den)` demodulates the input signal `rec_sig` using the quadrature frequency modulation method.

method = 'pm'

`rec_sig = ademodce(rec_sig,Fs,'pm',num,den)` demodulates the input signal `rec_sig` using the quadrature phase modulation method.

See Also

amod, dmod, ddemod, amodce, ademod

amod

Purpose

Analog modulation, passband simulation

Syntax

```
amod(method);
y = amod(x,Fc,Fs,method,...);
[y,t]= amod(x,Fc,Fs,method,...);
```

Description

Function `amod` performs analog modulation in passband. This function modulates the input signal. The string variable *method* specifies the type of modulation used. The table below lists the types of analog modulation available:

Method	Modulation Scheme
'amdsb-sc'	Amplitude modulation, double sideband suppressed carrier
'amdsb-tc'	Amplitude modulation, double sideband with transmission carrier
'amssb'	Amplitude modulation, single sideband suppressed carrier
'qam'	Quadrature amplitude modulation
'fm'	Frequency modulation
'pm'	Phase modulation

`amod(method)` provides more detailed help for using this function with a specific modulation method.

`y = amod(x,Fc,Fs,method,...)` modulates the message signal `x` with carrier frequency `Fc` (Hertz) and sample frequency `Fs` (Hertz). The time interval between two consecutive points in `x` is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default value is zero. By the Nyquist theorem, the sample frequency must be at least twice as large as the modulation carrier frequency.

`[y,t] = amod(x,Fc,Fs,method,...)` outputs the computation time in `t`.

method = 'amdsb-sc'

y = amod(x, Fc, Fs, 'amdsb-sc') modulates the input signal x using the double sideband suppressed carrier method.

method = 'amdsb-tc'

y = amod(x, Fc, Fs, 'amdsb-tc', offset) modulates the input signal x using the double sideband with transmission carrier method. offset is the value subtracted from x prior to the modulation. When offset is omitted, the function sets offset = abs(min(x)).

method = 'amssb'

y = amod(x, Fc, Fs, 'amssb') modulates the input signal x using the single sideband suppressed carrier method. A frequency domain Hilbert transform method is used in the computation.

y = amod(x, Fc, Fs, 'amssb', Hilbert_Filter) uses a time domain Hilbert filter for the single sideband modulation calculation. This function checks whether there is a marker in the Hilbert_filter position of the parameter list, so the contents of the variable Hilbert_Filter can be anything. The function uses a default Hilbert filter, which it generates using [num, den] = hilbiir(1/Fs).

y = amod(x, Fc, Fs, 'amssb', num, den) uses a time domain Hilbert filter for the single sideband modulation calculation. The numerator and the denominator of the Hilbert filter are specified by the input parameter num and den respectively. You can use function hilbiir to design the Hilbert filter.

method = 'qam'

y = amod(x, Fc, Fs, 'qam') modulates the input signal x using the quadrature amplitude modulation method. The input signal x is a two-column matrix with the first column as the in-phase signal and the second column as the quadrature single. The output y is a one column modulated signal.

method = 'fm'

y = amod(x, Fc, Fs, 'fm') modulates the input signal x using the quadrature frequency modulation method. The modulated signal takes the spectrum from min(x) + Fc to max(x) + Fc.

y = amod(x, Fc, Fs, 'fm', sensitivity) specifies the input signal sensitivity in the modulation, which is equivalent to using the command

```
y = amod(x*sensitivity, Fc, Fs, 'fm');
```

method = 'pm'

y = amod(x,Fc,Fs,'pm') modulates the input signal x using the quadrature phase modulation method.

y = amod(x,Fc,Fs,'pm',sensitivity) specifies the input signal sensitivity in the modulation, which is equivalent to using the command:

```
y = amod(x*sensitivity,Fc,Fs,'pm');
```

See Also

ademod, dmod, ddemod, amodce, ademodce

Purpose Analog modulation, baseband simulation with complex envelope

Syntax

```
amodce(method);
y = amodce(x,Fs,method,...);
```

Description Function `amodce` is designed for analog modulation in baseband simulation. This function modulates the input signal. The string variable *method* specifies which type of modulation used. The table below lists the types of analog modulation available:

Method	Modulation Scheme
'amdsb-sc'	Amplitude modulation, double sideband suppressed carrier
'amdsb-tc'	Amplitude modulation, double sideband with transmission carrier
'amssb'	Amplitude modulation, single sideband suppressed carrier
'qam'	Quadrature amplitude modulation
'fm'	Frequency modulation
'pm'	Phase modulation

`amodce(`*method*`)` provides more detailed help for using this function with a specific modulation method.

`y = amodce(x,Fs,`*method*`,...)` outputs the complex envelop of the modulation of the input signal `x`. The input and the output signals have the same sample frequency `Fs` (Hertz). The output `y` is a complex matrix. This function modulates the message signal `x` with carrier frequency `Fc` (Hertz) and sample frequency `Fs` (Hertz). The time interval between two consecutive points in `x` is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default is zero.

method = 'amdsb-sc'

`y = amodce(x,Fs,'amdsb-sc')` modulates the input signal `x` using the double sideband suppressed carrier method.

method = 'amdsb-tc'

`y = amodce(x,Fs,'amdsb-tc',offset)` modulates the input signal `x` using the double sideband with transmission carrier method. `offset` is the value subtracted from `x` prior to the modulation. When `offset` is omitted, the function sets the default `offset = abs(min(x))`.

method = 'amssb'

`y = amodce(x,Fs,'amssb')` modulates the input signal `x` using the single sideband suppressed carrier method. A frequency domain Hilbert transform method is used in the computation.

`y = amodce(x,Fs,'amssb/time')` uses a time domain Hilbert filter for the single sideband modulation calculation. The function uses a default Hilbert filter, which it generates using `[num,den] = hilbiir(1/Fs)`.

`y = amodce(x,Fs,'amssb/time',num,den)` uses a time domain Hilbert filter for the SSB modulation calculation. The numerator and the denominator of the Hilbert filter are specified by the input parameters `num` and `den` respectively. You can use function `hilbiir` to design the Hilbert filter.

method = 'qam'

`y = amodce(x,Fs,'qam')` modulates the input signal `x` using the quadrature amplitude modulation method. The input signal `x` is a matrix with an even number of columns. The odd columns are the in-phase signal and the even columns are the quadrature single. The output `y` has its number of column equal to one half of the column number of `x`.

method = 'fm'

`y = amodce(x,Fs,'fm')` modulates the input signal `x` using quadrature frequency modulation method. The modulated signal takes the spectrum from `min(x) + Fc` to `max(x) + Fc`.

method = 'pm'

`y = amodce(x,Fs,'pm')` modulates the input signal `x` using quadrature phase modulation method.

See Also

ademodce, dmod, ddemod, amod, ademod

Purpose

Compute and plot a QASK circle constellation

Syntax

```
apkconst(noc);
apkconst(noc, roc);
apkconst(noc, roc, poc);
y = apkconst(...);
```

Description

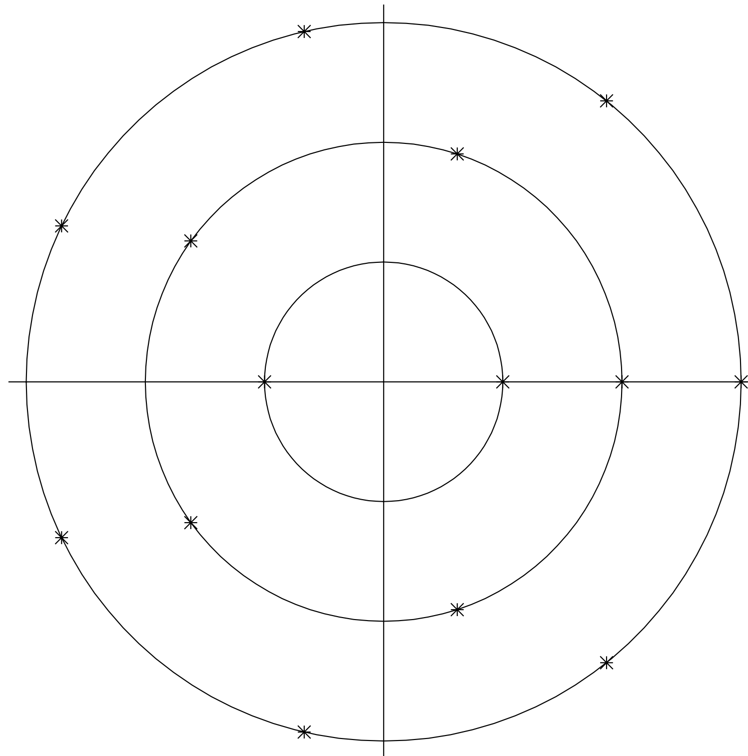
`apkconst(noc)` plots a QASK circle constellation. The parameter `noc` is either a scalar or a vector; its elements must be positive integers. If `noc` is a scalar, then the function plots a single circle with the number of points specified by `noc`. If `noc` is a vector, the function plots a circle for each element of the vector. The default radius of the first circle is 1, and each following circle increases one unit in radius. The constellation in each circle is evenly distributed with the phase of one of the constellation points equal to 0.

`apkconst(noc, roc)` specifies the radii in each circle. The vector length for `roc` must be the same as the vector length for `noc`. The elements in `roc` must be positive numbers.

`apkconst(noc, roc, poc)` specifies the signature phase in each circle. The signature phase specifies the phase for one of the constellation points. The vector length for `poc` must be the same as the vector length for `noc` and `roc`.

`y = apkconst(...)` outputs the constellation into a complex vector `y`. The real part of `y` is the in-phase component and the imaginary part of `y` is the quadrature component.

Example Setting `noc = [2 4 6]`, `apkconst(noc)` plots:



ASK/PSK Constellation

See Also dmod, modmap, ddemod, demodmap

Purpose A low-level function that is the core part of the BCH decode algorithm

Syntax

```
[msg, err, ccode] = bchcore(code, pow_dim, dim, k, t, tp);
```

Description `[msg, err, ccode] = bchcore(code, pow_dim, dim, k, t, tp)` decodes a BCH codeword vector. `code` is a codeword vector with its length equal to 2^{pow_dim-1}. `pow_dim` is a positive integer that specifies the size of the Galois field used in the decoding computation. `k` is the number of information bits; `t` is the BCH error correction capability; and `tp` is a complete list of the elements in GF(2^{dim}) in ascending order. This function is called by Simulink blocks and MATLAB functions.

Note: In order to reduce computational overhead, this function does not perform error-checking. Use this function with caution.

See Also `bchdeco`, `decode`, `bchpoly`

bchdeco

Purpose BCH error-control decoding

Syntax

```
msg = bchdeco(code, k, t);
msg = bchdeco(code, k, t, ord);
[msg, err] = bchdeco(...);
[msg, err, ccode] = bchdeco(...);
```

Description

`msg = bchdeco(code, k, t)` recovers a message signal `msg` from a codeword `code` using BCH decoding. `k` is the number of information bits; `t` is the error-correction capability.

`msg = bchdeco(code, k, t, ord)` specifies the order `ord` of the primitive polynomial required for the BCH code.

`[msg, err] = bchdeco(...)` outputs the received error. When the number of errors is larger than `t`, the error correction capability, `err` is a negative number.

`[msg, err, ccode] = bchdeco(...)` outputs the corrected codeword.

Example

The code below is an example of using `bchdeco` to process a three error correction BCH code:

```
[pg, pm, cs, h, t] = bchpoly(15, 5);
[h, gen] = cyclgen(15, pg);
msg = randint(100, 5);
code = encode(msg, gen);
code = rem(code + randbit(100, 15, [.3, .3, .3]), 2);
[dec, err] = bchdeco(code, 5, t);
```

See Also bchcore, bchenco, decode

Purpose

BCH error-control encoding

Syntax

```
code = bchenco(msg, n, k);
code = bchenco(msg, n, k, pg);
```

Description

`code = bchenco(msg, n, k)` encodes the `k`-column message `msg` to an `n`-column codeword `code` by using the BCH coding. `msg` is a `k`-column message; `code` is an `n`-column codeword.

`code = bchenco(msg, n, k, pg)` specifies the generator polynomial `pg` for BCH code.

Examples

See the example provided in `bchdeco`.

See Also

`bchdeco`, `encode`, `decode`

bchpoly

Purpose Produce a BCH code generator polynomial

Syntax

```
bchpoly
pg = bchpoly(N);
pg = bchpoly(m,t);
[pg,n,k] = bchpoly(m,t);
[pg,n,k,pm] = bchpoly(m,t);
```

Description

`bchpoly` produces a list of the codeword length, message length, and error-correction capability for BCH code. The codeword lengths listed are 7, 15, 31, 63, 127, 255, and 511.

`pg = bchpoly(N)` outputs an array `pg` that contains code word lengths equal or larger than N, the associated message lengths, and error-correction capability. The codeword lengths, message lengths and error-correction capabilities are in the first, second, and third columns of Pg respectively. When N > 511, the error control capability is not listed.

`pg = bchpoly(m,t)` produces a BCH code generator polynomial `pg` based on two positive integers `m` and `t`. `m` is a positive integer larger than 3 that determines the length of the codeword `n = 2`$^{m-1}$. `t` determines the error correction capability. This function uses the default primitive polynomial generated by `gfprimdf(m)`.

`m` can be a binary vector instead of a positive integer. In this case, it represents a primitive polynomial generated in the field GF(2^{deg}), where `deg` is the degree of the primitive polynomial.

`[pg,n,k] = bchpoly(m,t)` outputs the codeword length `n` and the number of information bits `k`.

`[pg,n,k,pm] = bchpoly(m,t)` outputs `the` minimum polynomial matrix `pm` in the field GF(2^m). Each row is a minimum polynomial; `pm` is a `t`-by-`(m+1)` matrix.

See Also `cyclpoly`, `encode`, `decode`

Purpose Binary vector to positive decimal integer conversion

Syntax

```
de = bi2de(bi);
```

Description

`de = bi2de(bi)` converts a binary vector `bi` to positive decimal integer. When `bi` is a matrix, the output of `bi2de` is a column vector with each element equal to the decimal representation of the corresponding row of `bi`.

Examples

```
di = randint(5,5);    % Generate a 5-by-5 random binary matrix.
de = bi2de(di);
```

The last line outputs a 5-by-1 vector with each element converted from a row of the matrix `di`.

See Also `de2bi`

biterr

Purpose Number of bit errors and bit error rate computation

Syntax

```
[number,ratio] = biterr(x,y);
[number,ratio] = biterr(x,y,K);
[number,ratio] = biterr(x,y,K,'columnwise');
```

Description

`[number,ratio] = biterr(x,y)` compares the elements in two matrices `x` and `y`. The elements of `x` and `y` must be either binary numbers or integers. The number of bits that differ is output in `number`, and the ratio of `number` to the total number of bits contained in `y` is output in `ratio`.

All elements in `x` and `y` should be non-negative integers. This function calculates the total bit number in `y` by `column_y*row_y*K`, where `K` is the number of bits per element in `y`. If `K` is not specified, `it` defaults to the smallest integer satisfying $MAX < 2^K$, where `MAX` is the maximum number in both `x` and `y`.

When `x` is a matrix and `y` is a vector, or vice versa, this function converts the vector into a matrix with the same number of columns as the matrix. The function fills each column of the new matrix with whatever value was in the corresponding index of the vector. For example:

$$\begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

`[number,ratio] = biterr(x,y,K)` specifies the bit number `K` in ratio calculation.

`[number,ratio] = biterr(x,y,K,'columnwise')` specifies a columnwise comparison.

Examples

The example below adds error to 10% of the elements in a matrix. Each element in the matrix is a two-bit number. This code computes the bit-error ratio:

```
rec_sig = randint(100,100,4);
[indx,indy] = find(rand(100,100) > .9);
y = randint(length(indx),1,4);
z = zeros(100,100);
for i = 1 : length(indx)
z(indx(i),indy(i)) = y(i);
end
y = rem(x + z,4);
[num_sym,rat_sym] = symerr(x,y)
num_sym = 720
rat_sym = .072
[num_bit,rat_bit] = biterr(x,y,2);
num_bit = 950
rat_bit = .047
```

The symbol error is 7.2% instead of 10% because of the nature of the function `randint`. There is a 25% chance of producing a zero element, which is not an error.

See Also

`symerr`

blkdiag

Purpose Construct a block diagonal matrix by the order of input matrices

Syntax

```
out = blkdiag(a,b,c,d,...);
```

Description `out = blkdiag(a,b,c,d,...)`, where a, b, ... are matrices, outputs a block-diagonal matrix, such that the output matrix `out` is

$$\textbf{out} = \begin{bmatrix} a & 0 & 0 & 0 & 0 \\ 0 & b & 0 & 0 & 0 \\ 0 & 0 & c & 0 & 0 \\ 0 & 0 & 0 & d & 0 \\ 0 & 0 & 0 & 0 & \ldots \end{bmatrix}$$

The number of input matrices is limited to 26. The input matrices do not need to be square, nor do they need to be of equal size.

See Also `diag`

Purpose Call a help file and display a section of the file contents in the MATLAB workspace window

Syntax

```
callhelp(file_name);
callhelp(file_name, str);
callhelp(file_name, str, addition_message);
err = callhelp(...);
```

Description

`callhelp(file_name)` calls the file specified in the string variable *`file_name`* and displays all contents of the file in the MATLAB workspace. The string in *`file_name`* must specify the full name of the file, including the file extension.

`callhelp(file_name, str)` searches the contents in *`file_name`*, prints the section between `[str, '_help_begin']` and `[str '_help_end']` and displays it in the MATLAB workspace, where *`str`* is any ASCII string.

`callhelp(file_name, str, addition_message)` appends the message in the string variable *`addition_message`* to the display.

`err = callhelp(...)` outputs the function execution error message. If the command executes successfully, `err=0`. If `[str, '_help_begin']` or `[str, 'help_end']` is not found, `err=-2`. If the file is not found, the function abandons the execution and displays an error message.

See Also `disp`

checkinp

Purpose Check input variables and fill in the default values

Syntax

```
[y1,y2,...,yN] = checkinp(x1,x2,...,XN,z1,z2,...,zN);
```

Description This function checks if the entry `xi` is empty. If `xi` is empty, this function fills the output `yi` with `zi`. Otherwise, the output `yi` has the value in `xi`.

The number of outputs `N` is limited to less than or equal to 10. This utility function is useful for checking input values and filling them with default values if any of the inputs is an empty entry.

Purpose

μ-law or A-law compressor and expander calculation in source coding

Syntax

```
out = compand(in, param, V);
out = compand(in, param, method);
```

Description

The word compander is a combination of the terms compressor and expander. Both μ-law companders and A-law companders use logarithm compression and expansion in their respective algorithms. The μ-law compressor is defined as

$$y = \frac{V\log(1+\mu|x|/V)}{\log(1+\mu)}\operatorname{sgn}(x)$$

and the A-law compressor is defined as

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\operatorname{sgn}(x) & \text{for } 0 \le |x| \le A/V \\ \dfrac{V(1+\log A|x|/V)}{1+\log A}\operatorname{sgn}(x) & \text{for } A/V < |x| \le V \end{cases}$$

where μ is the μ-law parameter and `A` is the A-law parameter. `V` is the signal peak value magnitude.

`out = compand(in, param, V)` implements a μ-law compressor with `m` specified in `param` and the peak magnitude in `V`.

`out=compand(in, param, V, method)` implements either μ- or A-law compander computation. `param` specifies the `m` or `A` value, and `V` the input signal peak magnitude. The parameter *`method`* specifies the computation. The table below list the available options for the *`method`* string:

Method	Meaning
`'mu/compressor'`	μ-law compressor
`'mu/expander'`	μ-law expander
`'A/compressor'`	A-law compressor
`'A/expander'`	A-law expander

See Also

`quantiz`, `lloyds`, `dpcmopt`, `dpcmenco`, `dpcmdeco`

Purpose Convolutional error-control encoding

Syntax

```
code = convenco(msg, tran_func);
```

Description `code = convenco(msg, tran_func)` encodes the `k`-column message `msg` to `n`-column codeword `code` by the convolutional code method. The transfer function matrix is provided in linear or nonlinear format `tran_func`, which you can generate by using `sim2tran`, `sim2gen`, or `sim2logi`. The column number of the input `msg` should be `k`. The column number of the output code is `n`. The row number of code is `row_number_of_msg+m`, where `m` is the memory length.

See Also `encode`, `decode`, `sim2gen`, `viterbi`

Purpose Produce a cyclic code generator matrix and a parity-check matrix

Syntax

```
h = cyclgen(n,p);
h = cyclgen(n,p,opt);
[h,g] = cyclgen(...);
[h,g,m,k] = cyclgen(...);
```

Description `h = cyclgen(n,p)` produces the parity-check matrix for a given length `n` codeword and generates a cyclic polynomial `p`. Polynomial `p` can generate a cyclic code only if `p` is a factor of the polynomial X^{n-1}. The number of information bits is
`k = n - m`, where `m` is the degree of polynomial `p`. The parity-check matrix is an `m`-by-`n` matrix.

`h = cyclgen(n,p,opt)` produces the parity-check matrix based on the instruction given in `opt`. When `opt = 'system'`, the function generates a systematic matrix for cyclic code.

`[h,g] = cyclgen(...)` produces the generator matrix `g` and the parity-check matrix `h`. The parity-check matrix is a `k`-by-`n` matrix, where `k = n - m`.

`[h,g,m,k] = cyclgen(...)` outputs the dimension information `n` and `k`.

See Also `cyclpoly`

cyclpoly

Purpose Produce a cyclic code generator polynomial

Syntax

```
p = cyclpoly(n,k);
p = cyclpoly(n,k,fd_flag);
```

Description `p = cyclpoly(n,k)` finds a nontrivial cyclic generator polynomial for a given codeword length `n` and number of information bits `k`.

`p = cyclpoly(n,k,fd_flag)` finds a cyclic generator polynomial with specified properties for a given codeword length `n` and number of information bits `k`. *`fd_flag`* is a string variable that allows you to select what kind of cyclic polynomial to use in the cyclic code computation. The table below lists the options available:

fd_flag	Meaning
'min'	Find cyclic generator polynomial with minimum degree
'max'	Find cyclic generator polynomial with maximum degree
'exh'	Perform an exhaustive search for all primitive polynomials of the given degree
L	Find all cyclic generator polynomials with L terms

When the output `p` is empty, there is no primitive polynomial found for the given conditions.

See Also `cyclgen`

Purpose

Digital demodulation, passband simulation

Syntax

```
ddemod(method);
x= ddemod(y, Fc, Fd, Fs, method, ...);
[x, t]= ddemod(y, Fc, Fd, Fs, method, ...);
```

Description

Function `ddemod` recovers a digital signal from a signal modulated by `dmod`. The demodulation method, specified by the string variable method, must correspond with the type of modulation used in `dmod`. The table below lists the types of demodulation available:

Method	Modulation Scheme
'ask'	Amplitude shift-keying modulation
'fsk'	Frequency shift-keying modulation
'msk'	Minimum shift-keying modulation
'psk'	Phase shift-keying modulation
'qask'	Quadrature amplitude shift-keying modulation
'sample'	Sampling `Fd` frequency signal to `Fs` frequency signal

`ddemod(method)` provides more detailed help on specific demodulation methods.

`rec_sig = ddemod(y, Fc, Fd, Fs, method, ...)` recovers the modulated digital signal. `rec_sig`, the digital signal to be recovered, has frequency `Fd`. The carrier frequency for the modulation is `Fc` (Hertz); the simulation sample frequency is `Fs` (Hertz). `Fs/Fd` must be a positive integer. When input `y` is a matrix, each column of `y` is processed independently. The row size of the output matrix `rec_sig` is:

```
Fd/Fs*row_size_of_y
```

The time interval between two consecutive points of input `y` is `1/Fs`, and the time interval between two consecutive points of output `rec_sig` is `1/Fd`. The time interval between two consecutive points in `x` is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency;

the second element, if present, is the initial phase in the carrier signal. The initial phase is in radians, and its default is zero.

For best results, `Fs > Fc > Fd` is recommended.

The time interval between two decision points is `1/Fd`. The default offset for the decision point is 0. When `Fd` is a two-element vector, the second element in `Fd` is the decision timing offset. This function measures the distance of the received signal to all possible digits in the coding scheme. The digit closest to the received signal point is selected as the output digit.

A digital demodulation includes two parts: analog demodulation and analog to digital demapping. To obtain the intermediate result, you can break the command:

```
rec_sig = ddemod(y, Fc, Fd, Fs, method, ...)
```

into two commands:

```
z = ddemod(y, Fc, Fd, Fs, [method, '/nomap'], ...);
rec_sig = demodmap(z, Fd, Fs, method, ...);
```

In this example, the switch `'/nomap'` disables the mapping process. The intermediate signal `z` has a sample frequency `Fs`.

The function plots an eye-pattern diagram when the string `'/eye'` is appended to the variable *method*; appending the string `'/scatter'` to the variable *method* produces a scatter plot. The function provides only one plot if both `'/eye'` and `'/scatter'` are specified. You can use the option `'/scatter'` with the `'ask'` and `'qask'` methods only.

method = 'ask'

`rec_sig = ddemod(y, Fc, Fd, Fs, 'ask', M)` demodulates an M-ary ASK modulated signal. The M-ary number for the M-ary ASK is `M`. The output digital signal is an integer in the range [0, M–1]. The function uses a default lowpass filter; the numerator `num` and denominator `den` are generated by using the command `[num, den] = butter(5, Fc*2/Fs)`.

`rec_sig = ddemod(y, Fc, Fd, Fs, 'ask', M, num, den)` specifies the denominator and numerator of the demodulation lowpass filter.

method = 'psk'
`rec_sig = ddemod(y,Fc,Fd,Fs,'psk',M)` demodulates an M-ary PSK modulated signal. The M-ary number for the M-ary PSK is `M`. The output digital signal is an integer in the range `[0, M-1]`. This function uses the correlation PSK demodulation method.

method = 'qask'
This method supports three different constellations: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by the M-ary number `M`. Usually, `M` equals 2^K, where `K` is a positive integer. The maximum in-phase and quadrature values in the constellation are:

$M=2^K$	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- `noc` (Number of symbols on each circle),
- `roc` (Radii on each circle), and
- `poc` (Phase shift on each circle, in radians).

The M-ary number of this block equals the summation of all elements in `noc`. Let `noc` be $[N_1, N_2,..., N_K]$; `roc` be $[R_1, R_2,..., R_N]$; and `poc` be $[P_1, P_2,..., P_K]$. Then the `i`th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the ith circle with their radii equal R_i. One of the points on the ith circle has

phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \quad R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \cdots \quad R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors `noc` and `roc` must be positive. In any of the three vectors, the elements can be repeated. For a correct demapping, the elements should have no repeating points.

You can define an arbitrary constellation by using two of same length vectors. The two vectors are the in-phase component, `in_phase`, and the quadrature component, `quad`. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol i, $i = 0, 1, \ldots, M-1$, are defined in the (i+1)th element in the `in_phase` vector and in the `quad` vector respectively.

In all the constellations discussed in this section, this function requires the use of a lowpass filter. In all cases where `num` and `den`, the numerator and denominator of the filter's transfer function, are not specified, the function generates a default filter transfer function by using the command:

```
[num,den] = butter(5,Fc*2/Fs).
```

You can override the default by specifying `num` and `den` in the function parameter list.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask',M)` demodulates a square constellation QASK modulated signal. The M-ary number for QASK is `M`. The output digital signal is in range `[0,M-1]`.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask',M,num,den)` specifies the numerator and denominator of the demodulation lowpass filter.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask/arb',in_phase,quad)` demodulates a QASK modulated signal by means of a user-defined arbitrary constellation. The in-phase quadrature components are defined in the variables `in_phase` and `quad` respectively. The vectors length of `in_phase` and `quad` must be the same; the length is the M-ary number for the QASK.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask/arb',in_phase,quad,num,den)` specifies the numerator and denominator of the demodulation lowpass filter.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask/circle',noc,roc,poc)` demodulates a QASK modulated signal using a circle constellation. The number of points on the circles (`noc`), radii of the circles (`roc`), and the phase of the points of the circle (`poc`) define the QASK constellation. The vector lengths of `noc`, `roc`, and `poc` must be equal.

`rec_sig = ddemod(y,Fc,Fd,Fs,'qask/circle',noc,roc,poc,num,den)` specifies the numerator and denominator of the demodulation lowpass filter.

method = 'fsk'

`rec_sig = ddemod(y,Fc,Fd,Fs,'fsk',M,tone)` demodulates a FSK modulated signal using the coherence method. The M-ary number for QASK is `M`. The tone space between two successive frequencies is defined in `tone`.

`rec_sig = ddemod(y,Fc,Fd,Fs,'fsk/nonco',M,tone)` demodulates a FSK modulated signal using the noncoherence method.

method = 'msk'

`rec_sig = ddemod(y,Fc,Fd,Fs,'msk')` demodulates the input signal `y` using the minimum shift-keying method. The elements in the output vector `rec_sig` are binary numbers. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = Fd`.

method = 'psk'

`rec_sig = ddemod(y,Fc,Fd,Fs,'psk',M)` demodulates the input signal `y` using phase shift-keying method. The M-ary number is specified by `M`.

method = 'sample'

`rec_sig = ddemod(y,Fc,Fd,Fs,'sample')` downsamples input signal `y` from the input sample rate `Fs` to the output sample rate `Fd`. `Fs / Fd` must be a positive integer.

See Also

`dmod`, `dmodce`, `ddemodce`, `modmap`, `demodmap`

ddemodce

Purpose Digital demodulation, baseband simulation with complex envelope

Syntax

```
ddemodce(method);
rec_sig = ddemodce(y, Fd, Fs, method, ...);
```

Description Function `ddemodce` is designed to recover a digital signal that was modulated by using `dmodce`. This function assumes the input signal is a complex variable that contains the complex envelope of the modulated signal. The demodulation method, specified by the string variable method, must correspond with the type of modulation used in `dmodce`. The table below lists the types of demodulation available:

Method	Modulation Scheme
'ask'	M-ary amplitude shift-keying modulation
'fsk'	M-ary frequency shift-keying modulation
'msk'	Minimum shift-keying modulation
'psk'	M-ary phase shift-keying modulation
'qask'	Quadrature amplitude shift-keying modulation
'sample'	Downsampling `Fd` frequency signal to `Fs` frequency signal utility

Note that `'sample'` is not a modulation scheme, but a utility that resamples the input signal. `ddemodce(method)` provides more detailed help on specific demodulation methods.

`rec_sig = ddemod(y, Fd, Fs, method, ...)` recovers the modulated digital signal. `rec_sig`, the digital signal to be recovered, has frequency `Fd`. The simulation sample frequency is `Fs` (Hertz). `Fs/Fd` must be a positive integer. When input `y` is a matrix, each column of `y` is processed independently. The row size of the output matrix `rec_sig` is:

```
Fd/Fs*row_size_of_y
```

The time interval between two consecutive points of input `y` is `1/Fs`, and the time interval between two consecutive points of output `rec_sig` is `1/Fd`. The

variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency, and the second element, if present, is the initial phase.

The time interval between two decision points is `1/Fd`, and the default offset for the decision point is 0. When `Fd` is a two-element vector, the second element in `Fd` is the decision timing offset.

This function measures the distance of the received signal to all possible digits in the coding scheme and selects the digit closest to the received signal point as the output digit.

A digital demodulation includes two parts: analog demodulation and analog to digital demapping. To obtain the intermediate result, you can break the command:

```
rec_sig = ddemodce(y,Fd,Fs,method,...)
```

into two commands:

```
z = ddemodce(y,Fd,Fs,[method,'/nomap'],...);
rec_sig = demodmap(z,Fd,Fs,method,...);
```

In this example, the switch `'/nomap'` is appended to the method to disable the mapping process. The intermediate signal `z` has a sample frequency `Fs`.

The function plots an eye-pattern diagram when the string '`/eye`' is appended to the variable *`method`*; appending the string '`/scatter`' to the variable *`method`* produces a scatter plot. The function provides only one plot if both '`/eye`' and '`/scatter`' are specified. You can use the option '`/scatter`' with the '`ask`' and '`qask`' methods only.

method = 'ask'

`rec_sig = ddemodce(y,Fd,Fs,'ask',M)` demodulates an ASK modulated signal. The M-ary number for ASK is `M`. The output digital signal is an integer in the range [0, M–1]. There is no default lowpass filter used in this method format.

`rec_sig = ddemodce(y,Fd,Fs,'ask',M,num,den)` specifies the denominator and numerator of an optional demodulation lowpass filter.

method = 'psk'

`rec_sig = ddemodce(y,Fd,Fs,'psk',M)` demodulates a PSK modulated signal. The M-ary number for PSK is `M`. The output digital signal is an integer

in the range [0, M–1]. This function uses the correlation PSK demodulation method.

method = 'qask'
This function supports three different constellations: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by the M-ary number M. Usually, M equals 2^K, where K is a positive integer. The maximum in-phase and quadrature values in the constellation are:

$M=2^K$	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- noc (Number of symbols on each circle),
- roc (Radii on each circle), and
- poc (Phase shift on each circle, in radians).

The M-ary number of this block equals the summation of all elements in noc. Let noc be [N_1, N_2,..., N_K]; roc be [R_1, R_2,..., R_N]; and poc be [P_1, P_2,..., P_K]. Then the i th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the ith circle with their radii equal R_i. One of the points on the ith circle has

phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \; R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \; \ldots \; R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors noc and roc must be positive. In any of the three vectors, the elements can be repeated. For a correct demapping, the elements should have no repeating points.

You can define an arbitrary constellation by using two of same length vectors. The two vectors are the in-phase component, in_phase, and the quadrature component, quad. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol i, $i = 0, 1, \ldots, M-1$, are defined in the (i+1)th element in the in_phase vector and in the quad vector respectively.

In all the constellations discussed in this section, this function requires the use of a lowpass filter. In all cases where num and den, the numerator and denominator of the filter's transfer function, are not specified, the function generates a default filter transfer function by using the command:

```
[num,den] = butter(5,Fc*2/Fs)
```

You can override the default by specifying num and den in the function parameter list.

rec_sig = ddemodce(y,Fd,Fs,'qask',M) demodulates a square constellation QASK modulated signal. The M-ary number for QASK is M. The output digital signal is in the range [0, M–1].

rec_sig = ddemodce(y,Fd,Fs,'qask',M,num,den) specifies the denominator and numerator of the demodulation lowpass filter.

rec_sig = ddemodce(y,Fd,Fs,'qask/arb',in_phase,quad) demodulates a QASK modulated signal with a user-defined arbitrary constellation. The in-phase component and the quadrature component are defined in the variables in_phase and quad respectively. The vector lengths of in_phase and quad must be equal; the length equals the M-ary number used in the QASK modulation.

`rec_sig = ddemodce(y,Fd,Fs,'qask/arb',in_phase,quad,num,den)` specifies the denominator and numerator of the demodulation lowpass filter.

`rec_sig = ddemodce(y,Fd,Fs,'qask/circle',noc,roc,poc)` demodulates a QASK modulated signal using circle constellation. The number on circle (`noc`), amplitude on circle (`roc`) and the phase on circle (`poc`) define the QASK constellation. The vectors length of `noc`, `roc` and `poc` must be of equal length.

`rec_sig = ddemodce(y,Fd,Fs,'qask/circle',noc,roc,poc,num,den)` specifies the denominator and numerator of the demodulation lowpass filter.

method = 'fsk'

`rec_sig = ddemodce(y,Fd,Fs,'fsk',M,tone)` demodulates a FSK modulated signal using coherence method. The M-ary number for QASK is `M`. The tone space between two successive frequencies is defined in `tone`.

`rec_sig = ddemodce(y,Fd,Fs,'fsk/nonco',M,tone)` demodulates a FSK modulated signal using noncoherence method.

method = 'msk'

`rec_sig = ddemodce(y,Fd,Fs,'msk')` demodulates the input signal `y` using minimum shift-keying method. The elements in the output vector `x` are binary numbers. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = Fd`.

method = 'psk'

`rec_sig = ddemodce(y,Fd,Fs,'psk',M)` modulates the input signal `y` using phase shift-keying method. The M-ary number is defined in `M`.

method = 'sample'

`rec_sig = ddemodce(y,Fd,Fs,'sample')` converts sampling rate `Fs` input signal `y` to sampling rate `Fd` output signal `x`, where `Fs > Fd` and `Fs / Fd` is a positive integer.

See Also

`dmod`, `dmodce`, `ddemod`, `modmap`, `demodmap`

Purpose Positive decimal integer to binary row vector conversion

Syntax

```
bi = de2bi(de);
bi = de2bi(de,n);
```

Description `bi = de2bi(de)` converts a positive decimal number `de` to a binary row vector. When `de` is a vector, the output of `de2bi`, `bi`, is a matrix. Each row of `bi` is the binary conversion of the corresponding element in `de`.

`bi = de2bi(de,n)` specifies the length of each row being `n`.

Examples

```
be = randint(5,1,[0 3]);    % Generate a 5-by-1 random binary
                   % matrix.
bi = de2bi(de,3);
```

The last line outputs a 5-by-3 matrix in which each row is the binary conversion of each decimal element of `de`.

See Also `bi2de`

decode

Purpose Error-control decoding

Syntax

```
msg = decode(code, N, K);
msg = decode(code, N, K, method, opt1, opt2, opt3, opt4);
[msg, err] = decode(...);
[msg, err, ccode] = decode(...);
[msg, err, ccode, cerr] = decode(...);
```

Description

`decode` is used to recover the original message from a received codeword using error-control decoding. The decode method and its parameters must exactly match the method that was used in encoding the original signal.

`msg = decode(code, N, K)` decodes binary messages using Hamming decoding technique. The codeword length is `N` and the number of information bits is `K`. The format of code can be either a vector or an `N` column matrix. A Hamming code has a codeword length 2^M-1, where `M` is an integer number greater than or equal to 3. The number of information bits is 2^M-N-1. Hamming code is a single error-correction code.

`msg = decode(code, N, K, method, opt1, opt2, opt3, opt4)` decodes the codeword using an error-control coding method specified by the string variable *`method`*. In all cases, the number of information bits is `K` and the codeword length is `N`, where `N` and `K` are integers and `N>K`. `opt1` to `opt4` are extra optional parameters used in some of the methods:

Method	Decode Scheme
'hamming'	Hamming code. `opt1=N`, the codeword length. `opt2=K`, the number of information bits. `opt1` can also be used to specify the primitive polynomial. The function uses a default primitive polynomial if `opt1` is not specified. `opt2` to `opt4` are not used in this method.
'linear'	Linear block code. `opt1=N`, the codeword length. `opt2=K`, the number of information bits. `opt1` must be used to specify the parity-check matrix. `opt2` specifies the error-detection logic circuit; when `opt2` is not provided, this function defaults to single-error-correction logic. `opt3` to `opt4` are not used.

Method	Decode Scheme
'cyclic'	Cyclic code. `opt1` is the generator polynomial, which you must specify. Use function `cyclpoly` to choose a suitable cyclic polynomial. `opt2` specifies the error detection logic; when `opt2` is not provided, this function defaults to single error-correction logic. `opt3` and `opt4` are not used.
'bch'	Binary BCH code. `opt1` specifies the error-correction capability. If you do not specify `opt1`, this function uses function `bchpoly` to find the error-correction capability. `opt2` specifies a BCH code generator polynomial. If you do not specify `opt2`, this function uses a default generator polynomial. Use the command `bchpoly` to view the valid (N, K) pairs for BCH code. `opt3` and `opt4` are not used.
'rs'	Reed-Solomon code. The codewords for RS code have length equal to 2^M-1, where M is an integer greater than or equal to 3. The error-correction capability is `T=floor((N-K)/2)`. Set `N-K` to an even number. `opt1` contains a list of all elements in GF(2^M). If you do not specify `opt1`, this function computes all the elements of GF(2^M) before proceeding to the encode calculations. `opt2` through `opt4` are not used.

Method	Decode Scheme
'convol'	Convolutional code using the Viterbi algorithm to decode. You must specify `opt1`, which can be the linear, nonlinear, or octal form of the transfer function matrix of the convolutional code. Read the description for `gen2abcd` for a definition of the octal form. `opt2` is an integer that specifies the memory length. When `opt2` is a nonpositive integer or is not specified, this function uses an unlimited memory length scheme. `opt3` specifies the transfer probability. The transfer probability is defined using a three row matrix in this toolbox. Refer to the description of function `viterbi` for a definition of the transfer probability. When `opt3` is not specified or is not a three row matrix, this function uses the "hard decision" decoding scheme. `opt4` is a nonnegative integer that specifies the figure number for plotting the trellis plot. If `opt4`≤0 or if it is omitted, this function suppresses the trellis figure plot.

The message signal in `code` can be a binary vector or an L-column matrix. Except for the `'rs'` method, `L = N`, the codeword length. In the `'rs'` method, `L = M`, where `M` is an integer greater than or equal to 3, and the codeword length $N=2^M-1$.

This function also supports decimal integer coding. This means that the message signal in `code` can be a non-negative integer in the range of $[0, 2^L-1]$. In this case, you have to append `'/decimal'` to the string variable *`method`*. For example, `'bch'` is equivalent to `'bch/binary'`, but `'bch/decimal'` means that the data in code is a decimal integer. Except for the `'rs'` method, the output `msg` is a vector when using the decimal method.

For the `'rs'` method, you have the option to specify `'rs/power'`. This tells the function that the code is a representation in the power format in $GF(2^M)$. The data range in the power format is –Inf to 2^M-2. (Note codeword length $N = 2^M-1$). In both `'rs/decimal'` and `'rs/power'` case, `msg` can be a vector or an N column matrix, where N is the codeword length. In the binary case, code can be either a vector or an M column matrix.

The output msg contains the recovered message. The format of msg matches the format of code. When code is an N-column matrix, the output msg is a K-column matrix.

When the format of the inputted code is not the same as the output of the function encode, this function stops processing.

[msg, err] = decode(...) indicates the number of errors detected in the decoding process. If the element in err is a negative number, it means that more errors occurred during processing than the error-correction algorithm was capable of handling. In the 'convol' method, instead of the number of errors corrected, the parameter err contains the metric calculations used in the decoding decision process.

[msg, err, ccode] = decode(...) outputs the corrected code in ccode.

[msg, err, ccode, cerr] = decode(...) outputs the errors corresponding to each row of ccode. In the 'convol' method, instead of the number of errors corrected, the parameter cerr contains the metric calculations used in the decoding decision process.

See Also

encode, bchpoly, bchdeco, rspoly, rsdeco, cyclpoly, cyclgen, sim2gen, viterbi

demodmap

Purpose Digital demodulation demapping

Syntax

```
demodmap(method);
demodmap(y, Fd, Fs, method);
demap_sig = demodmap(y, Fd, Fs, method, ...);
[x, t]= demodmap(y, Fd, Fs, method, ...);
```

Description A digital demodulation can be broken into two stages: analog signal demodulation and analog to digital signal demapping. The function `demodmap` demaps an analog signal to a digital signal; the resulting digital signal is the closest point to the input analog signal. The demapping result is different for different digital modulation methods. You can specify the modulation method using the string variable *method*. The input vector size for demapping varies for different methods.

Method	Digital Modulation Scheme	Input Size
'ask'	M-ary amplitude shift-keying modulation	no limit
'fsk'	M-ary frequency shift-keying modulation	no limit
'msk'	Minimum shift-keying modulation	no limit
'psk'	M-ary phase shift-keying modulation	even column
'qask'	Quadrature amplitude shift-keying modulation	even column
'sample'	A utility for resampling `Fd` frequency signal to `Fs` frequency signal	no limit
'eye'	Eye pattern diagram plot	no limit
'scatter'	Scatter plot	no limit

Note that 'same', 'eye', and 'scatter' are utilities that support demodulation and not digital demodulation schemes themselves. `demodmap(method)` provides more detailed help on a specific demodulation method.

`demodmap(y, Fd, Fs, method)` plots an eye-pattern diagram or scatter plot. This format can be used with method `'eye'` or `'scatter'` only.

`demap_sig = demodmap(y, Fd, Fs, method, ...)` recovers the mapped digital signal using the modulation method specified in variable `method`. `demap_sig`, the output digital signal to be recovered, has frequency `Fd`. The simulation sample frequency is `Fs` (Hertz), and the input signal `y` has sampling frequency `Fs`. `Fs/Fd` must be a positive integer.

When input `y` is a matrix, each column of `y` is processed independently, except in the cases where *method* equals `'qask'` or `'psk'`. In these cases, the input matrix signal is processed two columns at a time. The first column is the in-phase component of the input signal, and the second column is the quadrature component.The row number of the output matrix `demap_sig` is:

```
Fd/Fs*number_of_rows_of_y
```

The time interval between two consecutive points in `y` is `1/Fs`, while the time interval between two consecutive points in `demap_sig` is `1/Fd`.The time interval between two consecutive points in `x` is `1/Fs`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default is zero.

The time interval between two decision points is `1/Fd`. The default offset for the decision point is 0. When `Fd` is a two-element vector, the second element in `Fd` is the decision timing offset. This function measures the distance of the received signal to all possible digits in the coding scheme. The digit closest to the received signal point is selected as the output digit.

A digital demodulation includes two parts: analog demodulation and analog to digital demapping. This function can help you obtain the intermediate result. You can break the command:

```
rec_sig = ddemod(y, Fc, Fd, Fs, method, ...);
```

into two commands:

```
z = ddemod(y, Fc, Fd, Fs, [method, '/nmap'], ...);
rec_sig = demodmap(z, Fd, Fs, method, ...);
```

In this example, the switch `'/nmap'` is appended to the method that disables the mapping process. The intermediate signal `z` has a sample frequency `Fs`.

Except the format of method equals `'eye'` and `'scatter'`, the function can also plot an eye-pattern diagram when an amendment `'/eye'` is added to the end of the variable `method`. The function plots a scatter plot when an amendment `'/scatter'` is added to the end of the variable method. The function provides only one plot if both of '/eye' and '/scatter' are specified. The option `'/scatter'` is good for use with 'ask' and 'qask' methods only.

method = 'ask'

`rec_sig = demodmap(y,Fd,Fs,'ask',M)` demaps an M-ary ASK mapped signal. The M-ary number for the M-ary ASK is `M`. The output digital signal is an integer in the range [0, M–1].

method = 'psk'

`rec_sig = demodmap(y,Fd,Fs,'psk',M)` demaps an M-ary PSK modulated signal. The M-ary number for the M-ary PSK is `M`. The output digital signal is an integer in the range [0, M–1].

method = 'qask'

This function supports three constellations: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by the M-ary number `M`. Usually, $M=2^K$, where `K` is a positive integer. The maximum in-phase and quadrature values in the constellation are:

$M=2^K$	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- noc (Number of symbols on each circle),
- roc (Radii on each circle), and
- poc (Phase shift on each circle, in radian).

The M-ary number of this block equals the summation of all elements in noc. Let noc be $[N_1, N_2, \ldots, N_K]$; roc be $[R_1, R_2, \ldots, R_N]$; and poc be $[P_1, P_2, \ldots, P_K]$. Then the i th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the ith circle with their radii equal R_i. One of the points on the ith circle has phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \quad R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \quad \ldots \quad R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors noc and roc must be positive. For a correct demapping, the elements should have no repeating points.

You can define an arbitrary constellation by using two of same length vectors. The two vectors are the in-phase component, in_phase, and the quadrature component, quad. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol *i*, *i = 0, 1, . . . , M–1*, are defined in the (i+1)th element in the in_phase vector and in the quad vector respectively.

`rec_sig = demodmap(y, Fd, Fs, 'qask', M)` demaps a square constellation QASK mapped signal. The M-ary number for QASK is `M`. The output digital signal is in the range [0, M–1].

`rec_sig = demodmap(y, Fd, Fs, 'qask/arb', in_phase, quad)` demaps a QASK mapped signal using a customer-defined arbitrary constellation. The in-phase component and the quadrature component are defined in the variables in_phase and quad respectively. The vectors length of in_phase and quad must be the same, the length is the M-ary number for the QASK.

`rec_sig = demodmap(y, Fd, Fs, 'qask/circle', noc, roc, poc)` demaps a QASK mapped signal using circle constellation. The number on circle (`noc`), radii on circle (`roc`) and the phase on circle (`poc`) defines the QASK constellation. The vector length of `noc`, `roc`, and `poc` must be the same.

method = 'fsk'

`rec_sig = demodmap(y, Fd, Fs, 'fsk', M, tone)` demaps a FSK mapped signal using the coherent method. The M-ary number for QASK is `M`. The tone space between two successive frequency is defined in `tone`.

method = 'msk'

`rec_sig = demodmap(y, Fd, Fs, 'msk')` demaps a MSK mapped signal. The elements in the output vector `x` are binary numbers. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = Fd`.

method = 'psk'

`rec_sig = demodmap(y, Fd, Fs, 'psk', M)` demaps a PSK mapped signal. The M-ary number is defined in `M`.

method = 'sample'

`rec_sig = demodmap(y, Fd, Fs, 'sample')` converts sampling rate `Fs` input signal `y` to sampling rate `Fd` output signal `x`, where `Fs > Fd` and `Fs / Fd` are positive integers.

method = 'eye'

`demodmap(y, Fd, Fs, 'eye')` plots an eye-pattern diagram. The red vertical line with a "D point" mark indicates the decision point line by the given sample time.

method = 'scatter'

`demodmap(y, Fd, Fs, 'scatter')` produces a scatter plot from the input signal `y`.

See Also

`dmod`, `demod`, `dmodce`, `ddemodce`, `modmap`

Purpose

Digital modulation, passband simulation

Syntax

```
dmod(method);
dmod(method,...);
y = dmod(x,Fc,Fd,Fs,method,...);
[y,t]= dmod(x,Fc,Fd,Fs,method,...);
```

Description

Function `dmod` modulates an input signal and is designed for passband digital modulation simulation. You can specify the modulation method using the string variable method. The table below lists the types of modulation available:

Method	Modulation Scheme
`'ask'`	M-ary amplitude shift-keying modulation
`'fsk'`	M-ary frequency shift-keying modulation
`'msk'`	Minimum shift-keying modulation
`'psk'`	M-ary phase shift-keying modulation
`'qask'`	Quadrature amplitude shift-keying modulation
`'sample'`	Resampling frequency signal `Fd` to frequency signal `Fs`

Note that `'sample'` is a utility that supports modulation and not a modulation scheme itself.

`dmod(method)` provides more detailed help on a specific modulation method.

`dmod(method,...)` plots a constellation of signal points for the type of modulation selected.

`y = dmod(x,Fc,Fd,Fs,method,...)` modulates the digital message signal `x` with carrier frequency `Fc` (Hertz) and simulation sample frequency `Fs` (Hertz). The output signal `y` has sampling frequency `Fs`, and the input signal `x` has sampling frequency `Fd`. `Fs/Fd` must be a positive integer.

When the input `x` is a matrix, each column of `x` is processed independently. The row number of the output matrix `y` is:

```
Fs/Fd*row_number_of_x
```

The time interval between two consecutive points in `y` is `1/Fs` while the time interval between two consecutive points in `x` is `1/Fd`. The variable `Fs` can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default value is zero. For best results, `Fs > Fc > Fd` is recommended.

`[y,t] = dmod(...)` outputs the computation time `t`. `t` is a vector with its length equal to the row number of `y`.

A digital modulation includes two parts: digital to analog mapping and analog modulation. To obtain the intermediate result, you can break the command:

```
y = dmod(x, Fc, Fd, Fs, method, ...)
```

into two commands:

```
z = modmap(x, Fd, Fs, method, ...);
y = dmod(z, Fc, Fd, Fs, [method, '/nmap'], ...);
```

In this example, the switch `'/nmap'` appended to *method* disables the mapping process. The intermediate signal `z` has a sample frequency `Fs`.

method = 'ask'

`dmod('ask',M)` plots the ASK constellation.

`y = dmod(x,Fc,Fd,Fs,'ask')` modulates the input signal `x` using the amplitude shift-keying method. All elements in `x` are non-negative integers. The input integer is in the range `[0, M-1]`, and `M` = 2^K such that `max(x)`<2^K, where `K` is the minimum integer that satisfies the inequality.

`y = dmod(x,Fc,Fd,Fs,'ask',M)` specifies the M-ary number `M`.

method = 'fsk'

`dmod('fsk',M)` plots the FSK constellation.

`y = dmod(x,Fc,Fd,Fs,'fsk')` modulates the input signal `x` using the frequency shift-keying method. All elements in `x` are non-negative integers. The range of the input integers is `[0, M-1]`, `M` = 2^K. `K` is the minimum integer satisfying `max(x)`<2^K. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = 2*Fd/M`.

`y = dmod(x,Fc,Fd,Fs,'fsk',M,tone)` specifies the input digital signal range `M` and the tone space `tone`.

method ='msk'

`dmod('msk',M)` plots the MSK constellation.

`y = dmod(x,Fc,Fd,Fs,'msk')` modulates the input signal `x` using the minimum shift-keying method. All elements in `x` must be binary numbers. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = Fd`.

method = 'psk'

`dmod('psk',M)` plots the PSK constellation.

`y = dmod(x,Fc,Fd,Fs,'psk')` modulates the input signal `x` using the phase shift-keying method. All elements in `x` are non-negative integers. The range of the input integers is `[0, M-1]`. $M = 2^K$ such that $\max(x) < 2^K$, where `K` is the minimum integer that satisfies the inequality.

`y = dmod(x,Fc,Fd,Fs,'psk',M)` specifies the input M-ary number M.

method = 'qask'

This function supports three constellations: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by the M-ary number. Usually, the M-ary number $M=2^K$, where `K` is a positive integer. The maximum in-phase and quadrature values in the constellation are:

$M=2^K$	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- noc (Number of symbols on each circle),
- roc (Radii on each circle), and
- poc (Phase shift on each circle, in radian).

The M-ary number of this block equals the summation of all elements in noc. Let noc be $[N_1, N_2,..., N_K]$; roc be $[R_1, R_2,..., R_N]$; and poc be $[P_1, P_2,..., P_K]$. Then the i th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the ith circle with their radii equal R_i. One of the points on the ith circle has phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \quad R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \quad \ldots \quad R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors noc and roc must be positive. For a correct demapping, they should have no repeating points.

You can define an arbitrary constellation by using two of same length vectors. The two vectors are the in-phase component, in_phase, and the quadrature component, quad. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol *i*, *i = 0, 1, . . . , M-1*, are defined in the (i+1)th element in the in_phase vector and in the quad vector respectively.

`dmod('qask',M)` plots the square QASK constellation.

`dmod('qask/arb',in_phase,quad)` plots the user-defined arbitrary QASK constellation.

`dmod('qask/circle',noc,roc,poc)` plots the circle constellation.

`y = dmod(x,Fc,Fd,Fs,'qask')` modulates the input signal x using the square constellation QASK method. All elements in x are non-negative integers. The range of the input integers is `[0,M-1]`, `M = 2`^{K}. `K` is the minimum integer satisfying `max(x)<2`^{K}, `K` is the minimum integer that satisfies the inequality.

`y = dmod(x,Fc,Fd,Fs,'qask',M)` specifies the M-ary number of the square constellation QASK.

`y = dmod(x, Fc, Fd, Fs, 'qask/arb', in_phase, quad)` modulates the input signal `x` using the arbitrary constellation QASK method with the in-phase component and quadrature component value provided in the variables `in_phase` and `quad`.

`y = dmod(x, Fd, Fs, 'qask/circle', noc, roc, poc)` modulates the input signal `x` using the circle constellation QASK method with the number of symbols on each circle, radii for each circle, and phase shift for each circle specified in variables `noc`, `roc`, and `poc` respectively.

method = 'sample'

`y = dmod(x, Fc, Fd, Fs, 'sample')` converts sampling rate `Fd` input signal `x` to sampling rate `Fs` output signal `y`, where `Fs>Fd` and `Fs/Fd` are positive integers.

See Also

`amod`, `ademod`, `ddemod`, `amodce`, `ademodce`, `modmap`, `demodmap`

Purpose Digital modulation, baseband modulation with complex envelope

Syntax

```
dmodce(method);
dmodce(method,...);
y = dmodce(x,Fd,Fs,method,...);
[y,t]= dmodce(x,Fd,Fs,method,...);
```

Description Function `dmodce` modulates an input digital signal with complex envelope and is designed for baseband digital modulation simulation. You can specify the modulation method using the string variable method. The table below lists the types of modulation available:

Method	Modulation Scheme	Output `y` type
'ask'	Amplitude shift-keying modulation	complex
'fsk'	Frequency shift-keying modulation	complex
'msk'	Minimum shift-keying modulation	complex
'psk'	Phase shift-keying modulation	complex
'qask'	Quadrature amplitude shift-keying modulation	complex
'sample'	Resampling frequency signal `Fd` to frequency signal `Fs`	real

Except for `'sample'`, the modulated signal output from this function is the complex envelope of the modulated signal, meaning that the outputs are complex numbers. Note that `'sample'` is a utility that supports modulation and not a modulation scheme itself.

`dmodce(method)` provides more detailed help on a specific modulation method.

`dmodce(method,...)` plots a constellation of the given method.

`y = dmodce(x,Fd,Fs,method,...)` modulates the digital message signal `x`. `Ts` is the sampling frequency of output signal `y`, and `Fd` is the sampling frequency of the input signal `x`. `Fs/Fd` must be a positive integer. When the input `x` is a

matrix, each column of x is processed independently. The row number of the output matrix y is:

```
Fs/Fd*row_number_of_x
```

The time interval between two consecutive points in y is 1/Fs while the time interval between two consecutive points in x is 1/Fd. The variable Fs can be a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase is in radians, and its default value is zero.

[y, t] = dmodce(...) outputs the computation time t. t is a vector with its length equal to the row number of y.

A digital modulation includes two parts: digital to analog mapping and analog modulation. To obtain the intermediate result, you can break the command:

```
y = dmodce(x, Fd, Fs, method, ...)
```

into two commands:

```
z = modmap(x, Fd, Fs, method, ...);
y = dmodce(z, Fd, Fs, [method, '/nmap'], ...);
```

In this example, the switch '/nmap' appended to method disables the mapping process. The intermediate signal z has a sample frequency Fs.

method = 'ask'

dmodce('ask', M) plots the M-ary ASK constellation.

y = dmodce(x, Fd, Fs, 'ask') modulates the input signal x using the amplitude shift-keying method. All elements in x are non-negative integers. The input integer is in the range [0, M–1], and $M = 2^K$ such that $\max(x) < 2^K$, where K is the minimum integer that satisfies the inequality.

y = dmod(x, Fd, Fs, 'ask', M) specifies the M-ary number M.

method = 'fsk'

dmodce('fsk', M, tone) plots the M-ary FSK constellation.

y = dmodce(x, Fd, Fs, 'fsk') modulates the input signal x using the frequency shift-keying method. All elements in x are non-negative integers. The range of the input integers is [0, M–1], $M = 2^K$. K is the minimum integer satisfying

max(x)<2^K. The tone space, i.e., the frequency separation between successive frequencies, defaults to tone = 2*Fd / M.

y = dmodce(x, Fd, Fs, 'fsk', M, tone) specifies the input digital signal range M and the tone space tone.

method = 'msk'
dmodce('msk', M) plots the MSK constellation.

y = dmodce(x, Fc, Fd, Fs, 'msk') modulates the input signal x using the minimum shift-keying method. All elements in x must be binary numbers. The tone space, i.e., the frequency separation between successive frequencies, defaults to tone = Fd.

method = 'psk'
dmodce('psk', M) plots the PSK constellation.

y = dmodce(x, Fd, Fs, 'psk') modulates the input signal x using the phase shift-keying method. All elements in x are non-negative integers. The range of the input integers is [0, M-1]. M = 2^K such that max(x)<2^K, where K is the minimum integer that satisfies the inequality.

y = dmodce(x, Fd, Fs, 'psk', M) specifies the input M-ary number M.

method = 'qask'
This function supports three constellation definitions: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by the M-ary number M. Usually, M=2^K, where K is a positive integer. The maximum in-phase and quadrature values in the constellation are:

M=2^K	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5

M=2^K	In-phase	Quadrature
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- noc (Number of symbols on each circle),
- roc (Radii on each circle), and
- poc (Phase shift on each circle, in radian).

The M-ary number of this block equals the summation of all elements in noc. Let noc be $[N_1, N_2, ..., N_K]$; roc be $[R_1, R_2, ..., R_N]$; and poc be $[P_1, P_2, ..., P_K]$. Then the i th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the ith circle with their radii equal R_i. One of the points on the i th circle has phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \; R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \; \ldots \; R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors noc and roc must be positive. For a correct demapping, there should have no repeating points.

You can define an arbitrary constellation by using two of same length vectors. The two vectors are the in-phase component, in_phase, and the quadrature component, quad. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol i, $i = 0, 1, ..., M-1$, are defined in the (i+1)th element in the in_phase vector and in the quad vector respectively.

`dmodce('qask',M)` plots the square QASK constellation.

`dmodce('qask/arb',in_phase,quad)` plots the user-defined arbitrary QASK constellation.

`dmodce('qask/circle',noc,roc,poc)` plots the circle constellation.

`y = dmodce(x, Fd, Fs, 'qask')` modulates the input signal `x` using the square constellation QASK method. All elements in `x` are non-negative integers. The range of the input integers is `[0, M-1]`, `M` = 2^K. K is the minimum integer satisfying `max(x)` < 2^K, where `K` is the minimum integer that satisfies the inequality.

`y = dmodce(x, Fd, Fs, 'qask', M)` specifies the M-ary number of the square constellation QASK.

`y = dmodce(x, Fd, Fs, 'qask/arb', in_phase, quad)` modulates the input signal `x` using the arbitrary constellation QASK method with the in-phase component and quadrature component value provided in the variables `in_phase` and `quad`.

`y = dmod(x, Fd, Fs, 'qask/circle', noc, roc, poc)` modulates the input signal `x` using the circle constellation QASK method with the number of symbols on each circle, radii for each circle, and phase shift for each circle specified in variables `noc`, `roc`, and `poc` respectively.

method = 'sample'

`y = dmodce(x, Fd, Fs, 'sample')` converts sampling rate `Fd` input signal `x` to sampling rate `Fs` output signal `y`, where `Fs/Fd` is a positive integer.

See Also

`amod`, `ademod`, `dmod`, `ddemod`, `amodce`, `ademodce`, `modmap`, `demodmap`

Purpose Differential pulse code modulation decode computation in source coding

Syntax

```
sig = dpcmdeco(indx, codebook, predictor);
[sig, quant] = dpcmdeco(indx, codebook, predictor);
```

Description `sig = dpcmdeco(indx, codebook, predictor)` decodes a differential pulse code modulation encoded index signal, `indx`. You must specify the predictors in `predictor` and the predictive-error quantization codebook in `codebook`. To obtain a correctly decoded result, the parameters must match the parameters used in `encode`.

`[sig, quant] = dpcmdeco(indx, codebook, predictor)` outputs the quantized predictive error in `quant`.

The input parameters `codebook` and `predictor` can be estimated by using function `dpcmopt`.

In general, the predictor is an `M`th order transfer function has the form $[0, t_1, t_2, ..., t_M]$.

See Also `dpcmenco`, `dpcmopt`, `compand`

dpcmenco

Purpose

Differential pulse code modulation encode computation in source coding

Syntax

```
indx = dpcmenco(sig, codebook, partition, predictor)
[indx, quant] = dpcmenco(sig, codebook, partition, predictor)
```

Description

`indx = dpcmenco(sig, codebook, partition, predictor)` outputs a differential pulse code modulation encoded index, `indx`. `sig` is the input signal, and `predictor` specifies the predictive transfer function. The predictive-error quantization parameters are given in `partition` and `codebook`. In general, an `M`th order transfer function `predictor` has the form of [0, t_1, t_2, ... t_M].

`[indx, quant] = dpcmenco(sig, codebook, partition, predictor)` outputs the quantized value in `quant`.

The input parameters `codebook`, `partition`, and `predictor` can be estimated by using `dpcmopt`.

When `predictor` is an order one transfer function, the `dpcm` is simplified to be a delta-modulation.

See Also

`dpcmdeco`, `dpcmopt`, `compand`, `quantiz`

Purpose

Optimal training of differential pulse code modulation parameters by using training data

Syntax

```
predictor = dpcmopt(training_set, ord)
[predictor, codebook, partition] =
   dpcmopt(training_set, ord, ini_codebook)
```

Description

`predictor = dpcmopt(training_set, ord)` estimates the predictive transfer function `predictor` of a given order `ord` and the training set `training_set`.

`[predictor, codebook, partition] = dpcmopt(training_set, ord, ini_codebook)` produces the predictive transfer function `predictor`, and optimized `codebook` and `partition` for the predictive-error. The input variable `ini_codebook` can be either the initial values of the `codebook` vector or the size of the vector `codebook`.

See Also

`dpcmenco`, `dpcmdeco`, `quantiz`, `lloyds`

encode

Purpose

Error-control encoding

Syntax

```
code = encode(msg, N, K);
code= encode(msg, N, K, method, opt);
[code, added] = encode(...);
```

Description

`encode` generates a codeword from a message signal by adding parity check bits to the message. There are six major error-control code methods implemented in this toolbox. These methods are: Hamming code, linear block code, cyclic code, BCH code, Reed-Solomon code, and convolutional code. You can use `encode` for all these error-control methods.

`code= encode(msg, N, K)` encodes a binary message `msg` using Hamming code technique. The number of information bits is `K`, and the codeword length is `N`. The format of `msg` can be either a vector or a `K` column matrix. A Hamming code has a codeword length 2^M-1, where `M` is an integer number greater than or equal to 3. The number of information bits is 2^M-N-1. Hamming code is a single error-correction code.

`code= decode(msg, N, K, method, opt)` encodes the message signal `msg` using an error-control coding specified in *`method`*. The number of information bits is `K`, and the codeword length is `N`. `opt` is an optional parameter used in some of the methods. The table below lists the types of error-control code available with the associated variable string:

Method	Encode Scheme
`'hamming'`	Hamming code. `opt` can be used to specify the primitive polynomial. The function uses a default primitive polynomial if `opt` is not specified.
`'linear'`	Linear block code. `opt` must specify the parity-check matrix.
`'cyclic'`	Cyclic code. `opt`, which must be specified, contains the generator polynomial. You can use function `cyclpoly` to choose an appropriate cyclic polynomial.

Method	Encode Scheme
'bch'	Binary BCH code. opt is used to specify BCH code generator polynomial. The function uses a default generator polynomial if opt is not specified. Use command bchpoly to view the valid (N,K) pairs for BCH code.
'rs'	Reed-Solomon code. The codeword for RS code must have length equal to 2^M-1, where M is an integer no less than 3. The number of information bits K must be less than N. The error-correction capability T=floor((N-K)/2). Set N-K to an even number. opt should contain a list all elements in GF(2^M). If opt is not specified, this function computes all the elements of GF(2^M) prior to the encode calculation.
'convol'	Convolutional code. opt, which must be specified, can be a linear, nonlinear, or octal form of the transfer function matrix of the convolutional code. Read the description for gen2abcd for definitions of the different transfer function forms.

In all the methods, message signal in msg can be a binary vector or an L-column matrix. In using any method other than 'rs', L=N, the codeword length. In the 'rs' method, L=M, where N=2^M-1. M must be an integer greater than or equal to 3. code can be either a binary digit or an integer in the range of [0, 2^L-1]. In the latter case, you must append '/decimal' to the string variable *method* specification. For example, 'bch' is equivalent to 'bch/binary', but 'bch/decimal' means that the data in code is in an integer format. Except for the 'rs' method, msg can only be a vector when using decimal method (instead of a matrix).

For 'rs' method, you can also specify 'rs/power', which tells the function that the format for the input msg is in the power format in GF(2^M). The data range in power format is –Inf and 0 to 2^M-2. (Note that the codeword length N = 2^M-1). In both the 'rs/decimal' and 'rs/power' cases, msg can be a vector or an K column matrix, where K is the number of information bits. In the 'binary' case, code can be either a vector or an M column matrix.

The codeword output code contains the recovered message. The format of code matches the format of msg. When msg is an K-column matrix, the output code is an N-column matrix.

[code, added] = encode(...) outputs the number of columns that the function added in order to make the encoding an exact size for K length message input.

See Also

decode, bchpoly, bchdeco, rspoly, rsdeco, cyclpoly, cyclgen, sim2gen, viterbi

Purpose

Find the error-location polynomial for BCH and Reed-Solomon decoding

Syntax

```
[sigma, err] = errlocp(syndrome, t, tp, err);
```

Description

`[sigma, err] = errlocp(syndrome, t, tp, err)` calculates the error-location polynomial `sigma` from the vector `syndrome`, the error-correction capability `t`, the complete list of GF(2^m) elements `tp`, and error message information `err`. `type_flag = 1`, uses the normal method. `type_flag = 0` uses a simplified method. The simplified method can be used for binary code only.

Warning

This function has no error check. Please make sure all parameters are assigned correctly.

See Also

`bchdeco`, `rsdecode`

eyescat

Purpose

Eye-pattern diagram and/or scatter plot

Syntax

```
eyescat(x, Fd, Fs);
eyescat(x, Fd, Fs, offset);
eyescat(x, Fd, Fs, offset, Dpoint);
```

Description

An eye-pattern plot is a simple and convenient tool to study the effects of inter-symbol interference and other channel impairments for digital transmission. The received signal is plotted against time. When the *x*-axis time limit is reached, the signal goes back to the beginning of the time point. Thus, the plots overlay each other. Usually, the time range is an integer multiple of the digital transmission interval. The transmitted digital signal is a sampled data on the received analog signal. The sampling point is named as decision point. In the optimal case, the decision point is at the widest "eye" opening point.

A scatter plot records the signal value at a given decision point. In the best case, the decision point should be at the time when the eye is the most open in the eye-pattern diagram. When the input signal `x` is a two-column matrix, the scatter plot is a two-dimensional plot with the *x*-axis as the first input vector element and the y-axis as the second input vector element. The two-dimensional plot is widely used in the QAM application. If the input vector is not a two-column matrix, the plot is a one-dimensional plot.

For more information on eye-pattern and scatter plots, refer to Chapter 3, the *Tutorial*.

`eyescat(x, Fd, Fs)` plots the eye-pattern diagram. The time scale for the plot is from `0` to `1/Fd`. Signal `x` has sample frequency `Fs`. It is required that `Fs > Fd` and `Fs/Fd` is a positive integer.

`eyescat(x, Fd, Fs, offset)` plots an eye-pattern diagram with the time on the x-axis shift `offset` number of points, which is equivalent to `offset/Fs` length of time. The `offset` must be a positive integer. The eye-pattern diagram plots in the time range from `offset/Fd` to `offset/Fs+1/Fd`.

`eyescat(x, Fd, Fs, offset, Dpoint)` plots an eye-pattern diagram or a scatter plot depending on the contents in the variable `Dpoint`.

When Dpoint is a integer, the contents in Dpoint specify the decision point shift from the zero point; in other words, the decision point occurs at time equal to Dpoint/Fs+i/Fd.

When Dpoint is a string, the contents in Dpoint specifies the type of the symbols for the scatter plot. The string can be one of the following: * + o . - -. When a illegal string is specified, the function uses the default string '.'. The decision point is provided in offset in this case.

See Also

modmap, dmod, dmodce

flxor

Purpose Floating point number exclusive OR (XOR) computation

Syntax

```
z = flxor(x,y);
```

Description `z = flxor(x,y)` outputs exclusive OR computation between two integers `x` and `y`, or between elements of two equal size matrices. The matrix size of `x` must be the same as that of `y`.

See Also `gfplus`, `gfadd`

Purpose Convert a convolutional code transfer function into state space [A, B, C, D] form

Syntax

```
[A, B, C, D] = gen2abcd(gen);
[A, B, C, D] = gen2abcd(tran_func, code_param);
[A, B, C, D, N, K, M] = gen2abcd(...);
```

Description The input transfer function can be a nonlinear transfer function, a state space linear transfer function, an octal form, or a binary form. You can generate nonlinear transfer functions using `sim2logi`, state space transfer functions using `sim2tran`, and octal and binary forms using `oct2gen`.

`[A, B, C, D] = gen2abcd(gen)` converts the transfer function stored in `gen` into `[A, B, C, D]` format, where `A`, `B`, `C`, and `D` are matrices. When `gen` is a nonlinear transfer function as described in function `sim2logi`, the output `A` stores the `U` vector, and `B` stores the `V` vector. Vectors `U` and `V` vector are described in function `sim2logi`. When `gen` is a linear transfer function as described in function `sim2gen`, or an octal form transfer function as described in function `oct2gen`, the matrices `[A, B, C, D]` contain the binary coefficients of this difference equation:

$$x(k+1) = \mathrm{A}x(k) + \mathrm{B}u(k)$$
$$y(k) = \mathrm{C}x(k) + \mathrm{D}u(k)$$

`[A, B, C, D] = gen2abcd(tran_func, code_param)` converts the binary transfer function `tran_func` with `code_param`=[N, K, M] into `[A, B, C, D]` format. N is the output vector length, K is the input vector length, and M is the state vector length.

`[A, B, C, D, N, K, M] = gen2abcd(...)` outputs the length of the output, input and state vectors. `N` is the output vector length, `K` is the input vector length, and `M` is the state vector length.

Note that the conversion from the octal form to the state space `[A, B, C, D]` format conversion is not necessarily a minimal state space realization. You are encouraged to use the sim2tran instead of sim2gen to construct a linear transfer function.

See Also `encode`, `decode`, `sim2gen`, `sim2logi`, `sim2tran`, `oct2gen`

gen2par

Purpose

Conversion between generator matrix and parity-check matrix for linear block coding

Syntax

```
h = gen2par(g);
```

Description

`h = gen2par(g)` generates a parity-check matrix `h` from a generator matrix `g`. You can also use this function to convert a parity-check matrix into a generator matrix. This function works in conjunction with the linear block coding method.

In general, a generator matrix is a $k \times n$ matrix where $n > k$. The generator matrix is in the form $G = \left[P\ I_{k \times k}\right]$ where P is the $k \times (n-k)$ coefficient matrix with its elements equal to 0 or 1, and $I_{k \times k}$ an identity matrix. The parity-check matrix H is an $(n-k) \times n$ matrix that has the form:

$$H = \left[I_{(n-k) \times (n-k)}\ P^T\right]$$

Examples

Let `G` equal [1 1 1 1 1];

```
H = gen2par(G)
H =
    1  0  0  0  1
    0  1  0  0  1
    0  0  1  0  1
    0  0  0  1  1
G = gen2par(H)
G =
    1  1  1  1  1
```

See Also

encode, decode

Purpose Calculate the sum of two GF(p) polynomials or the sum of two GF(p^M) elements

Syntax

```
c = gfadd(a, b);
c = gfadd(a, b, p);
c = gfadd(a, b, p, len);
```

Description

`c = gfadd(a, b)` calculates the sum of two GF(2) polynomials. You must arrange the polynomials in ascending order, i.e, the input vector

$$a = \left[a_1\ a_2\ a_3\ \ldots\ a_{n-1}\ a_n\right]$$

represents the polynomial:

$$a(X) = a_1 + a_2X + a_3X^2 + \ldots.. + a_{n-1}X^{n-2} + a_nX^{n-1}$$

In the case of GF(2), all elements of `a` should be either 0 or 1.

`c = gfadd(a, b, p)` adds two polynomials `a` and `b` in GF(p) for a given prime number `p`. `p` can also be a matrix that lists all the possible elements of GF(p^M), where `p` is a prime number and `M` a positive integer greater than one. In this case the function takes the elements in `a` and `b` as powers of α in the exponential representation of GF(p^M). The output, a column vector c, is $\alpha^c = \alpha^a + \alpha^b$ in GF(p^M). You can generate a list of all elements in GF(p^M) by using the command:

```
p = gftuple([-1 : p^m-2]', m, q);
```

`c = gfadd(a, b, p, len)` adds two GF(p) polynomials `a` and `b`. The resulting GF(p) polynomial `c` has length equal to `len`. When `len` is a negative number, the length `c` defaults to `degree(c)+1`.

Examples

The addition between two GF(2) polynomials `a=[1 0 1 1]` and `b=[1 1 0]` is

```
c = gfadd(a, b)
c =
     1     1   0   1
```

The addition between two GF(2^4) vector `a = [-Inf 1 3 5]` and `b=[9 6 3 1]` is

```
p = gftuple([-1:2^4-2]',4);
c = gfadd(a,b,p);
c =
    9    11    -Inf    2
```

See Also

gfmul, gfdiv, gftuple, gfprimck

Purpose Galois field polynomial convolutional (multiplication)

Syntax

```
c = gfconv(a, b);
c = gfconv(a, b, p);
```

Description `c = gfconv(a, b)` multiplies two GF(2) polynomials `a` and `b`. The resulting binary polynomial `c` has degree of `degree(a) + degree(b)`.

`c = gfadd(a, b, p)` multiplies two polynomials `a` and `b` over GF(p). `p` must be a prime number.

The polynomials `a` and `b`, must be in ascending order, i.e, the input vector $a = \left[a_1\ a_2\ a_3\ \ldots\ a_{n-1}\ a_n\right]$ represents the polynomial:

$$a(X) = a_1 + a_2X + a_3X^2 + \ldots.. + a_{n-1}X^{n-2} + a_nX^{n-1}$$

a_i must be a non-negative number less than `p`. The output polynomial `c` is also in ascending order.

Examples The multiplication of two GF(2) polynomial `a` = [1 1 1] and `b` = [1 1] is

```
c = gfconv(a, b)
c =
     1     0     0     1
```

See Also `gfadd`, `gfmul`, `gfdiv`, `gfdeconv`

gfcosets

Purpose

Produce cyclotomic cosets mod(p^M-1) for a prime number p

Syntax

```
cs = gfcosets(m);
cs = gfcosets(m,p);
```

Description

`cs = gfcosets(m)` produces cyclotomic cosets mod(2^m–1) with p = 2.

`cs = gfcosets(m,p)` produces cyclotomic cosets mod(2^m–1) with prime p.

Each row of the output matrix `cs` is a coset. The first column is called the coset leader; refer to Lin & Costello for a discussion of this concept. Because the length of the coset may vary, an `NaN` is used to fill out the extra space to make all rows of `cs` have equal length.

Examples

Find the cosets for GF(2^4):

```
cs = gfcosets(4)
cs =

     0  NaN  NaN  NaN
     1    2    4    8
     3    6   12    9
     5   10  NaN  NaN
     7   14   13   11
```

See Also

`gfprimfd`, `gfroots`, `gfminpol`

Purpose

Galois field polynomial division

Syntax

```
[q, r] = gfdeconv(b, a);
[q, r] = gfdeconv(b, a, p);
```

Description

`[q, r] = gfdeconv(b, a)` divides `b` by `a` resulting in the quotient `q` and the remainder `r` in GF(2).

`[q, r] = gfdeconv(b, a, p)` divides `b` by `a`, the results of which are the quotient `q` and the remainder `r` in GF(p).

You must arrange the polynomials `a` and `b` in ascending order. The input vector $a = \left[a_1\ a_2\ a_3\ \dots\ a_{n-1}\ a_n\right]$ represents the polynomial

$$a(X) = a_1 + a_2X + a_3X^2 + \dots + a_{n-1}X^{n-2} + a_nX^{n-1}$$

a_i must be a non-negative number less than `p`.

Algorithm

This function uses the same algorithm as `deconv` in the Signal Processing Toolbox.

See Also

`gfconv`, `gfadd`, `gfmul`, `gfdiv`

gfdiv

Purpose

Galois field element-by-element division

Syntax

```
q = gfdiv(b,a);
q = gfdiv(b,a);
```

Description

`q = gfdiv(b,a)` divides `b` by `a`; the result is the quotient `q`, which is an element in GF(2). A zero element in `a` will result in a `NaN` solution.

`q = gfdiv(b,a,p)` divides `b` by `a`. The result is the quotient `q` in GF(p), where `p` is a prime number. `p` can also be a a list of all elements in GF(p^M), where `p` is prime and M is an integer greater than one. In this case, `q` is the result of the division operation in GF(p^M).

See Also

`gfadd`, `gfmul`, `gftuple`, `gfprimck`, `deconv`

Purpose Digital filtering over GF(2)

Syntax

```
y = gffilter(b, a, x);
y = gffilter(b, a, x, p);
```

Description

`y = gffilter(b, a, x)` filters the data `x` with the filter described by vectors `a` and `b` to create the filtered data `y` in GF(2).

`y = gffilter(b, a, x, p)` filters the data `x` with the filter described by vectors `a` and `b` to create the filtered data `y` in GF(`p`).

The filter solves a difference equation:

$$y(n) = b_{nb}x(n) + b_{nb-1}x(n-1) + \dots + b_1 x(n-nb-1) + a_{na-1}y(n-1) + \dots + a_1 y(n-na-1)$$

The polynomials a and b must be in ascending order, i.e, the input vector $a = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_{n-1} & a_n \end{bmatrix}$ represents the polynomial

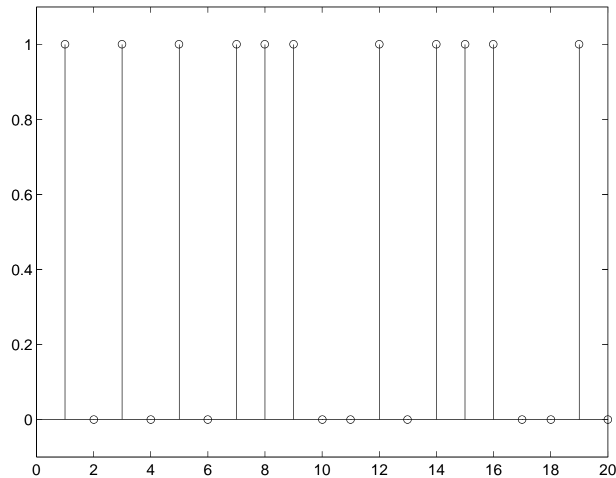
$$a(X) = a_1 + a_2 X + a_3 X^2 + \dots\dots + a_{n-1}X^{n-2} + a_n X^{n-1}$$

a_i must be a non-negative number less than `p`.

Examples

The impulse response of filter with `b = [1 0 0 1 0 1 0 1]`, `a = [1 0 1 1]`:

```
y = gffilter(b,a,[1,zeros(1,19)]);
stem(y);
axis([0,20,-.1 1.1])
```



Algorithm

This function uses an algorithm similar to function `filter` and applies to filters over GF(2) only. You can use `filter` for filters over GF(2) by using the format:

```
y = abs(rem(filter(b,a,x),2));
```

This may produce an error if `a` is not stable in the regular discrete-time system analysis and the vector `x` is too long, or for a high order filter. `gffilter` can produce an accurate result in all cases.

See Also

`gfadd`, `gfmul`, `gfdiv`, `gftuple`, `gfprimck`, `filter`

Purpose Solve the linear equation $A x = b$ over the GF(2) field

Syntax

```
x = gflineq(A,b);
```

Description `x = gflineq(A,b)` solves the linear equation $A x = b$ over GF(2).

Algorithm This function uses a Gaussian elimination method.

See Also `gfadd`, `gfdiv`, `gftuple`, `gfprimck`, `conv`

gfminpol

Purpose Find the minimal polynomial of a given element over the GF(2^m) field

Syntax

```
pl = gfminpol(k,m);
```

Description

`pl = gfminpol(k,m)` produces a minimal polynomial of the given elements `k` in the field GF(2^m), where `m` is a positive integer greater than or equal to one. Each element `i` of `k` represents α^{i-1}, where α is a generator element in the GF(2^m) field.

GF(2^m) is a field generated by the default degree `m` primitive polynomial, where `m` is a positive integer. You can view the default polynomial by using the command:

```
gfprimdf(m)
```

When `m` is a binary vector instead of a positive number, `m` is a primitive polynomial generated the field GF(2^{deg}), where `deg` is the degree of polynomial `m`(X).

The output `pl` is a row vector that represents the minimal polynomial. When `k` is a vector, the output `pl` is a matrix where the `i` th row represents the minimal polynomial associated with element `k(i)`.

Examples

A list of minimum polynomials of α^i, where i = 1, ..,2^m–1. In this case, we assume `m=4`:

```
disp('The default primitive polynomial')
gfpretty(gfprimdf(4))
m = 4;
pl = gfminpol([2:2^m]);
for i = 1:length(pl)
    disp(['element alpha^',num2str(i-1)])
    gfpretty(pl(i,:))
end
```

The commands output a list of minimum polynomials. This table lists the elements of GF(2^4) in exponential form and their associated polynomials:

Element of GF(2^4)	Minimal polynomial
Default primitive polynomial	$1+X+X^4$
α^1	$1+X+X^4$
α^2	$1+X+X^4$
α^3	$1+X+X^2+X^3+X^4$
α^4	$1+X+X^4$
α^5	$1+X+X^2$
α^6	$1+X+X^2+X^3+X^4$
α^7	$1+X^3+X^4$
α^8	$1+X+X^4$
α^9	$1+X+X^2+X^3+X^4$
α^{10}	$1+X+X^2$
α^{11}	$1+X^3+X^4$
α^{12}	$1+X+X^2+X^3+X^4$
α^{13}	$1+X^3+X^4$
α^{14}	$1+X^3+X^4$

See Also

`gfadd`, `gfdiv`, `gftuple`, `gfprimdf`

gfmul

Purpose

GF(p) polynomial multiplication and GF(p^M) element multiplication

Syntax

```
c = gfmul(a,b);
c = gfmul(a,b,p);
```

Description

`c = gfmul(a,b)` outputs the result of multiplying polynomial `a` with polynomial `b` in GF(2). The resulting binary polynomial `c` has degree equal to degree(a) + degree(b).

`c = gfmul(a,b,p)` multiplies two polynomials `a` and `b` over GF(p), where `p` is a prime number. When `p` is a matrix that contains all elements in GF(p^M), this function assumes `a` and `b` are powers of α in the exponential representation of GF(p^M). The output `c`, a column vector, is $\alpha^c = \alpha^a \alpha^b$ in GF(p^M).

You can produce a list of all elements in GF(p^M) by using the command:

```
p = gftuple([-1:p^M-2]',M,p)
```

The polynomials `a` and `b` must be in ascending order, i.e, the input vector $a = \begin{bmatrix} a_1 & a_2 & a_3 & \ldots & a_{n-1} & a_n \end{bmatrix}$ represents the polynomial

$$a(X) = a_1 + a_2X + a_3X^2 + \ldots.. + a_{n-1}X^{n-2} + a_nX^{n-1}$$

a_i must be a non-negative number less than `p`. The output `c` is also in ascending order.

See Also

`gfadd`, `gfdiv`, `gftuple`, `gfprimck`, `conv`

Purpose

Add two elements in GF(q^M)

Syntax

```
K = gfplus(i,j,alpha,beta);
```

Description

The elements in GF(q^M) are represented in exponential format. That is, all elements are of the form α^i, where i = –Inf, 0, 1, ..., q^M–2. In this toolbox, an element is represented by its power index, i.

`k = gfplus(i,j,alpha,beta)` outputs the additive result of two elements in GF(q^M), or $\alpha^k=\alpha^i+\alpha^j$. Variable `alpha` is a length q^M vector containing integers that are elements in GF(q^M). The vector `alpha` can be generated by using the function

```
alpha = gftuple([-1 : q^m-2]',m,q)*q.^[0 : M-1];
```

All elements in `alpha` are integers. `beta` is the index complement of `alpha`. That is, if `alpha` has index i, then `beta` has index q^M-2-i. You can use the command

```
beta(alpha + 1) = 0 : q^M - 1;
```

to obtain the variable `beta`. For most uses of `gfplus`, q = 2, which represents an operation on GF(2^M).

Variables `i` and `j` can be matrices, and they must have the same dimensions. This function is used in BCH and R-S decoding computations.

Example

This example adds two GF(2^5) matrices.

```
i = randint(100,10,2^5); % Create a 100-by-10 matrix in GF(2^5)
j = randint(100,10,2^5);
alpha = gftuple([-1 : 2^5 - 2]',5)*2.^[0 : M-1]';
beta(alpha + 1) = 0 : 2^5 - 1;
k = gfplus(i,j,alpha,beta); % Additive computation.
```

See Also

`gfplus`, `gfsub`, `gfmul`, `gfxor`

gfpretty

Purpose

Pretty print a GF(2) polynomial

Syntax

```
gfpretty(a);
gfpretty(a, st);
gfpretty(a, st, n);
```

Description

`gfpretty(a)` prints a polynomial representation over GF(2). The function uses the default variable X. The printed polynomial is in ascending order and the term with coefficient of zero is dropped.

`gfpretty(a, st)` uses the variable provided in the string variable `st`.

`gfpretty(a, st, n)` prints the polynomial using line width `n` instead of 79.

Examples

`gfpretty(rem(1:10, 2), 'Z')` outputs the following representation:

$$Z + Z^3 + Z^5 + Z^7 + Z^9$$

See Also

`gfadd`, `gfmul`, `gfdiv`, `gftuple`, `gfprimck`

Purpose

Check whether a GF(p) polynomial is irreducible and primitive

Syntax

```
ck = gfprimck(a);
ck = gfprimck(a, p);
```

Description

`ck = gfprimck(a)` checks the irreducible and primitive property of `a`, a GF(2) polynomial.

`ck = gfprimck(a, p)` checks the irreducible and primitive property of `a`, a GF(p) polynomial.

The polynomial `a` must be in ascending order, i.e, the input vector $a = \left[a_1\ a_2\ a_3\ \dots\ a_{n-1}\ a_n\right]$ represents the polynomial

$$a(X) = a_1 + a_2X + a_3X^2 + \dots.. + a_{n-1}X^{n-2} + a_nX^{n-1}$$

a_i must be a non-negative number less than `p`.

The function outputs a flag ck that categorizes the input polynomial:

`ck = -1` when `a` is not an irreducible polynomial.

`ck = 0` when `a` is irreducible but not a primitive polynomial.

`ck = 1` when `a` is a primitive polynomial.

Examples

`a = [1 0 1 1 1]` is not a irreducible polynomial in GF(2)

```
ck = gfmul(a)
ck =
    -1
```

`b = [1 1 1 1 1]` is a irreducible but not a primitive polynomial in GF(2)

```
ck = gfmul(b)
ck =
    0
```

`c = [1 0 0 1 1]` is a primitive polynomial in GF(2)

```
ck = gfmul(c)
ck =
    1
```

Algorithm

The irreducible property is verified by the theory of any irreducible m degree polynomial f(X) over GF(2) divides $1 + X^{2^m - 1}$.

An irreducible m degree polynomial f(X) over GF(2) is defined to be primitive if the smallest polynomial $1 + X^n$ is $n = 2^m + 1$

See Also

gfadd, gfmul, gfdiv, gftuple

Purpose

Provide default primitive polynomials

Syntax

```
pol = gfprimdf(m);
pol = gfprimdf(m,p);
```

Description

`pol = gfprimdf(m)` outputs the default primitive polynomial `pol` at the specified GF(2^m). The polynomial is in ascending order.

`pol = gfprimdf(m,p)` outputs the default primitive polynomial `pol` over the specified field GF(p^m). The polynomial is in ascending order.

Examples

List the GF(2^m) primitive polynomials for `m` from degree 3 to degree 10.

```
for i = 3 : 10
   gfpretty(gfprimdf(i))
end
```

$$1 + X + X^3$$

$$1 + X + X^4$$

$$1 + X^2 + X^5$$

$$1 + X + X^6$$

$$1 + X^3 + X^7$$

$$1 + X^2 + X^3 + X^4 + X^8$$

$$1 + X^4 + X^9$$

$$1 + X^3 + X^{10}$$

See Also

`gfprimfd`, `gfprimck`

gfprimfd

Purpose

Find GF(p^m) primitive polynomials

Syntax

```
pol = gfprimfd(m);
pol = gfprimfd(m,opt);
pol = gfprimfd(m,opt,p);
```

Description

`pol = gfprimfd(m)` finds one primitive polynomial for a given degree on GF(2^m). `m` is a positive integer.

`pol = gfprimfd(m,opt)` finds primitive polynomial(s) on GF(2^m) following the instructions provided in the variable `opt`.

`opt` can be an integer that indicates the number of terms of the primitive polynomial, i.e., the nonzero coefficients in the polynomial `p`.

`opt` can also be a string. When `opt = 'min'`, the function finds one primitive polynomial that has the minimal term in `pol`. When `opt = 'max'`, the function finds one primitive polynomial that has the maximal term in `pol`. When `opt = 'exh'`, the function searches to find all the primitive polynomials.

When the output is more than one polynomial, each row represents one primitive polynomial.

This function is useful for determining the Hamming code generator matrix and parity-check matrix.

`pol = gfprimfd(m,opt,p)` finds primitive polynomials in GF(p^m) following the instruction as given in `opt`.

This function outputs polynomials in ascending order form.

Examples

Find and display all primitive polynomials with degrees equal to 6.

```
x= gfprimfd(6,'exh');     % Find all primitive polynomials.
for i = 1 : size(x)
  gfpretty(x(i,:))      % Display all primitive polynomials
             % using a pretty form.
end
```

$1 + X + X^6$

$1 + X + X^3 + X^4 + X^6$

$1 + X + X^2 + X^5 + X^6$

$1 + X^2 + X^3 + X^5 + X^6$

$1 + X + X^4 + X^5 + X^6$

$1 + X^5 + X^6$

Find and display all three terms of primitive polynomials with degree equal to 7.

```
x= gfprimfd(7,3);       % Find all 3 terms primitive polynomials.
for i = 1 : size(x)
  gfpretty(x(i,:))     % Display all primitive polynomial using
            % a pretty form.
end
```

$1 + X + X^7$

$1 + X^3 + X^7$

$1 + X^4 + X^7$

$1 + X^6 + X^7$

Algorithm

The function does a search based on a test using `gfprimck`.

See Also

`gfadd`, `gfmul`, `gfdiv`, `gfprimck`

gfrank

Purpose Calculate the rank of a matrix in GF(p)

Syntax

```
rk = gfrank(a);
rk = gfrank(a, q);
```

Description `rk = gfrank(a)` calculates the rank of matrix `a` in GF(2).

`rk = gfrank(a, q)` calculates the rank of matrix `a` in GF(q), where `q` is a prime number.

This function uses an algorithm similar to the Gaussian elimination method.

See Also `gfline`

Purpose Convert between GF(2) polynomial representations

Syntax

```
p1 = gfrepcov(p2)
```

Description This function converts between two polynomial representations of GF(2). A GF(2) polynomial P is in ascending order with the form of

$$P(X) = p_0 + \dots + p_{n-1}X^{n-1} + p_nX^n$$

where p_i is either zero or one. The representation can be either

- A list all of the coefficients of *P(X)* (with its elements being either 0 or 1),
- Distinct positive integers. A nonzero element i indicates p_{i-1} in the above polynomial *P(X)*.

To eliminate ambiguity, in the second case, no element in P can be zero except when P is a scalar.
`p1 = gfrepcov(p2)` converts the second representation into the first one.

Examples Convert GF(2) polynomial $1 + X^2 + X^4$ from form 2 to form 1.

```
p2 = [1 3 5];
p1 = gfrepcov([1 3 5])
p1 =
    [1 0 1 0 1]
```

See Also `gfadd`, `gfmul`, `gfdiv`, `gftuple`

gfroots

Purpose

Find the roots of a binary polynomial in GF(p^m)

Syntax

```
rt = gfroots(f);
rt = gfroots(f, ord);
rt = gfroots(f, ord, p);
[rt, rt_tuple] = gfroots(...);
[rt, rt_tuple, alpha] = gfroots(...);
```

Description

`rt = gfroots(f)` finds all roots `rt` for a binary polynomial *f(X)* in the field GF(2^{ord}), where `ord` is the degree of the polynomial *f(X)*. The output `rt` is a column vector with its length less than or equal to the degree of *f(X)*. `rt = i` means that there is a root of α^{i-1}, where α is the element in GF(2^{ord}).

`rt = gfroots(f, ord)` finds all roots `rt` for binary polynomial f(X) in the field GF(2^{ord}). GF(2^{ord}) is a field generated by the degree `ord` primitive polynomial. `ord` is a positive integer that should be larger or equal to the degree of f(X). When `ord` is a binary vector instead of a positive number, this function assumes it represents a primitive polynomial that generates the field GF(2^{deg}), where `deg` is the degree of `ord(X)`. `deg` should be greater than the degree of f(X).

`rt = gfroots(f, ord, p)` finds all roots `rt` for binary polynomial f(X) on GF(2^{ord}).

`[rt, rt_tuple] = gfroots(...)` outputs the roots `rt` and the `m`-tuple of `rt`. Each row of `rt_tuple` represents the `ord`-tuple representation of `rt`.

`[rt, rt_tuple, alpha] = gfroots(...)` output the `ord`-tuple of α^i, where i=0, 1, ...2^{ord}–1.

`rt = [i, j, ... k]` means that the roots of f(X) are the elements α^i, α^j, ... α^k in GF(2^{ord}). The list of GF(p^{ord}) can be found by using

```
[tp, idx] = gftuple([-1 : p^ord-2], ord, p)
```

Limitation

The function only outputs the distinct roots of the polynomial f(X) in the field.

See Also

`gfprimdf`, `gfrepcov`, `hammgen`

Purpose Subtract two polynomials or elements in GF(p) or GF(p^M)

Syntax

```
c = gfsub(a, b);
c = gfsub(a, b, p);
c = gfsub(a, b, c, p, len);
```

Description

`c = gfsub(a, b)` is the same as `gfadd`, which calculates the sum of two GF(2) polynomials. The polynomials are in ascending order, i.e, the input vector $a = \begin{bmatrix} a_1 & a_2 & a_3 & \dots & a_{n-1} & a_n \end{bmatrix}$ represents the polynomial

$$a(X) = a_1 + a_2 X + a_3 X^2 + \dots + a_{n-1} X^{n-2} + a_n X^{n-1}$$

As defined in GF(2), all elements in `a` should be either 1 or 0.

`c = gfsub(a, b, p)` subtracts polynomials `b` from `b` in GF(p) for a given prime `p`. When `p` is a matrix that lists all elements in GF(p^M), this function takes the elements in `a` and `b` as powers of α in the exponential representation of GF(p^M). The output c, a column vector, is $\alpha^c = \alpha^a - \alpha^b$ in GF(p^M).

You can generate a list of all elements in GF(p^M) by using:

```
p = gftuple([-1 : p^m-2]', m, q)
```

`c = gfsub(a, b, p, len)` subtracts GF(p) polynomial `b` from `a`. The resulting GF(p) polynomial `c` keeps the given length `len`. When `len` is a negative number, the length `c` equals degree(c) + 1.

See Also `gfadd`, `gfmul`, `gfdiv`, `gfconv`, `gfdeconv`

gftrunc

Purpose Remove meaningless terms from a polynomial over GF(2)

Syntax

```
c = gftrunc(a);
```

Description `c = gftrunc(a)` removes all high order terms that have zero coefficients and outputs a shortened form of the polynomial. The polynomial `a` must be in ascending order, i.e, the input vector a = [a_1 a_2 a_3 ... a_{n-1} a_n] represents the polynomial:

$$a(X) = a_1 + a_2X + a_3X^2 + ... + a_{n-1}X^{n-2} + a_nX^{n-1}$$

where a_i is either 0 or 1.

When $a_{i+1}, a_{i+2}, ..., a_n$ are zeros and a_i is nonzero, `gftrunc` returns

$$c = \left[a_1\ a_2\ ...\ a_{i-1}\ a_i\right]$$

See Also `gfadd`, `gfmul`, `gfdiv`, `gftuple`, `gfprimck`

Purpose

Calculate the m-tuple expression of a polynomial over GF(p)

Syntax

```
tp = gftuple(a, ord);
tp = gftuple(a, ord, p);
tp = gftuple(a, ord, p, prim_ck);
[tp, indx] = gftuple(...);
```

Description

`tp = gftuple(a, ord)` converts GF(2) polynomial `a` to the `ord`-tuple representation `tp` in GF(2^{ord}). *A* is a matrix with each row representing in ascending order polynomial with *A(i, :)* = [a_1, a_2, ... a_n] representing

$$A_i(X) = a_1 + a_2X + a_3X^2 + ... + a_{n-1}X^{n-2} + a_nX^{n-1}$$

When a is a column vector or a scalar, each element represents a power of α in the exponential representation of the finite field.

The primitive polynomial can be found by using the commands `gfprimdf` and `gfprimfd`. When `ord` is a binary vector instead of a positive number, `ord` is considered to be a primitive polynomial generating the field GF(2^{ord}), where `deg` is the degree of `ord(X)`.

`tp = gftuple(a, ord, p)` generates the m-tuple representation in GF(p).

`tp = gftuple(a, ord, p, prim_ck)` is called only if `ord` is a polynomial. This format checks whether `ord` is a primitive polynomial before further computation.

`[tp, indx] = gftuple(...)` outputs the index form of the tuple in `indx`.

Note: When `a` is a vector, make certain whether it is a row or column vector. A row vector represents a polynomial. A column vector denotes powers of α in the exponential representation.

In the index representation, any negative number is equivalent to –Inf.

Example

This example prints the four-tuple representation table generated by the primitive polynomial $p(X) = 1 + X + X^4$:

```
for i=-1:2^4-2
  [tp,indx] = gftuple(i,[1 1 0 0 1]);
  st = num2str(indx);
  if i==0,
   disp('Index number | 4-Tuple representation');
  end;
  disp([st,setstr(ones(1,12-length(st))*32),' |
',  mat2str(tp))
end
```

Algorithm

This function uses recursive callbacks in determining the index.

See Also

gfadd, gfmul, gfdiv, gfprimck

Purpose Calculate the Hamming weight of a block code based on a generator matrix or a parity check matrix

Syntax

```
wt = gfweight(g);
wt = gfweight(g, gh_flag);
```

Description The Hamming weight of a block code is defined as the minimum Hamming distance between any two codewords. The Hamming distance is defined as the number of elements that differ between two codewords. For example, [1 0 0] and [1 0 1] have a Hamming distance equal to 1.

`wt = gfweight(g)` outputs the Hamming weight of the given generator matrix `g`.

`wt = gfweight(g, gh_flag)` outputs the Hamming weight based on the given generator matrix, parity check matrix, or generator polynomial. When `gh_flag='gen'`, `g` is a generator matrix. When `gh_flag='par'`, `g` is a parity-check matrix. When `gh_flag=n`, where `n` is a positive integer in specifying the order of the generator polynomial, `g` is a generator polynomial.

See Also `Hammgen`, `bchpoly`

hammgen

Purpose

Produce a parity-check matrix and a generator matrix for Hamming code

Syntax

```
h = hammgen(m);
h = hammgen(m,p);
[h,g] = hammgen(...);
[h,g,n,k] = hammgen(...);
```

Description

Hamming code is a class of perfect codes. Given a positive integer m≤3, the parameters of the Hamming code are provided in this table:.

Parameters	Number
number of information bits	$n = 2^m - 1$
Code length	$k = 2^m - m - 1$
Parity-check length	$m = n - k$
Error-correcting capability	t =1 (minimum distance is 3)

`h = hammgen(m)` produces m-by-n parity-check matrix of Hamming code provided a given positive integer m. The function gfprimf generates the GF(2) primitive polynomial used in producing the Hamming code.

`h = hammgen(m,p)` produces a parity-check matrix of Hamming code provided a given positive integer m and an mth degree GF(2) primitive polynomial p. p is in ascending order so that $p = \left[p_1\ p_2\ \dots\ p_{n-1}\ p_n\right]$ represents,

$$p(X) = p_1 + p_2X + \dots\dots + p_{n-1}X^{n-2} + p_nX^{n-1}$$

`[h,g] = hammgen(...)` produces the k-by-n generator matrix, g, for Hamming code as well as the parity-check matrix, h.

`[h,g,n,k] = hammgen(...)` produces the parity-check matrix, h, generator matrix, g, and the dimension information n and k.

Examples

You can easily produce the parity-check matrix and generator matrix for any `m` greater than or equal to 3. For example, the parity-check matrix, `h` and generator matrix for `m=3` are:

```
[h, g] = hammgen(3)
h =
    1  0  0  1  0  1  1
    0  1  0  1  1  1  0
    0  0  1  0  1  1  1
g =
    1  1  0  1  0  0  0
    0  1  1  0  1  0  0
    1  1  1  0  0  1  0
    1  0  1  0  0  0  1
```

Algorithm

Unlike `gftuple` which processes one m-tuple at a time, `hammgen` generates the entire sequence from 0 to 2^m-1. The computation algorithm uses all previously computed values to produce the computation result.

See Also

`gftuple`, `gfrepcov`, `gfprimck`

hank2sys

Purpose

Generate a discrete-time system model from a Hankel matrix

Syntax

```
[num,den] = hank2sys(hankel,ini,tol);
[num,den,sv] = hank2sys(...);
[a,b,c,d] = hank2sys(...);
[a,b,c,d,sv] = hank2sys(...);
```

Description

`[num,den] = hank2sys(hankel,ini,tol)` generates a single input/single output system transfer function from the Hankel matrix `hankel`. The system impulse response at time `ini` is based on the tolerance `tol`. The input Hankel matrix must be a square matrix.`ini` and `tol` are scalar values. Parameters `ini` and `tol` can be omitted, their default values are zero 0.01respectively. When `tol`>1, `ceil(tol)` is the order of the output system. When tol<1, it is the tolerance value used in selecting the singular value to determine the order of the system.`[num,den]` are vectors that represent the numerator and denominator of the system model transfer function.

`[a,b,c,d] = hank2sys(...)` outputs the state-space representation of a linear system. `[a,b,c,d]` are matrices.

`[a,b,c,d,sv] = hank2sys(...)` includes a vector `sv` that contains the singular values of the Hankel matrix.

Algorithm

This function uses singular value decomposition.

See Also

`imp2sys`

Purpose Design an analog Hilbert filter

Syntax

```
hilbaf;
hilbaf(Fc);
hilbaf(Fc,Ts);
hilbaf(Fc,Ts,Tol);
[num,den] = hilbaf(...);
[num,den,dly] = hilbaf(...);
[a,b,c,d] = hilbaf(...);
[a,b,c,d,sv] = hilbaf(...);
```

Description

An ideal Hilbert filter is a non causal filter that has a transfer function $H(s) = -j\,\mathrm{sgn}(s)$. Its impulse response is $h(t) = 1/t/\pi$. This implementation of the Hilbert filter requires the addition of a time delay, which must be greater than 0. This function uses a singular value decomposition method to design the filter, and it uses a residue method to convert a discrete filter to an analog filter.

There is a discontinuity in the impulse response of the filter. The output of the analog Hilbert filter depends on the input signal frequency, the computation time interval, and the time delay. See the function `hilbana` for a discussion of the compensator design for a Hilbert filter.

`hilbaf` designs a fourth order analog Hilbert filter for a 1Hertz input signal. The computation time interval defaults to 1/90 seconds and the delay is set to 7/180 seconds. This command plots the filter output for a 1 Hertz input signal.

`hilbaf(Fc)` designs a fourth order analog Hilbert filter for a `Fc` Hertz input signal. The function sets the computation time interval to `1/Fc/90` and the delay to `7/Fc/180` seconds. This command plots the filter output for a 1 Hertz input signal.

`hilbaf(Fc,Ts)` designs an analog Hilbert filter for a `Fc` Hertz input signal. The computation time interval is `Ts` and the delay equals `(ceil(7/180/Fc/Ts) + 0.5)*Ts`. The function determines the order of the filter using the calculation tolerance `tol` = 0.05. This command plots the filter output for a 1 Hertz input signal.

`hilbaf(Fc,Ts,Tol)` uses a specified tolerance. When `tol` is in the range (0, 1), the order of the filter is determined by

$$\frac{\text{truncated singular value}}{\text{maximum singular value}} \leq \text{tol}$$

`[num,den] = hilbaf(...)` outputs the numerator, `num`, and denominator, `den`, of the analog Hilbert filter.

`[num,den,dly] = hilbaf(...)` outputs the filter time delay, `dly`.

`[a,b,c,d] = hilbaf(...)` outputs the state-space representation of the filter.

`[a,b,c,d,sv] = hilbaf(...)` outputs the singular values generated in the computation.

See Also

`hilbiir`, `hilbana`

Purpose Hilbert transform filter design

Syntax

```
hilbiir;
hilbiir(st);
hilbiir(st, dly);
hilbiir(st, dly, tol);
[num, den] = hilbiir(...);
[num, den, sv] = hilbiir(...);
[a, b, c, d] =hilbiir(...);
[a, b, c, d, sv] = hilbiir(...);
```

Description Function `hilbiir` designs a Hilbert transform filter. An ideal Hilbert transform filter has the transfer function as $H(s) = -j\mathrm{sgn}(s)$, where $j = \sqrt{-1}$ and sign(.) is the sign function. The impulse response of the Hilbert transform filter is $h(t) = \frac{1}{\pi t}$. Since the Hilbert transform filter is a noncausal filter, this function introduces a time delay, `dly`. A Hilbert transform filter with this delay has an impulse response $h(t) = \frac{1}{\pi(t - dly)}$.

`hilbiir` designs a fourth order digital Hilbert transform filter with a one second delay. The sample time is 2/7. In this particular design, the tolerance index is 0.05. This format produces a plot of the impulse response of the filter and, for comparison, the impulse response of the ideal Hilbert transform filter with a one second delay.

`hilbiir(st)` designs a Hilbert transform filter with a sample time of `st` seconds and a one second delay. The default value for `tol` = 0.05, which determines the order of the filter. This format produces a plot of the impulse response of the filter and, for comparison, the impulse response of the ideal Hilbert transform filter with a one second delay.

`hilbiir(st, dly)` designs a Hilber transform filter with a sample time of `st` seconds and a time delay of `dly`. The default value for `tol` = 0.05, which determines the order of the filter. This format produces a plot of the impulse response of the filter and a comparison of the ideal Hilbert transform filter with a `dly` second delay.

`hilbiir(st, dly, tol)` designs a Hilber transform filter with a sample time of `st` seconds and a time delay of `dly`. When `tol` < 1, the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < tol$$

When `tol` >1, the order of the filter is `tol`.This format produces a plot of the impulse response of the filter and a comparison of the ideal Hilbert transform filter with `dly` second delay.

`[num, den] = hilbiir(...)` outputs the numerator and denominator of the IIR filter's transfer function.

`[num, den, sv] = hilbiir(...)` outputs the numerator and denominator of the IIR filter's transfer function and the singular values of the Hankel matrix used in the computation.

`[a, b, c, d] = hilbiir(...)` outputs the discrete-time state-space model of the designed Hilbert transform filter. `[a, b, c, d]` are matrices.

`[a, b, c, d, sv] = hilbiir(...)` outputs the discrete-time state-space model of the designed Hilbert transform filter and the singular-values of the Hankel matrix used in the computation.

Since the filter design is an approximation, to get an accurate result `dly` should be at least a few times larger than `st`. At the point `t = dly`, the impulse response of the Hilbert transform filter can be interpreted as 0, –Inf, or Inf. In the function, this point is set to be zero, if the point is encountered. To get an accurate result, the point should be avoid. A suggested related between `dly` and `st` is `rem(dly, st) = st/2`. For example, when `dly` is fixed, `st` could be chosen as `2*dly/N`, where `N` is an positive integer.

Algorithm

`hilbiir` calculates the impulse response of the ideal Hilbert transform filter response with delay. The response curve is fitted by using the SVD method introduced in S.Y.Kung's paper and refined in W.Wang's paper.

Examples

At the MATLAB prompt, type `hilbiir` or `[num, den]=hilbiir` for a default value example.

htruthtb

Purpose Generate a single error-correction logic matrix

Syntax

```
trt = htruthtb(g);
```

Description This function generates a single error-correction logic matrix. It is based on the Hamming code truth table generator.

`trt = htruthtb(g)` outputs a truth table for single error-correction code, especially Hamming code. The input of this block is a N-by-K matrix, which indicates the codeword length is N and the number of information bits is K. The output of this function is an N-by-N matrix. The correction vector is `trt(syndrome,:)`, where `syndrome` is represented in a decimal form.

You can calculate the `syndrome` by typing:

```
syndrome = received_vector*transpose_of_parity_check_matrix
```

See Also `decode`, `hammgen`

imp2sys

Purpose

Generate a discrete-time system from an impulse response sequence

Syntax

```
[num,den] = imp2sys(impulse,tol,ord);
[num,den,sv] = imp2sys(...);
[a,b,c,d] = imp2sys(...);
[a,b,c,d,sv] = imp2sys(...);
```

Description

`[num,den] = imp2sys(impulse,tol,ord)` generates a single input/single output linear system model from its impulse response sequence `impulse` based on the specified tolerance `tol`. When `tol`>1, `ceil(tol)` is the order of the output system. When `tol`<1, `tol` is the tolerance value in selecting the singular value to determine the order of the system. The default value of `tol` is 0.01. This function constructs a Hankel matrix for use in generating the discrete-time realization. The size of the Hankel matrix can be specified by variable `ord`; the default size is:

`[num,den]` contains the numerator and denominator of the system model transfer function.

`[a,b,c,d] = imp2sys(...)` outputs the state-space representation of a linear system. `[a,b,c,d]` are matrices.

`[a,b,c,d,sv] = imp2sys(...)` outputs a vector `sv` that contains the singular value of the Hankel matrix.

Examples

You can use this code to compare the impulse response of a given model to the output of the `imp2sys`:

```
[a,b,c,d] = drmodel(5,1,1);
imp = dimpulse(a,b,c,d,1,100);
[a1,b1,c1,d1] = imp2sys(imp,5);
imp1 = dimpulse(a1,b1,c1,d1,1,100);
plot([imp,imp1]);
```

The impulse response of the two systems should match closely if the original system is stable.

See Also

hank2sys

Purpose

Verify whether a number is prime

Syntax

```
isp = isprime(x);
```

Description

`isp = isprime(x)` returns 1 for the elements of `x` which are primes, 0 otherwise.

Examples

Find all prime numbers less than or equal to 100:

```
x = [1:2:100];
primes_numbers = x(find(isprime(x)==1))
prime_numbers =
```

3	5	7	11	13	17	19	23	29	31	37	41
43	47	53	59	61	67	71	73	79	83	89	97

See Also

`primes`

lloyds

Purpose

The Lloyd algorithm for scalar quantization parameter optimization using training data

Syntax

```
[partition, codebook] = lloyds(training_set, ini_codebook);
[partition, codebook]
    = lloyds(training_set, ini_codebook, tol);
[partition, codebook]
    =lloyds(training_set, ini_codebook, tol, p_flag);
[partition, codebook, distortion] = lloyds(...);
[partition, codebook, distortion, rel_distor] = lloyds(...);
```

Description

`[partition, codebook] = lloyds(training_set, ini_codebook)` optimizes the scalar quantization parameters `partition` and `codebook` using the vector `training_set`. `ini_codebook` is the initial guess of the `codebook` values. A `codebook` length should be greater than or equal to 2. The output `codebook` has the same length as `ini_codebook`. The output `partition` has its vector length equal to the vector length of `codebook` minus 1. When `ini_codebook` is a scalar integer, the function uses it as the desired `codebook` size. The optimization processing ends when the relative distortion value is less than 10^{-7}

`[partition, codebook] = lloyds(training_set, ini_codebook, tol)` uses the user-specified tolerance for the optimization termination criterion.

`[partition, codebook] = lloyds(training_set, ini_codebook, tol, p_flag)` plots the original signal, the optimized partition, and codebook in a figure.

`[partition, codebook, distortion] = lloyds(...)` outputs the final distortion value in the optimization.

`[partition, codebook, distortion, rel_distor] = lloyds(...)` outputs the final relative distortion values in terminating the computation.

`training_set` should be the typical data for the signal to be processed. The output `partition` and `codebook` from this function are optimized for the training set only.

Examples

Train the quantization codebook and partition for a 3-bit transfer channel for a sinusoidal transmission.

```
x = sin([0:1000]*pi/500); % Generate a complete period of
              sinusoidal signal.
[partition,codebook] = lloyds(x,2^3)
partition =
-0.8118 -0.5502 -0.2727 0.0039 0.2781 0.5516 0.8118
codebook =
-0.9361 -0.5876 -0.4127 -0.1327 0.1405 0.4157 0.6876 0.9361
```

See Also

compand, dpcmopt, quantiz

modmap

Purpose

Map a digital signal to an analog signal prior to modulation

Syntax

```
modmap(method);
modmap(method,...);
y = modmap(x, Fd, Fs, method,...);
```

Description

A digital modulation can be broken into two stages: digital to analog mapping and analog signal modulation. Function `modmap` maps a digital signal to an analog signal, so that the analog signal is ready for modulation. You can specify the type of modulation you want to use by specifying the string variable `method`. The output variable, `y`, is a signal ready for analog modulation. The size of variable `y` varies depending on which modulation method you choose. The table below lists the types of modulation available:

Method	Digital Modulation Scheme	Output Size	Modulation Method
'ask'	M-ary amplitude shift-keying modulation	m-by-n	'am'
'fsk'	M-ary frequency shift-keying modulation	m-by-n	'fm'
'msk'	Minimum shift-keying modulation	m-by-n	'fm'
'psk'	M-ary phase shift-keying modulation	m-by-2n	'qam'
'qask'	Quadrature amplitude shift-keying modulation	m-by-2n	'qam'
'sample'	Sampling `Fd` frequency signal to `Fs` frequency signal	m-by-n	open

In the table,

```
m = row_number_of_x*Fs/Fd;
n = column_number_of_x;
```

where x is a vector, row_number_of_x equals the vector length of x, and the column_number_of_x equals 1. In the cases where the output size equals m-by-2n, the odd numbered columns contain the in-phase component and the even numbered columns contain the quadrature component. You can modulate the output of this function by using either amod (with a *method* listed on the right hand column of the above table) or dmod (with '/nomap' appended to the *method* string). For example:

```
x = randint(100,1,4);
y = modmap(x,1,10,'qask',4);
z1 = amod(y,3,10,'qam');
z2 = dmod(y,3,1,10,'qask/nomap',4);
```

In this example z1 is exactly the same as z2. Both z1 and z2 equal z3 as given by:

```
z3 = dmod(x,3,1,10,'qask',4);
```

modmap(*method*) provides more detailed help on a specific modulation method. The variable *method* is a string chosen from the above *method* listing.

modmap(*method*,...) plots a constellation of the given method with specified parameters.

y = modmap(x,Fd,Fs,*method*,...) modulates the digital message signal x with carrier frequency Fc (Hertz) and simulation sample frequency Fs (Hertz). The output signal y has sampling frequency Fs, and the input signal x has sampling frequency Fd. Fs/Fd must be a positive integer. Input x can be a vector or a matrix. When it is a matrix, each column of x is processed independently. The row number of the output matrix y is:

```
Fs/Fd*row_number_of_x
```

The time interval between two consecutive points in y is 1/Fs while the time interval between two consecutive points in x is 1/Fd. The variable Fs can be either a scalar or a two-element vector. The first element is the sample frequency; the second element, if present, is the initial phase in the carrier signal modulation. The initial phase u is in radians; the default initial phase is zero.

method = 'ask'

modmap('ask',M) plots the M-ary ASK constellation.

`y = modmap(x, Fd, Fs, 'ask')` maps the input signal `x` using the M-ary amplitude shift-keying method. All elements in `x` are non-negative integers. The input integer is in the range `[0, M-1]`. `M = 2`^{K}, where `K` is the minimum integer that satisfies the inequality `max(x)<2`^{K}.

`y = modmap(x, Fd, Fs, 'ask', M)` specifies the M-ary number `M`.

method = 'fsk'

`modmap('fsk', M)` plots the FSK constellation.

`y = modmap(x, Fd, Fs, 'fsk')` maps the input signal `x` using the frequency shift-keying method. All elements in `x` are non-negative integers. The range of the input integers is `[0, M-1]`, where `M = 2`^{K}. `K` is the minimum integer satisfying the inequality `max(x)<2`^{K}. The tone space, i.e., the frequency separation between successive frequencies, defaults to `tone = 2*Fd/M`.

`y = modmap(x, Fd, Fs, 'fsk', M, tone)` specifies the input digital signal range `M` and the tone space `tone`.

method = 'msk'

`modmap('msk', M)` plots the MSK constellation.

`y = modmap(x, Fd, Fs, 'msk')` maps the input signal `x` using the minimum shift-keying method. All elements in `x` must be binary numbers. The tone space defaults to `tone = Fd`.

method = 'psk'

`modmap('psk', M)` plots the M-ary PSK constellation.

`y = modmap(x, Fd, Fs, 'psk')` maps the input signal `x` using the phase shift-keying method. All elements in `x` are non-negative integers. The range of the input integers is `[0, M-1]`, where `M = 2`^{K}. `K` is the minimum integer that satisfies the inequality `max(x)<2`^{K,}.

`y = modmap(x, Fd, Fs, 'psk', M)` specifies the input M-ary number M.

method = 'qask'

This function supports three constellations: square, circle, and arbitrary constellations.

The square constellation has a square shape, which is determined by a M-ary number M. Usually, M=2^K, where K is a positive integer. The maximum in-phase and quadrature values in the constellation and the output variable y are:

M=2^K	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

A circle constellation is defined by using three equal length vectors. These three vectors are:

- noc (Number of symbols on each circle),
- roc (Radii on each circle), and
- poc (Phase shift on each circle, in radian).

The M-ary number of this block equals the summation of all elements in noc. Let noc be [N_1, N_2,..., N_K]; roc be [R_1, R_2,..., R_N]; and poc be [P_1, P_2,..., P_K]. Then the i th circle has N_i elements, i = 1, ..., N. The N_i points are evenly distributed on the i th circle with their radii equal R_i. One of the points on the i th circle has phase P_i. In another words, the N_i points on the circle are located at the positions:

$$\left[R_i e^{jP_i} \quad R_i e^{j\left(P_i + \frac{2\pi}{N_i}\right)} \quad \ldots \quad R_i e^{j\left(P_i + \frac{2\pi(N_i - 1)}{N_i}\right)} \right]$$

The elements in the vectors noc and roc must be positive. For a correct demapping, there should be no repeating points.

You can define an arbitrary constellation by using two equal length vectors. The two vectors specify the in-phase component, in_phase, and the quadrature component, quad. The M-ary number of the mapping equals the length of the vectors. The in-phase value and the quadrature value of symbol *i*, *i = 0, 1, . . . , M-1*, are defined in the (i+1)th element in the in_phase vector and in the quad vector respectively.

modmap('qask', M) plots the square QASK constellation.

modmap('qask/arb', in_phase, quad) plots the user-defined arbitrary QASK constellation.

modmap('qask/circle', noc, roc, poc) plots the circle constellation.

y = modmap(x, Fd, Fs, 'qask') maps the input signal x using the square constellation QASK method. All elements in x are non-negative integers. The range of the input integers is [0, M-1], $M = 2^K$. K is the minimum integer satisfying the inequality $max(x) < 2^K$.

y = modmap(x, Fd, Fs, 'qask', M) specifies the M-ary number of the square constellation QASK.

y = modmap(x, Fd, Fs, 'qask/arb', in_phase, quad) maps the input signal x using the arbitrary constellation QASK method with the in-phase component and quadrature component values provided in the variable in_phase and quad.

y = modmap(x, Fd, Fs, 'qask/circle', noc, roc, poc) maps the input signal x using the circle constellation QASK method with the number of symbols on each circle, radii for each circle, and phase shift for each circle specified in variables noc, roc, and poc respectively.

method = 'sample'

y = modmap(x, Fd, Fs, 'sample') converts sampling rate Fd input signal x to sampling rate Fs output signal y, where Fs>Fd and Fs/Fd is a positive integer.

See Also

amod, ademod, dmod, ddemod, amodce, ademodce, demodmap

Purpose

Convert between octal and binary forms of a convolutional code transfer function matrix

Syntax

```
[tran_func, code_param] = oct2gen(gen);
gen = oct2gen(tran_func, code_param);
```

Description

Assume the convolutional code is a `(N, K, M)` code, which means that the word length is `N`, number of information bits is `K`, and the maximum degree among all transfer functions is `M`. We use a three element vector `code_param` to represent the code parameter, i.e., `code_param=[N, K, M]`.

The binary transfer function matrix `trans_func` is a `K`-by-`N*(M+1)` binary matrix. `tran_func` can be written as the following form:

$$\text{tran_func} = \begin{bmatrix} t_{11} & t_{12} & \dots & t_{1N} \\ \dots & & \dots & \\ t_{K1} & t_{K2} & \dots & t_{KN} \end{bmatrix}$$

where t_{ij} is a length M+1 binary row vector, which is the transfer function from `i` th input to `j` th output. The row vector t_{ij} represents an ascending ordered polynomial. For example the (2,1,4) convolutional code

$$\begin{bmatrix} 1 + X^3 + X^4 & 1 + X + X^2 + X^4 \end{bmatrix}$$

can be represented by the `code_param=[2, 1, 4]` and

$$\text{tran_func} = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}$$

The octal form of the transfer function matrix `gen` has the form of

$$\text{gen} = \begin{bmatrix} g_{11} & g_{12} & \dots & g_{1N} \\ \dots & & \dots & \\ g_{K1} & g_{K2} & \dots & g_{KN} \end{bmatrix}$$

in which g_{ij} is the `i` th input `j` th output octal form transfer function. Note that the binary form of the transfer function is in ascending order, the octal form is also a conversion of the binary vector along the ascending direction. Taking the

same example as we have used above, the octal form of the transfer function gen is:

$$\text{gen} = \begin{bmatrix}46 & 72\end{bmatrix}$$

`[tran_func, code_param] = oct2gen(gen)` converts an octal form of the transfer function matrix `gen` into a binary form of the transfer function `tran_func`. The function converts `gen` into the convolutional code parameter `code_param = [N, K, M]` and the convolutional code transfer function `tran_func`.

`gen = oct2gen(tran_func, code_param)` converts a binary form of the transfer function matrix `tran_func` and `code_param` into a octal form of transfer function matrix `gen`.

Examples

Take the example of converting the octal form of transfer function in the example in `sim2gen` to binary form and transfer it back to the octal form.

```
gen = sim2gen('fig_10_9');
[tran_func, code_param] = oct2gen(gen)
tran_func =

          1 0 0 0 0 0 1 1 1
          0 0 0 1 0 0 1 0 1

code_param =


            3 2 2


gen2 = oct2gen(tran_func, code_param)
gen2 =

       4 0 7
       0 4 5
```

See Also

`sim2gen`

Purpose Generate a list of prime numbers

Syntax

```
p = primes(n);
```

Description `p = primes(n)` generates a row vector `p`, which contains the prime numbers less than or equal to `n`.

See Also `isprime`

qaskdeco

Purpose

QASK square constellation demapping computation

Syntax

```
msg = qaskdeco(x, y, m);
msg = qaskdeco(x, y, m, minmax);
```

Description

`msg = qaskdeco(x, y, m)` demaps the message signal `msg` from the in-phase component value specified in `x` and quadrature component value specified in `y`. The multiple value `m` must be a positive integer number that is the power of 2. The function uses the default minimum and maximum values of the in-phase component and quadrature component, which are provided in the table in function `qaskenco`.

`msg = qaskdeco(x, y, m, minmax)` specifies the in-phase and quadrature component minimum and maximum values in variable `minmax`. `minmax` is a 2-by-2 matrix

$$\text{minmax} = \begin{bmatrix} \text{in_phase_min} & \text{in_phase_max} \\ \text{quadrature_min} & \text{quadrature_max} \end{bmatrix}$$

See Also

`qaskenco`, `decode`, `demodmap`

Purpose QASK square constellation mapping computation

Syntax

```
qaskenco(m)
qaskenco(msg, m)
[x, y] = qaskenco(m)
[x, y] = qaskenco(msg, m)
```

Description `qaskenco(m)` plots the square constellation for QASK with given M-ary numbers in `m`. `m` must be power of `2`.

`qaskenco(msg, m)` plots the location of the digital signal `msg` with the M-ary number given in `m`. The elements in `msg` must be integers in the range `[0, m–1]`.

`[x, y] = qaskenco(m)` outputs the square constellation points for `0: m–1`. The in-phase values are provided in `x` and the quadrature points are provided in `y`.

`[x, y] = qaskenco(msg, m)` maps the message signal `msg` into in-phase component in `x` and quadrature component in `y`. `m` is the M-ary number and the elements in `msg` must be integers in the range `[0, m–1]`.

The output maximum numbers are:

$M=2^K$	In-phase	Quadrature
2	1	1
4	1	1
8	3	1
16	3	3
32	5	5
64	7	7
128	11	11
256	15	15

See Also encode, modmap, qaskdecode

quantiz

Purpose

Produce a quantization index and a quantized output value

Syntax

```
indx = quantiz(sig, partition);
[indx, quant] = quantiz(sig, partition, codebook);
[indx, quant, distor] = quantiz(sig, partition, codebook);
```

Description

This function performs quantization using the parameter `partition`. The vector `partition` must be a strictly ascending ordered real vector. Assuming the vector length of `partition` is N–1, the value of `index` is one of the `N` integers in the range [`0`, `N–1`]. The output `index` equals:

- 0 if the signal `sig` is less than or equal to `partition(1)`,
- i if the signal `sig` is larger than `partition(i)` but less than or equal to `partition(i+1)`, and
- N–1 if the signal `sig` is larger than `partition(N–1)`.

`codebook` holds the values assigned to each partition in the quantization; the value for `indx=i–1` is in `codebook(i)`. For a length `N–1` `partition`, the `codebook` length must be `N`.

`indx = quantiz(sig, partition)` produces a quantized index `indx`. `indx` has the same length as the signal vector `sig`.

`[indx, quant] = quantiz(sig, partition, codebook)` produces the quantized value in `quant`.

`[indx, quant, distor] = quantiz(sig, partition, codebook)` outputs `distor`, the estimated distortion value of this quantization data set.

Note: There is no dedicated function for the decode process for `quantiz` because the decoding is as simple as:

```
quant = codebook(indx+1);
```

See Also

`lloyds`, `dpcmenco`, `dpcmdeco`

Purpose Randomly add ones into a zero vector

Syntax

```
out = randbit(n, m);
out = randbit(n, m, prob);
out = randbit(n, m, prob, seed);
```

Description `out = randbit(n, m)` generates an n-by-m zero matrix with a single 1 placed in random positions in each row.

`out = randbit(n, m, prob)` generates an `n`-by-`m` zero matrix with ones placed into random positions. The probability of a single 1 in a row is specified in the first element of the vector `prob`. The probability of double 1s in a row is specified in the second element of the vector prob, and so on. The vector length of `prob` cannot be larger than `m`, the column size of the zero matrix. The element value in `prob` must be in the range [0, 1], `sum(prob)` must be less than or equal to one.

`out = randbit(n, m, prob, seed)` specifies the `seed` in the random number generation.

See Also `rand`, `randint`

randint

Purpose Uniformly distributed random integer and matrix generator

Syntax

```
out = randint(n);
out = randint(n, m);
out = randint(n, m, range);
out = randint(n, m, range, seed);
```

Description

`out = randint(n)` generates an `n`-by-`n` uniformly distributed random binary matrix.

`out = randint(n, m)` generates an `n`-by-`m` uniformly distributed random binary matrix.

`out = randint(n, m, range)` generates an `n`-by-`m` uniformly distributed integer matrix with the minimum and maximum number specified in the length 2 vector `range`. When the input variable `range` is a positive integer, the `range` is automatically set as `[0, range–1]`.

`out = randint(n, m, range, seed)` generates random numbers using the given `seed`.

Examples To generate an evenly distributed 10-by-10 matrix with the elements of the matrix in the range from 0 to 7, you can use either one of the following commands.

```
out = randint(10, 10, [0, 7])
out = randint(10, 10, 2^3)
```

See Also `rand`

Purpose

Raised cosine FIR filter design

Syntax

```
rcosfir(R);
rcosfir(R,n_T,rate,T);
rcosfir(R,n_T,rate,T,color);
rcosfir(R,n_T,rate,T,filter_type);
rcosfir(R,n_T,rate,T,filter_type,color)
b = rcosfir(...)
[b,sample_time] = rcosfir(...)
```

Description

The time response of the raised cosine filter has the form of

$$h(t) = \operatorname{sinc}\left(\frac{t}{T}\right)\frac{\cos\left(\frac{\pi R t}{T}\right)}{1-\frac{4R^2t^2}{T^2}}$$

`rcosfir(R)` produces time response and frequency response plots of the raised cosine filter at given `rolloff` factor `R` with default `n_T` value [–3, 3], default rate value 5, and default `T` value 1.

`rcosfir(R,n_T,rate,T)` produces the time and frequency responses of the raised cosine filter at a given `rolloff` factor `R`. `R` is limited to the range of $0 \le R \le 1$. `n_T` is a length two vector that indicates the number of `T` windows before and after zero. The default value for `n_T` is [–3, 3]. The parameter `rate` is the sample point in each `T`, or, the sampling rate of the filter is `T/rate`. The default value of `rate` is 5. The order of the FIR filter is:

```
(n_T(2)-n_T(1))*rate
```

`T` is the duration of the digital bit. The default value of `T` is 1.

`rcosfir(R,n_T,rate,T,color)` produces time and frequency responses of the raised cosine filter and a curve plot with the specified `color`. The choices for color are the same as those listed for the `plot` function.

`rcosfir(R, n_T, rate, T, filter_type)` produces time response and frequency response plots of a square root raised cosine filter if `filter_type == 'sqrt'`. The impulse response of a square root raised cosine filter is:

$$h(t) = 4r\frac{\cos\left((1+r)\pi\frac{t}{T}\right) + \dfrac{\sin\left((1-r)\pi\frac{t}{T}\right)}{4r\frac{t}{T}}}{\pi\sqrt{T}\left(\left(4r\frac{t}{T}\right)^2 - 1\right)}$$

`rcosfir(R, n_T, rate, T, filter_type, color)` produces time and frequency responses with the curve color specified in the string variable `color`.

`b = rcosfir(...)` returns the raised cosine FIR filter. The size of vector `b` is `(n_T(2) - n_T(1))*rate + 1`.

`[b, sample_time] = rcosfir(...)` returns the FIR filter and its sample time.

Examples

A simple example to compare the difference between different `rolloff` factors can be done by using these commands:

```
rcosfir(0);
subplot(211); hold on;
subplot(212); hold on;
rcosfir(.5,[],[],[],[],'r-');
rcosfir(1,[],[],[],[],'g-');
```

See Also

`rcosiir`

Purpose

Filter the input signal using raised cosine filter

Syntax

```
y = rcosflt(x, Fd, Fs);
y = rcosflt(x, Fd, Fs, type_flag);
y = rcosflt(x, Fd, Fs, 'filter', num);
y = rcosflt(x, Fd, Fs, 'filter', num, den);
y = rcosflt(x, Fd, Fs, type_flag, delay);
y = rcosflt(x, Fd, Fs, type_flag, delay, r);
y = rcosflt(x, Fd, Fs, type_flag, delay, r, tol);
[y, t] = rcosflt(...);
```

Description

Function `rcosflt` passes an input signal through a raised cosine filter. You can specify the raised cosine filter by the numerator and denominator of the filter's transfer function. The function designs a raised cosine filter if the one is not given.

`y = rcosflt(x, Fd, Fs)` filters the input signal `x` using a raised cosine FIR filter. The sample frequency for the input digital signal `x` is `Fd`, and the sample frequency for the output signal `y` is `Fs`. `Fs` must be larger than `Fd`, and `Fs/Fd` must be an integer. The row number of `y` (or vector length of `y`) is:

```
Fs/Fd*row_number_of_x
```

The other default values are:

Variable Name	Description	Default Value
`r`	rolloff factor	0.5
`delay`	filter time delay	3
`type_flag`	filter type	'fit/normal'

y = rcosflt(x, Fd, Fs, *type_flag*) gives specific computation instructions in string variable *type_flag*. The table below lists the types of designs available:

Type_flag	Meaning
'fir'	Design an FIR raised cosine filter before filtering the input signal.
'iir'	Design an IIR raised cosine filter before filtering the input signal.
'normal'	The designed filter is a normal raised cosine filter.
'sqrt'	The designed filter is a square root raised cosine filter.
'wdelay'	Keep the full length of the filtered result. In this case, the row number of y (or the vector length of y) equals (length_of_x + delay)*Fs/Fd.
'Fs'	The input variable x has sample frequency Fs instead of the default Fd. In this case, only element x(i*Fs/Fd + 1,:) is used in the calculation. All other elements in x are ignored.
'filter'	Specify the transfer function of the filter. Provide the numerator of the filter for an FIR filter; provide the numerator and the denominator for an IIR filter.
'default'	Use all default settings, which means design a normal FIR raised cosine filter, and cut off the extra length of the delay. Input signal has frequency Fd.

The instructions can be combined together by using a '/' to separate the individual instructions. For example, 'iir/sqrt/wdelay/Fs' is a valid *type_flag* string. The specified instructions cannot be contradict each other. For example, 'fir/iir' and 'normal/sqrt' are not valid instructions. Specifying 'default' with any other combinations results in the override of all other instructions; in this case, the function uses only the default settings in the filter design.

y = rcosflt(x, Fd, Fs, 'filter', num) passes input signal x through an FIR filter given in num.

`y = rcosflt(x, Fd, Fs, 'filter', num, den)` passes input signal `x` through an IIR filter. The numerator and the denominator of the filter is given in `num` and `den` respectively.

`y = rcosflt(x, Fd, Fs, type_flag, delay)` specifies the time delay in the raised cosine filter design. The value in delay is a positive integer. The actual time delay is `delay/Fd` in the filter design.

`y = rcosflt(x, Fd, Fs, type_flag, delay, r)` specifies the rolloff factor `r`. In general, it is a real number in range [0, 1].

`y = rcosflt(x, Fd, Fs, type_flag, delay, r, tol)` specifies the tolerance in the IIR filter design.

`[y, t] = rcosflt(...)` outputs the sampling time points for the output vector `y`.

See Also

`rcosfir`, `rcosiir`, `rcosine`

rcosiir

Purpose

Raised cosine IIR filter design

Syntax

```
rcosiir(R);
rcosiir(R,T_delay,rate,T,tol);
rcosiir(R,T_delay,rate,T,tol,filter_type);
rcosiir(R,T_delay,rate,T,tol,filter_type,color);
[num,den] = rcosiir(...);
[num,den,sample_time] = rcosfir(...);
```

Description

`rcosiir(R)` produces time and frequency response plots of the raised cosine IIR filter. `n_t` has the default value 3, `rate` the default value 5, `T` the default value 1, and `tol` the default value 0.01.

`rcosfir(R,n_T,rate,T,tol)` produces time and frequency response plots of a raised cosine IIR filter. The parameters of the filter are the same as those described in the function `rcosfir`. `R` is the `rolloff` factor; its value is limited to $0 \leq R \leq 1$. `n_T` specifies how many times of `T` delay it would be. `n_T` is a positive integer. The default value of `n_T` is 3. `rate` is a integer specifying the sample pints in each `T`. The default value of `T` is 1. `tol` is the tolerance value that determines the order of the IIR filter. If `tol`<1, `tol` is the tolerance factor. If `tol` >= 1, `ceil(tol)` is the order of the IIR filter. The default value of `tol` is 0.01.

`rcosiir(R,n_T,rate,T,tol,filter_type)` produces a square root raised cosine filter if `filter_type == 'sqrt'`.

`rcosfir(r,n_T,rate,T,tol,filter_type,color)` produces time and frequency responses with the curve color specified in the string variable `color`.

`[num,den] = rcosfir(...)` generates the transfer function of the raised cosine FIR filter. The size of vector `b` is `(n_T(2) - n_T(1))*rate + 1`.

`[num,den,sample_time] = rcosfir(...)` returns the transfer function and sample time of the FIR filter.

Examples

A simple example to compare the difference between different delay taps can be done by using these commands.

```
rcosiir(0,10);
subplot(211); hold on;
subplot(212); hold on;
col = ['r-';'g-';'b-';'m-';'c-';'w-'];
R = [8,6,4,3,2,1];
for i = R
   rcosiir(.0,i,[],[],[],[],col(find(R==i),:));
end;
```

From this example, you can see the frequency response of the filter approximate to the idea raised cosine filter with increased T-delay value.

See Also

`rcosfir`

rcosine

Purpose Design a raised cosine filter

Syntax

```
num = rcosine(Fd, Fs);
[num, den] = rcosine(Fd, Fs, type_flag);
[num, den] = rcosine(Fd, Fs, type_flag, delay);
[num, den] = rcosine(Fd, Fs, type_flag, delay, r);
[num, den] = rcosine(Fd, Fs, type_flag, delay, r, tol);
```

Description

`num = rcosine(Fd, Fs)` designs a finite impulse response (FIR) raised cosine filter. The input digital signal has frequency `Fd`. The sampling frequency for the filter is `Fs`. `Fs` must be larger than `Fd` and `Fs/Fd` must be an integer. The default rolloff factor is .5; and the time `delay` is 3, which means the filter delays `3/Fd` second.

`[num, den] = rcosine(Fd, Fs, type_flag)` designs a raised cosine filter using a method specified in the string variable *type_flag*. There are three possible options for *type_flag*:

- `'iir'`, which directs the function to generate an infinite impulse response (IIR) filter.
- `'sqrt'`, which directs the function to generate a square root raised cosine filter.
- `'default'`, which directs the function to generate an FIR, normal raised cosine filter. The default tolerance value in IIR filter design is 0.01.

You can create suitable combinations of design methods; `'sqrt'` with the `'iir'` or `'fir'`:`'iir/sqrt'`, but combinations like `'iir/fir'` are not allowed.

`[num, den] = rcosine(Fd, Fs, type_flag, delay)` provides the time `delay` step. The value in `delay` is a positive integer. The actual time delay is `delay/Fd` in the filter design.

`[num, den] = rcosine(Fd, Fs, type_flag, delay, r)` specifies the rolloff factor. In general, it is a real number in the range [0, 1].

`[num, den] = rcosine(Fd, Fs, type_flag, delay, r, tol)` specifies the tolerance in the IIR filter design.

See Also `rcosfir`, `rcosiir`, `rcosflt`

Purpose

The core of the Reed-Solomon decode calculation

Syntax

```
msg = rscore(code, k, tp, dim, pow_dim);
[msg, err, ccode] = rscode(code, k, tp, dim, pow_dim);
```

Description

`msg = rscore(code, k, tp, dim, pow_dim)` decodes a single row of codewords in `code` vector using the Reed-Solomon decoding technique. The number of information bits is `k`. The complete list of all members in GF(2^{dim}) is in `tp`. `dim` is a positive integer. The codeword length is provided in `pow_dim`, which equals $2^{dim}-1$.

`[msg, err, ccode] = rscore(code, k, tp, dim, pow_dim)` outputs the message signal `msg`, the error detected in the decoding `err`, and the corrected codeword `code`.

Note that unlike the other encode/decode functions, this function takes an exponential input in the processing. For example $\left[-Inf\ 0\ 1\ 2\ \ldots\right]$ represents $0\ 1\ \alpha\ \alpha^2\ \ldots$ in GF(2^{dim}). There are 2^m elements in GF(2^m). Hence, the input `code` represents $2^m(2^m-1)$ bits of information. The decoded `msg` represents $2^m * k$ bits of information.

See Also

`decode`, `encode`, `rsdecode`, `rsdeco`

rsdeco

Purpose Reed-Solomon decode computation with a choice of different formats

Syntax

```
msg = rsdeco(code, n, k);
msg = rsdeco(code, n, k, type_flag);
[msg, err] = rsdeco(...);
[msg, err, ccode] = rsdeco(...);
[msg, err, ccode, cerr] = rsdeco(...);
```

Description

`msg = rsdeco(code, n, k)` decodes the binary message in `msg` using the Reed-Solomon decoding technique. The codeword length is `n`, and the number of information bits `k`; these values must agree with those used in `rsenco` to encode the original message. A valid Reed-Solomon code should have its codeword length $n = 2^m-1$, where `m` is an integer greater than or equal to 3, and must have its number of information bits `k<n`. The input variable `code` must have the same format as the output of `rsenco`. There are two possible input formats for `code`:

- A column vector with length `L` or
- An `L`-by-`m` matrix.

In the column vector format, the output `msg` is an `Lc1` column vector where `Lc1 = L*k/n`. In the matrix format, the output `msg` is a `Lc2`-by-`m` matrix where `Lc2 = L*k/n`. In either case, if `Lc1` or `Lc2` is not an integer, the function will give error message and stop the decoding calculation. For faster calculation, the codeword length `n` can be the list of all elements in GF(2^m), a 2^m-by-`m` matrix. You can generate this list using `gftuple`.

`msg = rsdeco(code, n, k, type_flag)` specifies the format of `code` by using *type_flag*. There are three options for `type_flag`:

- `'binary'` specifies the binary representation for elements in `code`.
- 'decimal' specifies that the elements in code are base 10 integers in the range [0 , n–1].
- `'power'` specifies that the elements of `code` are in the power format for GF(2^m). See the *Tutorial* for a discussion of the power format for Galois fields. The elements in this representation are –Inf and the integers 0 to `n–1`. Any negative number in this form is equivalent to –Inf.

In either the 'decimal' or 'power' case, the elements in code must match the code type. In both these cases, code can be either:

- A length L column vector, in which case the output msg is an Lc column vector with Lc = L*k/n.
- An L-by-n matrix (note the dimensions are different from the 'binary' case, in which case the output code is a L-by-k matrix.

[msg, err] = rsdeco(...) outputs the number err, which specifies the number of errors that occurred in the decoding of the message.

[msg, err, ccode] = rsdeco(...) outputs the corrected code. The format of ccode matches the format of code in the input.

[msg, err, ccode, cerr] = rsdeco(...) outputs the number cerr, which specifies the number of errors found in ccode column.

Example

Example of using rsenco and rsdeco for error-control coding:

```
l = 1000;          % Bit number in calculation.
m = 4;
n = 2^m - 1;       % Set codeword length
k = n - 4;         % Set Number of information bits, error
                   %correction capability.
msg = randint(l,1); % Testing 1000 bits transfer.
tp = gftuple([-1 : n-1]',m);  % Generate the elements in
                   % gf(2^m).
[code,added] = rsenco(msg,tp,k);     % Encode.
noi = rand(length(code)/m,1) < .03;    % Add 3% noise.
noi = (noi*ones(1,m))';
noi = noi(:); % Construct noise for burst error.
code_noi = rem(code + noi,2);     % Add noise.
[dec,err,ccode,err_c] = rsdeco(code_noi,tp,k); % Decode.
msg = [msg; zeros(added,1)];  % Pad msg length for comparison.
max(abs(dec-msg))        % Compare received to transmitted
                   % message.
x = [1:length(noi)];      % x-axis for noise.
z =[1:m*n:length(noi)];
y=zeros(1,length(z));
```

```
z=[z;z];
y=[y+min(err_c);
y+max(err_c)];
subplot(211);
plot(x,noi,'yo',x,err_c,'rx',z,y,'g-'); % Placed vs. detected
                     error.
title('error detection record');
xlabel('o--placed error;x--detected error;vertical bar: rs-deco
section.');
axis([1,length(noi),min(err_c),max(err_c)]);
x = [1:length(msg)];                  % x-axis for msg.
z = [1:m*k:length(msg)];
y=zeros(1,length(z));
z=[z;z];
y=[y;y+max(msg)];
subplot(212);
plot(x,msg,'yo',x,dec,'rx',z,y,'g-');
title('message and decoded signal comparison');
xlabel('o--original message; x--decoded result.');
axis([1,length(msg),min(min(msg)),max(max(msg))]);
```

This example is a two-error correction case. If fewer than two burst errors occur between the indication lines on the plot, the method is able to correct the error.

See Also

rspoly, rsenco, rsdecode, encode, decode

Purpose

Reed-Solomon decoding for data in the exponential vector input/output format

Syntax

```
msg = rsdecode(code, k);
msg = rsdecode(code, k, tp);
[msg, err] = rsdecode(...);
[msg, err, ccode] = rsdecode(...);
```

Description

This function accepts codewords in the exponential input format. For example $\left[-Inf\ 0\ 1\ 2\ ...\right]$ represents $0\ 1\ \alpha\ \alpha^2$ ... in GF(2^m), where m is a positive integer. There are 2^m elements in GF(2^m). Hence, the input code represents $2^m(2^m-1)$ bits of information. The decoded msg represents $2^m * k$ bits of information.

msg = rsdecode(code, k) recovers the message information from codeword code. code must be the output of rsenco, which uses Reed-Solomon encoding technique. code is a matrix with a row length n = 2m -1, where m is a positive integer no less than 3. k is the number of information bits, which is usually an odd number. The error-correction capability of a Reed-Solomon code is t = floor((n-k)/2). The recovered message stored in msg; the row number of msg is the same as the row number of code. The column number of msg is k.

msg = rscode(code, k, tp) recovers the length k message from codeword code with the elements of GF(2^m) specified in tp. To speed up the process, we recommend this format even if the GF(2^m) is a default format. You can generate a default tp by using:

```
tp = gftuple([-1:2m-2]', m)
```

[msg, err] = rsdecode(...) provides the error correction information. A non-negative integer in err indicates the number of errors corrected; a negative integer indicates that there are more errors in the codeword than can be corrected.

[msg, err, ccode] = rsdecode(...) outputs the corrected codeword.

See Also

encode, decode, rsdeco

rsdecof

Purpose Reed-Solomon decoding for an encoded ASCII file

Syntax

```
rsdecof(file_in, file_out);
rsdecof(file_in, file_out, err_cor);
```

Description This function is the inverse process of the function `rsencof`. It decodes an `rsencof` encoded file.

`rsdecof(file_in, file_out)` decodes an ASCII file `file_in`, which must be the output of `rsencof`. The codeword and message lengths are fixed at 127 and 117, respectively. The error-correction capability is 5 for every 117 message characters and 127 codeword characters. The decoded message is written to `file_out`. Both `file_in` and `file_out` are string variables.

`rsdecof(file_in, file_out, err_cor)` specifies the error-correction capability for every 127 codewords. The message length is `127 - 2*err_cor`. The value in `err_cor` must match the value used in `rsencof`.

See Also `rsencof`, `encode`, `decode`

Purpose Reed-Solomon encoding with various input/output formats

Syntax

```
code = rsenco(msg, n, k);
code = rsenco(msg, n, k, type_flag);
code = rsenco(msg, n, k, type_flag, pg);
[code, added] = rsenco(...);
```

Description Reed-Solomon code is specially designed to correct burst errors instead of random errors. The function is designed to be able to take binary, decimal (non-negative integers), and power formats.

`code = rsenco(msg, n, k)` encodes the binary message in msg by using codeword length `n`, and number of information bits `k`. A valid Reed-Solomon code should has its codeword length `n = 2m–1`, where `m` is an integer no less than 3 and should have its number of information bits `k < n`. The input variable `msg` can be either a column vector with length `L`; or a `L`-by-`m` matrix. For the former `msg` input, the output code is a `Lc1` column vector with `Lc1=ceil(L/m/k)*m*n`. For the latter `msg` input, the output code is a `Lc2`-by-`m` matrix with `Lc2=ceil(L/k)*n`. To speed up the calculation, parameter `n` can be changed to a list of all elements in GF(2^m), a 2^m-by-`m` matrix.

`code = rsenco(msg, n, k, type_flag)` specifies the format of `msg` by using the string variable `type_flag`. There are three options for t `type_flag`:

- `'binary'`, the default option, specifies that the elements of code are in binary format.
- `'decimal'` specifies that the elements in `msg` are base 10 integers in the range [0, n].
- `'power'` specifies that the elements of `code` are in the power format for GF(2^m). See the *Tutorial* for a discussion of the power format for Galois fields. The elements in this representation are –Inf and the integers 0 to `n–1`. Any negative number in this form is equivalent to –Inf.

In either the 'decimal' or 'power' case, the elements in code must match the code type. In both these cases, code can be either:

- A length L column vector, in which case the output msg is an Lc column vector with Lc = L*k/n.
- An L-by-n matrix (note the dimensions are different from the 'binary' case, in which case the output code is a L-by-k matrix.

code = rsenco(msg, n, k, *type_flag*, pg) specifies the generator polynomial of the Reed-Solomon code.

[code, added] = rsenco(...) outputs the number added, which is the number of the elements added to msg based on whichever msg format it is.

The output code type will match the input msg data type. In the computation in this function, the type is converted in the order as follows:

For the message format;

msg: binary ⟶ decimal ⟶ power

For the codeword format:

code: power ⟶ decimal ⟶ binary

See Also

rsdeco, encode, decode

Purpose

Reed-Solomon encoding for data in the exponential vector input/output format

Syntax

```
code = rsencode(msg, pg, n);
```

Description

`code = rsencode(msg, pg, n)` encodes the message `msg` using Reed-Solomon code technique. `pg` is the generator polynomial. `n` is the codeword length. The decode pair for this function is `rsdecode`.

Note that the elements of `msg`, `pg`, and `code` are in GF(2^m) power form.

See Also

`rsdecode`, `encode`, `decode`, `rspoly`

rsencof

Purpose Reed-Solomon encoding for an ASCII file

Syntax

```
rsencof(file_in, file_out);
rsencof(file_in, file_out, err_cor);
```

Description

`rsencof(file_in, file_out)` encodes an ASCII file specified in `file_in` using (127, 117) Reed-Solomon code. The error-correction capability of this code is 5 (for every 127 characters). This function writes the encoded text to file `file_out`. Both `file_in` and `file_out` are string variables.

`rsencof(file_in, file_out, err_cor)` specifies the error correction capability for each 127 codewords. The message length is `127–2*err_cor`.

See Also `rsdecof`, `encode`, `decode`

Purpose

Produce a Reed-Solomon code generator polynomial

Syntax

```
Pg = rspoly(n,k);
Pg = rspoly(n,k,tp);
[Pg,t] = rspoly(...);
```

Description

Reed-Solomon code is a burst-error correction code. The codeword length of a Reed-Solomon code is $n=2^m-1$ where `m` is a positive number larger or equal to 3. The number of information bits `k` is a positive number less than `n`. The error correction capability of a Reed-Solomon code is `t = floor((n-k)/2)`. Since `n` is an odd number, for efficiency of the coding, we recommend that `k` is an odd number also.

`Pg = rspoly(n,k)` outputs the generator polynomial `Pg` of a Reed-Solomon code with codeword length `n`, number of information bits `k`. The output `Pg` is a polynomial in GF(2^m). Each element is represented by a power form.

`Pg = rspoly(n,k,tp)` produces generator polynomial using the specified GF(2^m) with its complete element listed in `tp`. `tp` is a (n+1)-by-`m` matrix where $n=2^m-1$. This parameter speeds up the calculation. When `tp` is a scalar, it specifies `m`.

`[Pg,t] = rspoly(...)` outputs the error-correction capability of the Reed-Solomon code.

Examples

The (15, 11) Reed-Solomon code generator polynomial can be obtained by

```
rspoly(15,11)
ans =
        10  3  6  13  0
```

which means that the generator polynomial is

$$\alpha^{10} + \alpha^{3}X + \alpha^{6}X^{2} + \alpha^{13}X^{3} + X^{4}$$

See Also

encode, decode, rsenco, rsdeco

sim2gen

Purpose

Produce a convolutional code transfer function in octal form

Syntax

```
gen = sim2gen(sim_file)
```

Description

`sim2gen` uses Simulink to define a specific structure of the convolutional code. The blocks used in building a convolutional code are:

memory blocks	Serve as the register blocks in convolutional code. The initial condition of the code is always set to be zero no matter what parameters have been set in the Simulink memory block entry.
XOR logic	Is the exclusive OR logic in a convolutional code structure.
inport block	Accepts the input. The inport number represents the sequential number of the input switch.
outport block	Defines the output sequence on the output side.

The edited convolutional block is used as an user interface only. You cannot use it for simulation purpose.

`gen = sim2gen(sim_file)` reads in a Simulink block diagram in building a convolutional code structure and outputs to a octal form transfer function matrix. The definition of an octal form transfer function is defined in the description for the function `oct2gen`. `sim_file` is a string that is the convolutional code Simulink block diagram filename.

Example

We take (3,2,2) convolutional code as an example. The structure of the code is taken from Figure 10.9 in reference . This figure shows the Simulink block diagram generated by `sim2gen`:

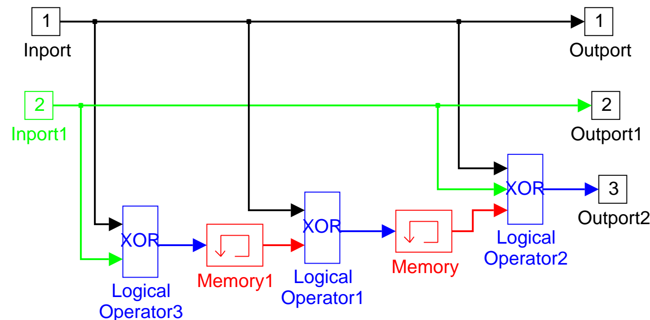


Figure 5-1: Simulink Block Diagram for (3,2,2) Convolutional Code

This Simulink block diagram is saved in `fig_10_9.m`. Using the function `sim2gen`, the octal form of the transfer function can be obtained by using the command:

```
gen = sim2gen('fig_10_9')
gen =

        4 0 7
        0 4 5
```

Limitation

No loops are allowed in the convolutional code structure. Use `sim2tran` if there are feedback loops in the convolutional code structure. Only the memory, XOR, Inport, and outport blocks are allowed in the Simulink diagram. The Simulink block diagram used for this function cannot be used in Simulink simulation.

See Also

`oct2gen`, `sim2tran`, `sim2log`, `gen2abcd`, `encode`, `decode`

sim2logi

Purpose Generate a nonlinear convolutional code transfer function

Syntax

```
gen = sim2gen(sim_file);
```

Description This function is a lower level function used called by the function `sim2tran`. `sim2logi` builds a nonlinear convolutional code transfer function. Please refer the reference page for `sim2tran` for a description of nonlinear transfer function and an example of the construction of a nonlinear transfer function. This function uses the Simulink delay block as a register in building nonlinear transfer functions.

`gen = sim2logi(sim_file)` reads in a Simulink block diagram to build a convolutional code structure and outputs a nonlinear transfer function matrix. The transfer function is represented in a nonlinear system form. A nonlinear system can be written as the difference equation:

$$x(k+1) = f(x(k), u(k))$$
$$y(k) = g(x(k), u(k))$$

where *u(k)* is the input vector, *x(k)* is the state, and *y(k)* is the output vector. `gen` is a two-column matrix. The contents of the output `gen` are:

$$\text{gen} = \begin{bmatrix} \text{Inf} & \text{N} \\ \text{M} & \text{K} \\ \text{U} & \text{V} \end{bmatrix}$$

where `N` is the output vector length, `K` is the input vector length, `M` is the state length, and `U` and `V` are column vectors with their length as 2^{K+M}.

Limitation You can only use the Simulink unit delay block to represent the register. Any other blocks with memory in the block diagram may cause errors in the generation of the nonlinear transfer function. This is because there is no specific way to distinguish the order of the registers. The Simulink block diagram used for this function cannot be used in Simulink simulation.

See Also `encode`, `decode`, `sim2gen`, `sim2tran`, `gen2abcd`

Purpose Produce a linear or a nonlinear convolutional code transfer function

Syntax

```
gen = sim2tran(sim_file);
```

Description `sim2tran` uses Simulink to define a specific structure of the convolutional code. The transfer function can be either a linear transfer function or a nonlinear function.

Linear Transfer Function
The blocks used in building a linear convolutional code transfer function are:

memory blocks	Serve as the register blocks in convolutional code. The initial condition of the code is always set to be zero no matter what parameters have been set in the Simulink memory block entry.
XOR logic	Is the exclusive OR logic in a convolutional code structure.
inport block	Accepts the input. The import number represents the sequential number of the input switch.
outport block	Defines the output sequence on the output side.

The edited convolutional block is used as an user interface only. You cannot use the block diagram for simulation purposes.

`gen = sim2tran(sim_file)` reads in a Simulink block diagram to build a convolutional code structure and outputs a transfer function matrix. The transfer function is represented in a linear system form, which can be written as a difference equation:

$$x(k+1) = \mathrm{A}x(k) + \mathrm{B}u(k)$$
$$y(k) = \mathrm{C}x(k) + \mathrm{D}u(k)$$

where $u(k)$ is the input vector, $x(k)$ is the state, and $y(k)$ is the output vector. Multiplication and addition are binary calculations. This function outputs the transfer function in this form:

$$\text{gen} = \left[\begin{array}{cc|c} A & B & \\ C & D & V \end{array}\right]$$

where V is a column vector with its first three element being the output vector length, the input length, and the state vector length. The last element of the vector is assigned to be *-Inf*.

Nonlinear Transfer Function

The blocks used in building a convolutional code are:

unit delays	Serve as register blocks in convolutional code. The initial condition of the code is always set to be zero no matter what parameters have been set in the Simulink memory block entry. The sample time is disregarded in generating the transfer function.
XOR logic	Is the exclusive OR logic in a convolutional code structure.
AND logic	Is the binary multiplication logic in a convolutional code structure.
inport block	Accepts the input. The import number represents the sequential number of the input switch.
outport block	Defines the output sequence in the output side.

You cannot use any other Simulink blocks to build a convolutional code block diagram. The convolutional code block diagram cannot be used for simulation purposes.

`gen = sim2tran(sim_file)` reads in a Simulink block diagram in building a convolutional code structure and outputs a nonlinear transfer function matrix.

The transfer function is represented in a nonlinear system form, which can be written as a difference equation:

$$x(k+1) = f(x(k), u(k))$$
$$y(k) = g(x(k), u(k))$$

where *u(k)* is the input vector, *x(k)* is the state, and *y(k)* is the output vector. gen is a two-column matrix. The contents of the output transfer function gen are:

$$\text{gen} = \begin{bmatrix} \text{Inf} & \text{N} \\ \text{M} & \text{K} \\ \text{U} & \text{V} \end{bmatrix}$$

where N is the output vector length, K is the input vector length, M is the state length, and U and V are column vectors with their length equal to 2^{K+M}.

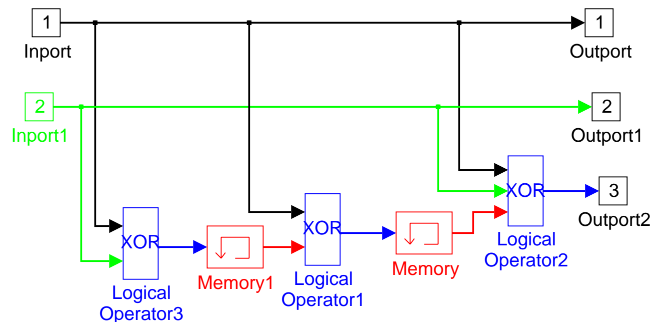
See the *Tutorial* for a discussion of nonlinear transfer functions in convolutional code.

Examples

This section provides two examples, a linear transfer function example and a nonlinear transfer function example. Note the different blocks used in building the linear and nonlinear transfer functions.

Linear Transfer Function

We take (3,2,2) convolutional code as an example. A Simulink block diagram is built as shown in the figure.



The Simulink block diagram is saved as file `fig_10_9.m`. Using the function `sim2tran`, the transfer function can be obtained by typing the command:

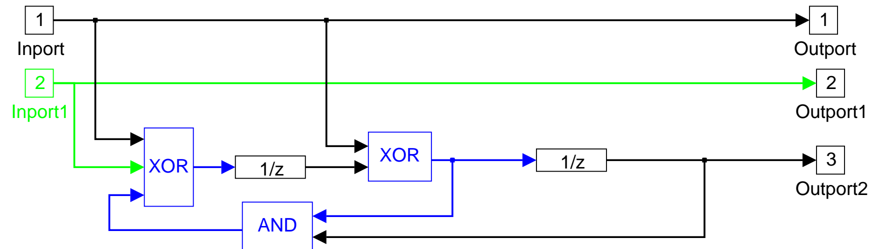
```
gen = sim2tran('fig_10_9')
gen =
```

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 3 \\ 1 & 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & -\mathrm{Inf} \end{bmatrix}$$

Using the command `gen2abcd`, you can further find `[A, B, C, D]` matrices in the description.

Nonlinear Transfer Function

We take (3,2,2) convolutional code as an example. The structure of the code is shown in the figure.



The Simulink block diagram is saved as file `nlconcd.m`. Using the function `sim2tran`, the transfer function can be obtained by typing the command:

```
gen = sim2tran('nlconcd')
gen' =
```

$$\begin{bmatrix} Inf & 2 & 0 & 0 & 1 & 3 & 3 & 1 & 2 & 2 & 2 & 2 & 3 & 1 & 1 & 3 & 0 & 0 \\ 3 & 2 & 0 & 4 & 0 & 4 & 1 & 5 & 1 & 5 & 2 & 6 & 2 & 6 & 3 & 7 & 3 & 7 \end{bmatrix}$$

Limitation

To build a linear convolutional code, you can use only the memory, XOR, Inport, and Outport blocks in the Simulink diagram. The Simulink block diagram used for this function cannot be used in Simulink simulation.

To build a nonlinear convolutional code transfer function, you can use only the unit delay block to represent the register. Any other blocks with memory in the block diagram may cause errors.

There is no specific way to distinguish the order of the registers. The Simulink block diagram used for this function cannot be used in Simulink simulation.

See Also

`sim2gen`, `sim2logi`, `gen2abcd`, `encode`, `decode`

symerr

Purpose

Compare elements in two matrices to find the number of differences and the ratio of the number of differences to the dimension of the matrices

Syntax

```
[numb,rati] = symerr(x,y);
[numb,rati] = symerr(x,y,direct);
```

Description

`symerr` compares two matrices of equal dimensions and finds the number of elements that do not agree. It also calculates the ratio of the number of elements that differ to the dimension of the matrices. Here if the matrices are `m`-by-`n`, the dimension is defined to be `m*n`.

`direct = 'row'` or `'column'` specifies in which direction the comparison is to occur.

Examples

See example in `biterr`.

See Also

`biterr`

Purpose

Convert a vector into a matrix

Syntax

```
mat = vec2mat(vec,m);
mat = vec2mat(vec,m,ad_to);
[mat,err] = vec2mat(...);
```

Description

`mat = vec2mat(vec,m)` converts vector `vec` into an `m` column matrix with row priority arrangement. If the length of `vec` is not multiple of `m`, the function adds 0 to the matrix. The column number of the matrix equals:

```
ceil(length(vec),m)
```

`mat = vec2mat(vec,m,ad_to)` specifies the elements to be added. Variable `ad_to` can be a vector or matrix. This function takes only the necessary number of elements from `ad_to` to make the matrix. The extra elements in `ad_to` are disregarded.

`[mat,err] = vec2mat(...)` outputs an integer `err`, which indicates how many elements has been added to make the matrix.

viterbi

Purpose

Decoding of convolutional code using the Viterbi algorithm

Syntax

```
msg = viterbi(code, tran_func);
msg = viterbi(code, tran_func, memory_length);
msg = viterbi(code, tran_func, memory_length, tran_prob);
msg = viterbi(code, tran_func, memory_length, tran_prob, plot_flag);
[msg, metric] = viterbi(...);
[msg, metric, survivor] = viterbi(...);
```

Description

`viterbi` recovers the message `msg` from the received codeword, which must be in the convolutional code format. The Viterbi algorithm is a maximum likelihood method designed to detect messages from codewords. This function can also generate trellis plots used to analyze the detection process.

`msg = viterbi(code, tran_func)` uses the Viterbi algorithm to recover the message in `code` with the transfer function matrix provided in `tran_func`. `tran_func` can be a linear, nonlinear, or octal form. See function `oct2gen` for the format of the octal form of a transfer matrix, `sim2gen` for the linear format, and `sim2logi` for the nonlinear format. `code` is a N column matrix received in a binary symmetric channel (BSC). All elements in `code` are either one or zero. `msg` is a K column matrix. The row number of `msg` will be the row number of code minus (M- 1).

`msg = viterbi(code, tran_func, memory_length)` specifies the maximum memory length in the decoding process by an integer number `memory_length`. The decode makes a decision based on the best metric in the limited memory length. As a rule of thumb, when the memory length is defined not less than 5∗K∗M, the decision is very close to optimal. When `memory_length` is defined less or equals to zero, the `memory_length` is considered as no limit.

`msg = viterbi(code, tran_func, memory_length, tran_prob)` recovers the `msg` using the transfer probability provided in `tran_prob`. `tran_prob` is a three row matrix. The first row specifies the receiving range, the second row the

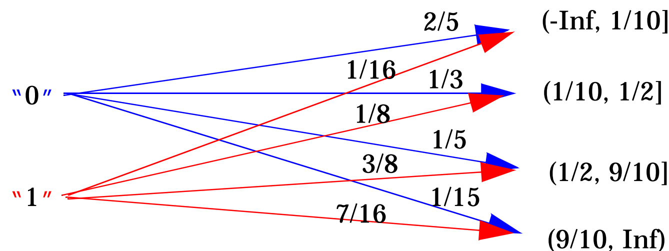
probability of a zero transmission to the receiving range, and the third row the probability of signal one transfering to the receiving range. For example,

$$\text{tran_prob} = \begin{bmatrix} -Inf & \frac{1}{10} & \frac{1}{2} & \frac{9}{10} \\ \frac{2}{5} & \frac{1}{3} & \frac{1}{5} & \frac{1}{15} \\ \frac{1}{16} & \frac{1}{8} & \frac{3}{8} & \frac{7}{16} \end{bmatrix}$$

means that signal 0 has 2/5 probability being received in range (–Inf, 1/10], 1/3 in range (1/10, 1/2], 1/5 in range (1/2, 9/10], and 1/15 in range (9/10, Inf). Signal 1 has 1/16 probability being received in range (–Inf, 1/10], 1/8 in range (1/10, 1/2], 3/8 in range (1/2, 9/10], and 7/16 in range (9/10, Inf). The table below illustrates the transition probabilities specified in the `tran_prob` matrix:

Receiving range	(–Inf, 1/10]	(1/10, 1/2]	(1/2, 9/10]	(9/10, Inf)
Signal zero transfer probability	2/5	1/3	1/5	1/15
Signal one transfer probability	1/16	1/8	3/8	7/16

A tree diagram below is another way of organizing the probability data:



`code` is transmitted in a discrete memoryless channel (DMC). When `tran_prob` is simply a non-matrix scalar, the parameter is ignored. In case of `tran_prob` is ignored, `code` must be transmitted in BSC.

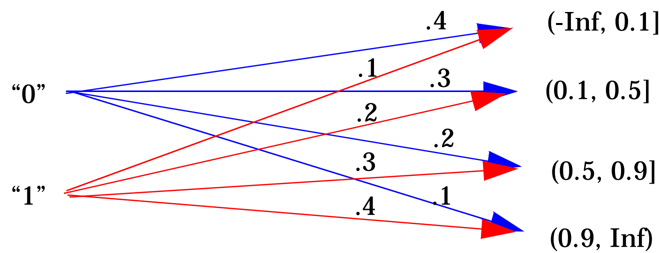
`msg = viterbi(code, tran_func, memory_length, tran_prob, plot_flag)` plots the trellis when `plot_flag` is a positive integer. When `plot_flag` is less or equal to zero, this parameter is ignored. The function opens a figure with the figure number equal to the value of `plot_flag`. The trellis figure has the time as *x*-axis and transfer state as *y*-axis. The white lines are the possible transfers. The yellow lines are the survivors in a single step calculation. The red lines are the final survivor path. The numbers at the state points are the metric values (when `tran_prob` is not specified, the numbers are Hamming distance values).

`[msg, metric] = viterbi(...)` outputs the metric value in the calculation. When `tran_prob` is not specified, `metric` outputs the Hamming distance in the calculation.

`[msg, metric, survivor] = viterbi(...)` outputs the codeword in the survivor path.

Examples

Assume the transfer channel has the transfer probability as shown in the following figure.



The transfer probability matrix is as follows.

```
tran_prob = [
-Inf, .1  .5  .9
.4    .3  .2  .1
.1    .2  .3  .4];
```

Assume the receiver have received the following signal

```
r = [1 .8 0;  1 1 .2;  1 1 0;  1 1 1;  0 .8 0;  .8 .2 0;  .8 0 1];
```

The convolutional transfer function matrix is a (3, 1, 2) structure with the octal form of the transfer function being:

```
tran_func = [6 5 7];
```

Using function `viterbi`, assuming the memory length is 5, the maximum likelihood recovered message `msg`, the metric `mtc`, and the survivors `suv` can be calculated by the following line:

```
[msg, mtc, suv] = viterbi(r, tran_func, 5, tran_prob, 1);
```

The trellis figure generated from the command is shown in the figure on the next page. The survivors are:

```
suv = [1 1 1; 0 1 0; 1 1 0; 0 1 1; 0 0 0; 0 0 0; 0 0 0];
```

The example is modified from example 11.1 in Reference .

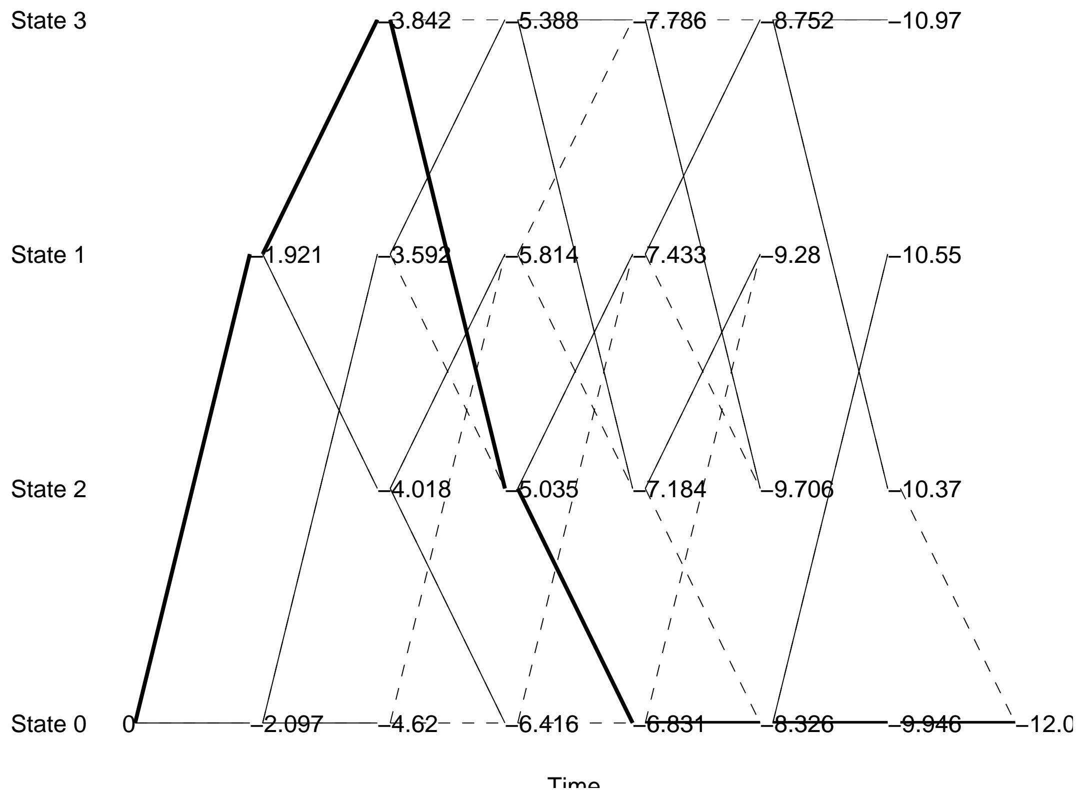


Figure 5-2: A Trellis Path

The thicker lines form the path of the survivors.

See Also

sim2gen, oct2gen, convenco, encode, decode

Algorithm

The dynamic programming method is used in the computation in this function. See the *Tutorial* for a discussion of the implementation of this algorithm.

6

Simulink Block Library Reference

Sources and Sinks in Communication Systems 6-4

Source Coding 6-38

Error-Control Coding 6-53

Modulation and Demodulation 6-110

Signal Multiple Access 6-235

Transmitting and Receiving Filters 6-249

Channels 6-261

Synchronization 6-275

Utilities 6-285

The Simulink block library of the Communications Toolbox is shown below. The structure of the library is arranged as a high level overview block diagram of a typical single direction transmitting communication system. Each block in this library contains a sublibrary. A sublibrary window opens when you double-click on a block in this library.

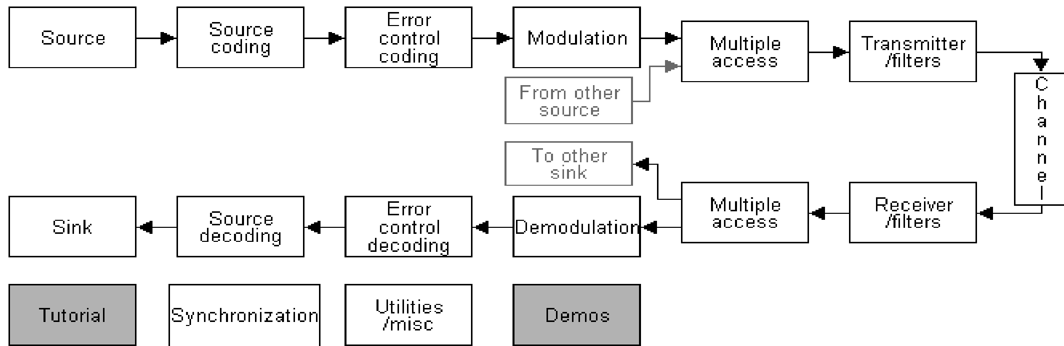


Figure 6-1: The Simulink Block Library of The Communications Toolbox

This figure contains a comprehensive set of communication subsystems. A specific implementation of a communication system may contain only some of the subsystems provided in the Simulink block library.

In communication systems, transmission and receiving techniques are used together. In most, but not all, cases, if a technique is used in the transmitting side, the same technique should be used in the receiving side. For example, if an amplitude modulation method is used in the transmission side, then an amplitude demodulation method must be used in the receiving side in order to recover the original signal correctly. Because the methods are used in pairs, the transmitting and receiving blocks in the same category are arranged into the same sublibrary. For example, double-clicking the Error-Control Coding block

or the Error-Control Decoding block opens the same block library window. This chapter discusses the following sublibraries in the Simulink block library:

- Sources and Sinks of Communication Systems
- Source Coding in Transmitting and Receiving
- Error-Control Encoding and Decoding
- Signal Modulation and Demodulation
- Multiple Access in Transmitting and Receiving
- Transmitting and Receiving Filters
- Channel
- Synchronization
- Utilities
- Error Analysis

The last item, Error Analysis, is not shown in the figure6-1. The functions in this category are available only using MATLAB functions.

In the Communications Toolbox Simulink block library, there are two kinds of blocks: function blocks and example blocks. Function blocks are ready-to-use blocks for specific applications. Example blocks are cyan colored blocks with no input or output ports. This toolbox provides example/demo blocks to show how to use the function blocks. A block diagram window opens when you double-click on an example/demo block.

Most of the blocks in the Simulink block library use “masking.” You can unmask any block to find the details of the implementation. You can also change the block implementation to meet your specific application needs.

Sources and Sinks in Communication Systems

In a communication system, signal source, storage, and display play very important roles. A signal source can be an analog signal, a digital signal, a variable, or a computer data file. The task of the transmitting side of a communi-cation system is to convert information into a signal that can then transmit through the communication channel. The receiving side requires storage and display elements. Communications engineers often store received signals in such a way that the received signal is available for recall at any time.

There are a number of special plots for communication system analysis. This toolbox includes some of the most frequently used plots, such as eye-pattern, scatter, and error-rate meter. This figure shows the Source and Sink Sublibrary:

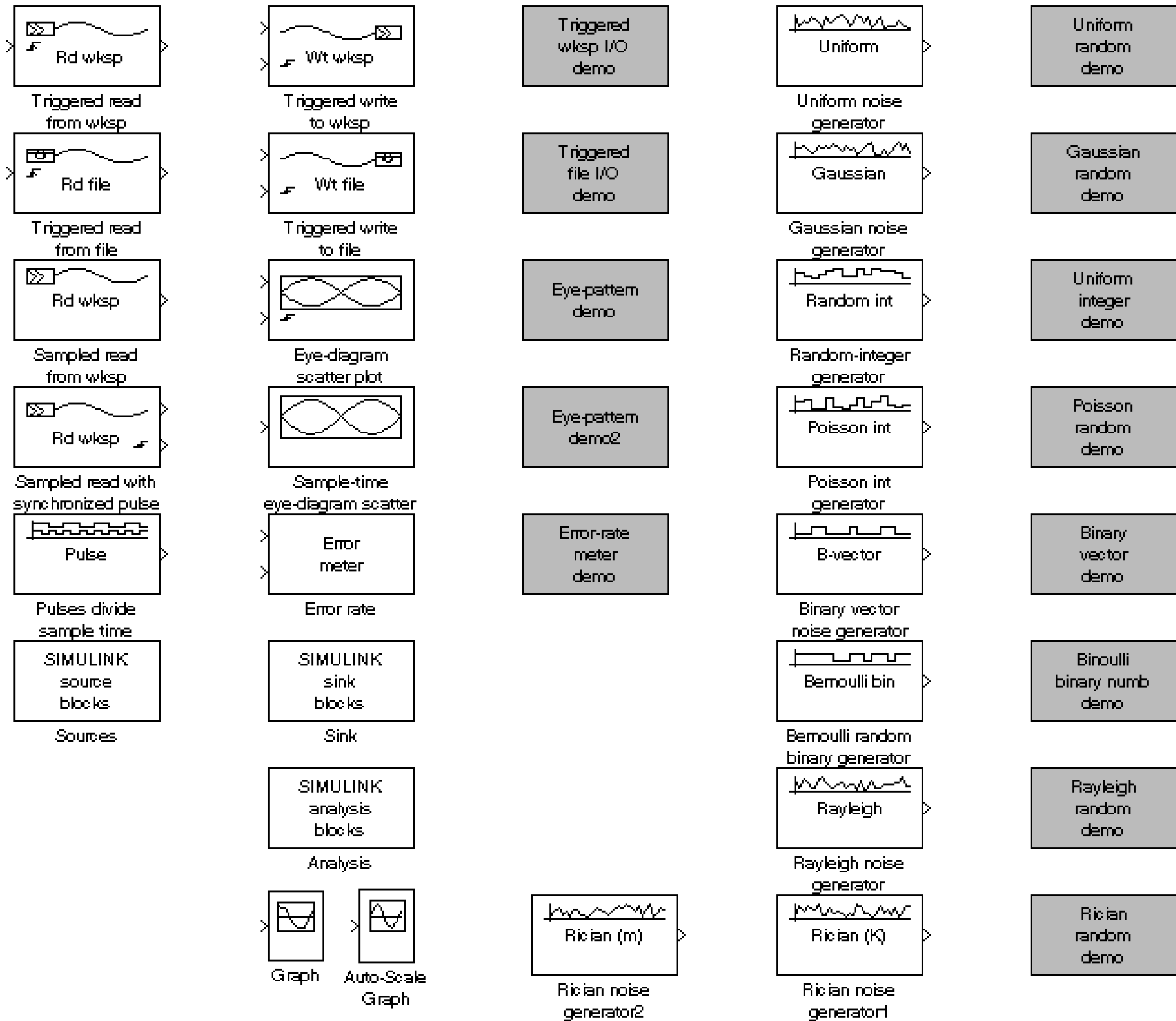


Figure 6-2: Source and Sink Sublibrary

Sources and Sinks Reference Table

This table lists the Simulink blocks in the Sources and Sinks sublibrary. (They are listed alphabetically in this table for your convenience.):

Block Name	Description
Bernoulli Random Binary Noise Generator	Generates Bernoulli distributed binary numbers
Binary Vector Noise Generator	Generates arbitrary length vectors with random 0's and 1's
Error-Rate Meter	Calculates and displays symbol and bit error rates
Eye-Pattern & Scatter Plot	Generates eye-pattern diagrams and scatter plots
Gaussian Random Noise Generator	Generates average white Gaussian noise (AWGN)
Poisson Random Integer Generator	Generates Poisson distributed integers in a specified range
Rayleigh Noise Generator	Generates Rayleigh distributed noise
Rician Random Noise Generator	Generates Rician distributed noise
Sample Time Eye-Pattern Diagram & Scatter Plot	Generates eye-pattern diagrams and scatter plots
Sampled Read From Workspace	Reads data from a workspace file at specified sample times
Triggered Read From File	Reads a row of data from a file when triggered
Triggered Read From Workspace	Reads a row of data from a work-space variable when triggered
Triggered Write To File	Writes a row of data to a file when triggered

Block Name	Description
Triggered Write To Workspace	Writes a row of data to a workspace variable when triggered
Uniform Random Integer Generator	Generates uniformly distributed integers in a specified range
Uniform Random Noise Generator	Generates uniformly distributed random noise
Vector Pulse	Generates a pulse train of vectors

Uniform Random Noise Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

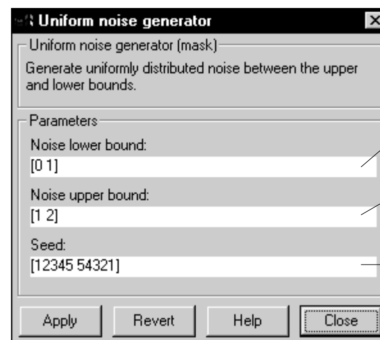
Description

The Uniform Random Noise Generator block generates uniformly distributed noise. The output data of this block is uniformly distributed between specified lower and upper boundaries. The upper bound must be larger than the lower boundary.

You must specify the seed in the simulation. When the seed is a fixed number, the generated noise is repeatable.

When the seed is a vector, the output of the block is a vector with a length equal to the length of the input seed. You can specify the boundaries of each element of the output vector individually or, if all the elements of the output vector are independent, identically distributed (i.i.d)., you can use a scalar for the boundary. If the boundaries are vectors, the vector length must equal the seed vector length.

Dialog Box



The lower bound of the uniform distribution noise.

The upper bound of the uniform distribution noise. The vector size must be the same size as the lower bound of the output.

Initialize the seed value. The vector size of the seed sets the vector size of the output.

Equivalent M-function

`rand` with control of upper and lower boundaries

Gaussian Random Noise Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

Description The Gaussian Random Noise Generator block generates white Gaussian noise.

You must specify the seed in the simulation. When the seed is a fixed number, the generated noise is repeatable. When the input seed is a vector, the output of the block is a vector of equal length.

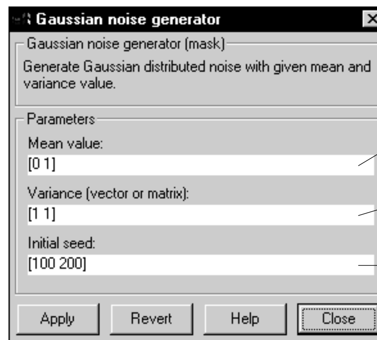
The mean value and the covariance matrix can be either scalars or vectors. When the mean value is a scalar, every element of the output signal shares the same mean value. Likewise, when the covariance matrix is a scalar, every element of the output signal shares the same variance and the elements are uncorrelated each other.

When the covariance matrix is a vector, the vector size must be the same as the vector size for seed. In this case, the covariance matrix is a diagonal matrix with the diagonal element provided in the vector. Since the off-diagonal elements are zero, the output Gaussian random variables are uncorrelated.

When the covariance matrix is a square matrix, the off-diagonal elements are the correlations between pairs of output Gaussian random variables. Note that in this case, the covariance matrix must be a positive semi-definite. The matrix must be `N`-by-`N`, where `N` is the vector length of the seed.

Gaussian Random Noise Generator

Dialog Box



Equivalent M-function

rand with control of upper and lower boundaries

Uniform Random Integer Generator

Catagory

Noise Generator

Location

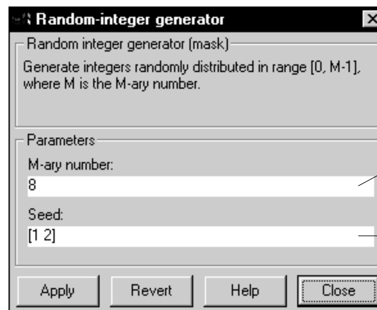
Source/Sink Sublibrary

Description

The Uniform Random Integer Generator block generates uniformly distributed random integers in the range [0, M-1], where M is the multiple number defined in the dialog box.

The output vector has the same length as the input seed. The output of the block is a vector when the input seed is a vector. The multiple number M can be either a scalar or a vector. If it is a scalar, all the output random variables are i.i.d. If the multiple number is a vector, its length must equal the length of the seed; in this case each output has its own output range.

Dialog Box



Specify the multiple number(s), which must be a positive integer. The integers generated in this block are in the range [0, M-1].

Initialize the seed. The vector length of the seed sets the length of the output vector.

Equivalent M-function

`randint`

Poisson Random Integer Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

Description

The Poisson Random Integer Generator block generates random integers with a Poisson distribution.

The Poisson probability density function is

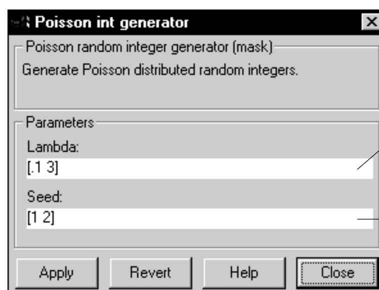
$$y = \frac{\lambda^{x}}{x!} e^{-\lambda}$$

where x is a non-negative integer and λ is the Poisson parameter, which must be positive. The probability density function gives the probability of the appearance of a non-negative integer x in the block output sequence.

You can use the Poisson random integer generator to generate noise in a binary transmission channel. In this case, the Poisson parameter λ should be (much) less than 1.

When the input seed is a vector, the output of the block is a vector of equal length.

Dialog Box

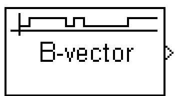


Specify the Poisson parameter λ. When it is a scalar, every element in the output vector shares the same λ. When it is a vector, the vector length must be same as the seed's.

Initialize the seed.

Equivalent M-function

`poissrnd` in the Statistics Toolbox



Binary Vector Noise Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

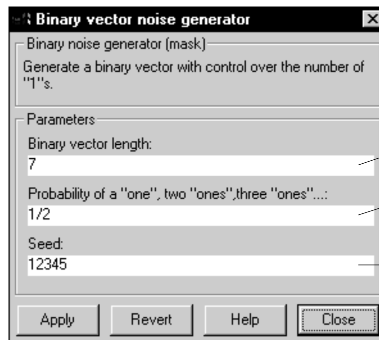
Description

The Binary Vector Noise Generator outputs a length N binary random vector, where N is the **Vector length**. The appearance of the “1”s in the output vector is uniformly distributed.

You can specify the entry for this block as a vector, [p1, p2, p3,...pM], where p1 is the probability of a single “1” in the binary vector, p2 is the probability of two “1”s in the binary vector, p3 is the probability of three “1”s in the binary vector, and so on. The vector size M must be less than or equal to the block output length. Please note that $\sum_{i=1}^{m} p_i \le 1$. The probability of all zero vectors is $1 - \sum_{i=1}^{m} p_i$. The minimum vector size for this probability vector is 1 and the maximum vector size for this vector is the vector length specified in **Vector length.**

This block is useful in testing error-control coding algorithms.

Dialog Box



Specify the vector length N.

Specify the probability sequence. The sum of this vector must be less than or equal to 1.

Initialize the seed.

Rayleigh Noise Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

Description The Rayleigh Noise Generator block generates Rayleigh distributed noise.

The Rayleigh probability density function is given by

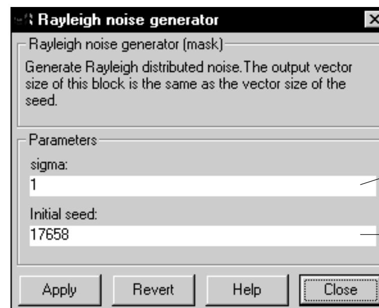
$$f(x) = \begin{cases} \frac{x}{\sigma^2} e^{-\frac{x^2}{2\sigma^2}} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where σ^2 is known as the fading envelope of the Rayleigh distribution.

The block requires you to specify the seeds in the simulation. When the seed is a fixed number, the generated noise is repeatable.

When the input seed is a vector, the output of the block is a vector of equal length. The **Fading envelope** can be either a scalar or a vector with its length equal to the length of the **Seed**. When the fading envelope is a scalar, every element of the output signal shares the same fading envelope.

Dialog Box



Specify σ as defined in the Rayleigh probability density function.

Initialize the seed.

Equivalent M-function `raylrnd` in the Statistics Toolbox

Rician Random Noise Generator

Catagory Noise Generator

Location Source/Sink Sublibrary

Description The Rician Random Noise Generator block generates Rician distributed noise.

The Rician probability density function is given by

$$f(x) = \begin{cases} \frac{x}{\sigma^2} I_0\left(\frac{mx}{\sigma^2}\right) e^{-\frac{x^2+m^2}{2\sigma^2}} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where σ^2 is the variance of the Gaussian distribution in generating the Rician distribution noise. $m^2 = m_I^2 + m_Q^2$, where m_I and m_Q are the mean values of the two independent Gaussian components used in constructing the Rician

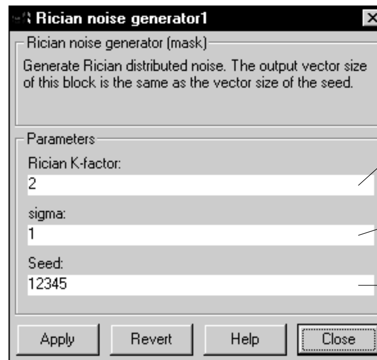
distribution. Note that **m** and σ^2 are not the mean value and the variance for the Rician distributed random number. Function $I_0(.)$ is the modified 0th order Bessel function of the first kind given by:

$$I_0(y) = \frac{1}{2\pi}\int_{-\pi}^{\pi} e^{y\cos t}\, dt$$

You must specify the seeds in the simulation. When the seed is a fixed number, the generated noise is repeatable.

When the seed is a vector, the output of the block is a vector of equal length. The mean value and the variance can be either a scalar or a vector; in the second case, the vector size must equal the seed vector size. When the mean value (or variance) is a scalar, every element of the output signal shares the same mean value (or variance).

Dialog Box



Specify the Rician K-factor, where

$$K = \left(m_I^2 + m_Q^2\right)/(2\sigma^2) .$$

Specify the variance is the variable σ^2 in the above Rician probability density function.

Note that the mean and variance specified are not those of the Rician distribution, but of the underlying Gaussian random variable.

Bernoulli Random Binary Noise Generator

Category

Noise Generator

Location

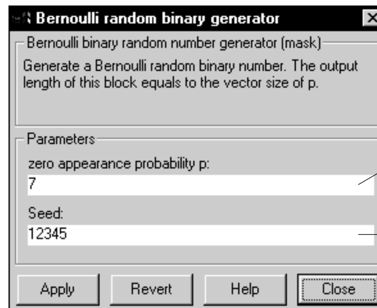
Source/Sink Sublibrary

Description

The Bernoulli Random Binary Noise Generator block generates Bernoulli random binary numbers. The output of this block are binary numbers. The probability of the zero output is *p* and the probability of the one output is *1-p*. Using probability notation, we have $P_X(0) = p; P_X(1) = 1-p$. The mean value of the Bernoulli distribution is *1-p*. The variance of the Bernoulli distribution is *p(1-p)*. The value of *p* must be no less than zero and no larger than one.

You must specify the zero occurring probability *p* in the simulation. When *p* parameter is a vector in the entry, the output of this block is a vector with its vector length the same as the parameter *p*. You need also to specify the seed in the simulation. The seed parameter can be a scalar or a vector with the same length as the vector *p*.

Dialog Box



The probability of outputing zeros. The output size of this block is the same as the vector size of this parameter.

Initialize the seed value. The seed can be a scalar or a vector with the same length as the vector in the first entree of this block.

Equivalent M-function

```
x = rand(n, m) > p;
```

Triggered Read From Workspace

Catagory Triggered Read

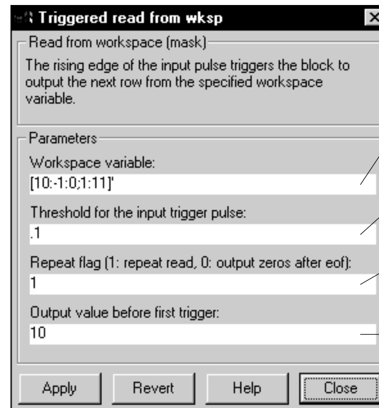
Location Source/Sink Sublibrary

Read a row of data from a workspace variable at the raising edge of the trigger signal.

Description Unlike the Simulink Read From Workspace block, The Triggered Read From Workspace block reads the workspace variable only at the raising edge of the input signal to this block. There are four parameter entries to this block: **Workspace variable**, **Threshold**, **Repeat flag**, and **Output value before first trigger**.

The **Workspace variable** is a matrix with its column number being the vector size of the output of this block. Before receiving any triggered signal, this block outputs the value in **Output value before first trigger**. When the trigger signal crosses from below the **Threshold** value, the block reads a row of data from the given workspace variable. When the block reaches the last row of the workspace, it returns to the first row of the variable or it outputs zero depending on the **Repeat flag** specification.

Dialog Box

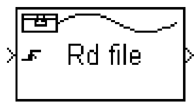


Specify the workspace variable name or the contents of the variable. The column size of this parameter sets the output vector's size.

Set the threshold for the rising edge of the trigger signal.

Specify whether to cycle continuously through the workspace variable or to output zeros after end-of-file (eof).

Initialize the output value prior to the first trigger.



Triggered Read From File

Catagory Triggered Read

Location Source/Sink Sublibrary

Description Unlike the Simulink Read From File block, the Triggered Read From File block reads a record from a file only at the raising edge of the input trigger signal. The file can be an ASCII text file, a data file, or a binary file.

You must specify the vector length of the block output. The block reads the given length of record from the file when a raising edge of the input trigger signal is detected. When the data type is specified as ASCII form, the output is an integer. The map between decimal integer, hexadecimal, and ASCII character is shown in the table below. When the data type is other than ASCII text, this block reads a given length of data from the file.

Decimal	Hex Code	Ascii Char	Decimal	Hex Code	Ascii Char	Decimal	Hex Code	Ascii Char	Decimal	Hex Code	Ascii Char
0	00	NUL	32	20	SP	64	40	@	96	60	\
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	“	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	‘	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

When end-of-file is reached, you can specify a recursive index to control the output value. The block goes back to read the very beginning of the file when the recursive index is “1”. The block outputs zeros when the recursive index is “0”.

You can also specify how many triggers the block receives between file reads in the **Number of trigger pulses between saved data** parameter. Setting this parameter to 0 forces the block to read a record at every trigger raising edge. The very first record read from the file always occurs after the first raising edge.

Dialog Box

Triggered read from file

Read from file (mask)

The rising edge of the signal from the 2nd port triggers the block to read the next row of the specified file.

Parameters

File name (string, include extension): 'junk.tmp' — Specify the filename, including its extension.

Data type ('ascii','float','integer'): 'ascii' — Specify the data type as a string variable: `'ascii'`, `'float'`, `'integer'`, or `'binary'`.

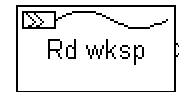
Number of trigger pulse between saved data: 0 — Control how many trigger raising edges should be received between file reads.

Output vector length: 1 — Set the vector length of the file output.

Repeat flag (1: repeat read, 0: output zeros after eof): 0 — Specify whether to cycle continuously through the workspace variable or to output zeros after eof.

Threshold in detecting trigger signal: .3 — Set the threshold for the raising edge of the trigger signal.

Apply | Revert | Help | Close



Sampled Read From Workspace

Catagory Sampled Read

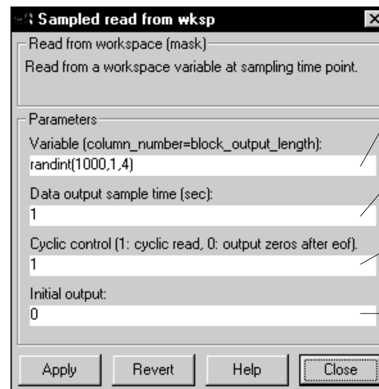
Location Source/Sink Sublibrary

Description

The Sampled Read From Workspace block reads a row of data for a workspace variable at every sampling point. This block reads the workspace variable only at the exact sampling point of the sample time.

The workspace variable is a matrix. The column number of the workspace variable is the vector length of the block output. When the simulation time is less than the offset, this block outputs the initial output value. When simulation time reaches the next sampling point, the block reads a row of data from the given workspace variable. When the last row of the workspace is reached, the block goes back to the first row of the variable or it outputs zeros depending on the **Repeat flag** value.

Dialog Box

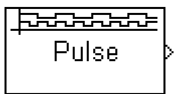


Specify the variable name or the contents of the variable. The column size of this variable sets the vector length of the output.

The sampling time of this block. When this parameter is a two-element vector, the second element is the offset value.

Specify whether to cycle continuously through the workspace variable or to output zeros after end-of-file (eof).

Initialize the block output. This is the value output before the block reaches the offset value.



Vector Pulse

Catagory Pulse Signal Generator

Location Source/Sink Sublibrary

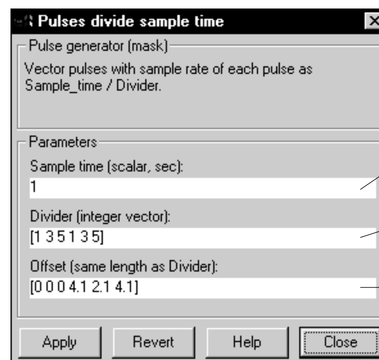
Description

The Vector Pulse block generates a vector of pulse signals. The sample frequency of every element in the vector pulse signal is an integer multiplied by a common frequency F_0. You can specify the offset for each pulse. The advantage of using the common frequency is that this block uses common frequency in the timing calculation, which eliminates any relative timing error between the elements outputted from the block. This makes the calculation more accurate.

The pulses generated in this block have their pulse width equal to exactly half of the pulse sampling time. The pulse amplitude is 1.

You must specify the **Divider** in this block. This is a vector of positive integers that set the sample times for the individual output pulses. Each element of the vector is divided by the sample time to produce the individual sample times. If t_s is the sample time and [d1, d2, ..., dn] is the divider vector, then the individual pulse sample times are $d1/t_s$, $d2/t_s$, ... dn/t_s. In other words, the frequencies for the individual pulses are $t_s/d1$, $t_s/d2$, ..., t_s/dn. The length of the **Divider** also sets the length of the output vector.

Dialog Box



Specify the sample time. The inverse of the sample time is the common frequency.

A vector of positive integers.The output vector length equals the length of the divider.

The offset for the beginning of the pulses. The elements of this vector must be non-negative real numbers.

Example

This example assumes that the three parameter entries are as follows. The **sample time** is Fo, the **Divider is** [2 3], and the **Offset** is [offset, 0]. The output of this block is shown belowFigure 6-3:. This figure is drawn with the assumption that offset < 1/Fo/2.

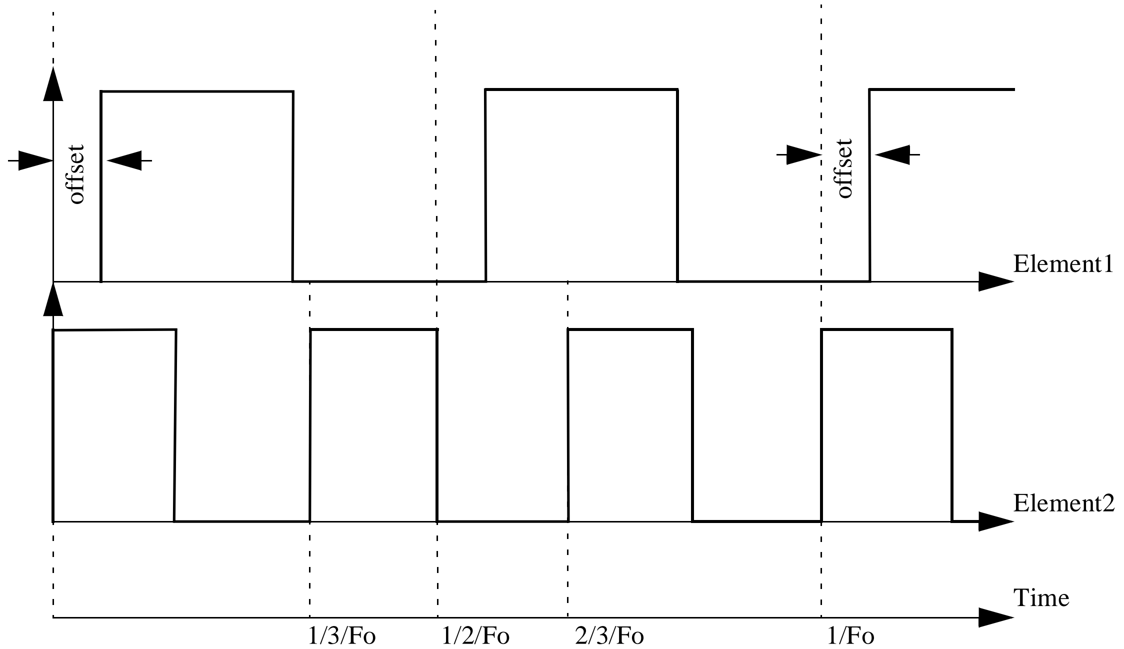


Figure 6-3: Two-Element Vector Output

Triggered Write To Workspace

Catagory Triggered Write

Location Source/Sink Sublibrary

Description Unlike the Simulink Write To Workspace block, the Triggered Write To Workspace block writes to a workspace variable only at the raising edge of the trigger signal. There are six entries to this block: **Workspace variable**; **Data type**; **Number of trigger pulses between save data**; **Maximum row number**; **Section to keep when overflow**; and **Threshold**.

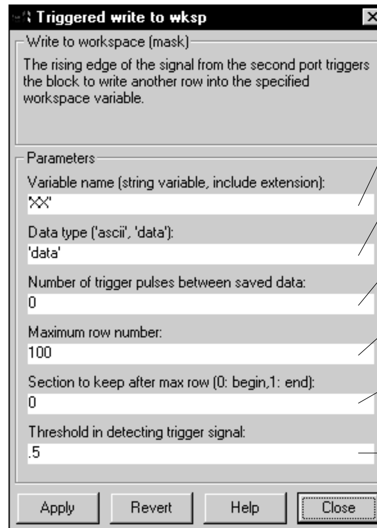
The first input port accepts the message signal. The second input port takes the trigger signal. The message signal to the first input port can be either a scalar or a vector. The trigger signal to the second input port must be a scalar.

The saved workspace variable is a column vector when the first port has a scalar input. It is a matrix when the first port has a vector input. The first element of the input signal vector is saved in the first column; the second element of the input signal vector is saved in the second column, and so on.

The block saves the first record at the first raising edge of the trigger signal. The data can be saved as string or data. The maximum row of the output variable is limited by a pre-defined number. This number cannot be changed during simulation. After the limit is reached, the block keeps the first part of the simulation data or the latest simulation data depending on the entry into **Section to keep when overflow**.

You can specify how many triggers the block receives between file reads in the **Number of trigger pulses between save data** parameter. Setting this parameter to 0 forces the block to read a record at every trigger raising edge. The very first record read from the file always occurs after the first raising edge.

Dialog Box



A string variable. You must include the extension if there is one.

Choose one of two formats: `'ascii'` or `'data'`. `'ascii'` forces a save in the ascii format, `'data'` in the MATLAB data format.

Control how many trigger raising edges should be received between file reads.

Limit the number of columns of data that can be saved.

If the variable reaches its maximum record number, select 0 to keep the first part of the simulation record and 0 to keep the last part.

Set the threshold for the raising edge of the trigger signal.

Triggered Write To File

Catagory Triggered Write

Location Source/Sink Sublibrary

Description

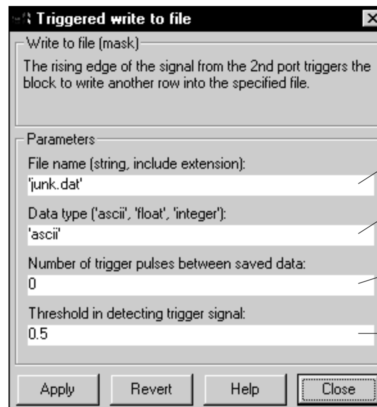
Unlike the Simulink Write To File block, the Trigger Write To File block writes a record to a file only at the raising edge of the input trigger signal. The file can be an ASCII text file or a data file.

The first input port accepts the message input signal. The second input port takes the trigger signal, which must be a scalar. The block writes to the file at the raising edge of the input trigger signal.

When the data type is specified as ASCII form, this block converts the input data into ASCII characters by using the map given in the Triggered Read From File block. When the data type is a type other than ASCII text, this block directly writes the data into the file. Each triggered record is written to a separate line in the file.

You can also specify how many triggers the block receives between file reads in the **Number of trigger pulses between saved data** parameter. Setting this parameter to 0 forces the block to read a record at every trigger raising edge. The very first record read from the file always occurs after the first raising edge.

Dialog Box



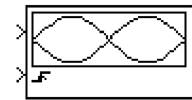
Specify the filename, including its extension.

Specify the data type as a string variable: `'ascii'`, `'float'`, `'integer'`, or `'binary'`.

Control how many trigger raising edges should be received between file reads.

Set the threshold for the raising edge of the trigger signal.

Eye-Pattern & Scatter Plot



Catagory Plotting

Location Source/Sink Sublibrary

Description The Eye-Pattern & Scatter Plot block plots eye-pattern plots, scatter plots or x-y plots depending on the parameter specification.

Eye-pattern plot
An eye-pattern plot is a simple and convenient tool to study the effects of inter-symbol interference and other channel impairments for digital transmission. The received signal is plotted against time, but when the *x*-axis time limit is reached, the signal goes back to the beginning of the time point. This produces an overlaid set of plots. Typically, the time range is an integer multiple of the symbol interval.

The offset time (or decision point) is plotted as a vertical line. Generally this should occur when the "eye" of the plot is open to its widest point. This is the point when the Scatter plot samples the signal, which is known as the decision point.

Scatter plot
Scatter plots record the signal value at a given decision point. When the input signal is a two-dimensional vector, the scatter plot is a two-dimensional plot with the *x*-axis as the first input vector element and the *y*-axis as the second input vector element. The two-dimensional plot is widely used in the QAM application. In the case of an input vector that is not a length two vector, the plot is a one-dimensional plot.

x-y plot
x-y plot is valid only when the input message signal is a length two vector. The simulation will be terminated if the input signal has a dimension other than length two.

The above mentioned plots can be plotted in a same window. You can set up the parameter to activate a plot or eliminate a plot before the simulation. Please note that the eye-pattern plot and the x-y plot are plots recording the trajectories of the message signal. The plot may take a long time to draw these trajectories at every simulation point. In contrast, the scatter plot plots only the point when the decision is made. The simulation is much faster when only scatter plot is activated.

The first input port accepts the message signal. The second input port takes the decision signal. The raising edge of the decision signal is the specified timing for the decision point, which triggers the scatter plot. There is a vertical line in the eye-pattern plot, which indicates the time when the decision is made. The decision pulse is a scalar input signal.

The figure below is a snap shot of the plot created by using this eye-pattern and scatter plot.

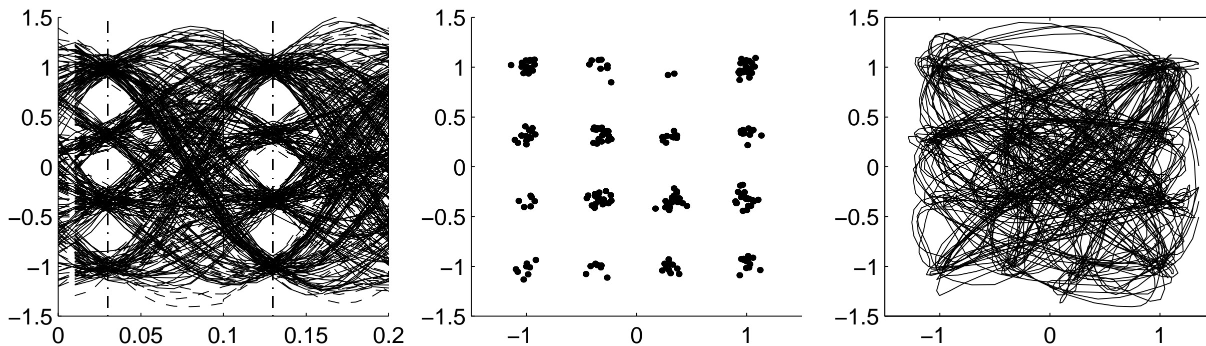


Figure 6-4: Eye-pattern, Scatter and x-y plots

You must specify the x-axis of the eye-pattern plot using the **Eye-pattern plot time frame** parameter. Typically, the time frame should be an integer multiplied by the symbol interval. When this entry is a two-element vector, the second element is the `offset`; the default value of `offset` is 0. The plot takes all plots into the time frame `[offset, offset+time_frame]`. All signals at time `t` are plotted at the time point

```
rem(t-offset, time_fram) in the eye-pattern plot.
```

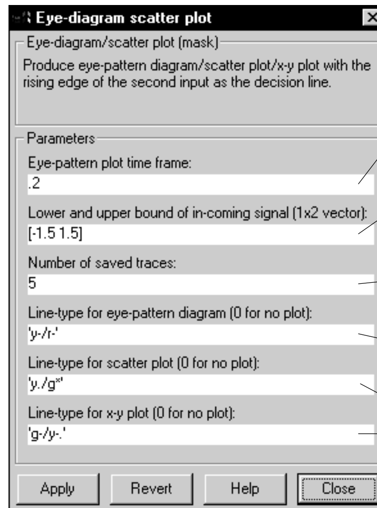
The **Line type for eye-pattern diagram** parameter specifies the color and the line type or plot symbols for eye-pattern plot. A slash (/) is used to separate line types of different elements in the plot vector. You can choose the color and line type code from following table

Color Character	Meaning	Marker-type Character	Meaning	Line-type Character	Meaning
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

When the line type is set to 0, the eye-pattern plot is eliminated.

Please note that the x-y plot can only be used when the input message signal is a two-element vector.

Dialog Box



Specify the *x*-axis of the eye-pattern plot. When this entry is a two-element vector, the second element is the offset value.

A two-element vector. The first element is the lower bound of the input signal, the second the upper bound

An integer that specifies the number of saved trace plots.

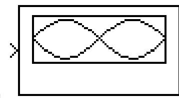
Specify the color, line type, and/or plot symbols for the eye-pattern plot.

The color, line type, and/or plot symbols for the scatter plot and x-y plot respectively. A 0 entry suppresses the plot entirely.

Equivalent M-function

eyescat

Sample Time Eye-Pattern Diagram & Scatter Plot



Category Plotting

Location Source/Sink Sublibrary

Description The Sample-time Eye-Pattern Diagram & Scatter Plot block plots eye-pattern plot, and/or scatter plot based on the parameter specification.

You should read the reference page for the Eye-Pattern Diagram & Scatter Plot block for the detailed description of the eye-pattern plot and scatter plot.

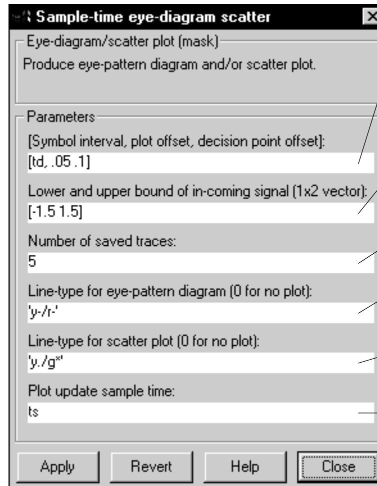
Different from the Eye-Pattern & Scatter Plot block, this block does not provide the x-y plot. This block does not have a dedicated port for the decision timing signal. The decision timing is set inside the parameter instead.

You can specify the color code and line type/marker type in this block. A slash (/) is used to separate type code in the plot parameter. You can choose the color code and line type or marker type code from the following table.

Color Character	Meaning	Marker-type Character	Meaning	Line-type Character	Meaning
y	yellow	.	point	-	solid
m	magenta	o	circle	:	dotted
c	cyan	x	x-mark	-.	dashdot
r	red	+	plus	--	dashed
g	green	*	star		
b	blue	s	square		
w	white	d	diamond		
k	black	v	triangle (down)		
		^	triangle (up)		
		<	triangle (left)		
		>	triangle (right)		
		p	pentagram		
		h	hexagram		

If you like to eliminate the eye-pattern plot or the scatter plot, simply set the line type to be the number 0.

Dialog Box



A three elements vector. The first element is symbol interval, which is the eye-pattern diagram x-axis length. It is also the update interval for the scatter plot. The second element is the offset for the eye-pattern plot. The third element is the decision point offset. It is also the offset for the update of the scatter plot. The unit for all three elements is in second.

A two-element vector. The first element is the lower bound of the input signal, the second the upper bound.

Specify the number of saved trace plots.

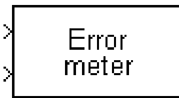
Specify the color, marker or line type for the eye-pattern plot.

Specify the color, marker or line type symbols for the scatter plot.

The sample time for eye-pattern diagram plot update.

Equivalent M-function

eyescat



Error Rate Meter

Catagory Symbol-Error Display

Location Source/Sink Sublibrary

Description The Error Rate Meter block calculates and displays the symbol-error and bit-error rates.

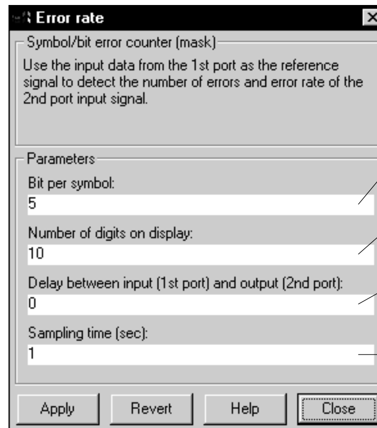
One of the major tasks for communication system simulation is to estimate the symbol-error rate and the bit-error rate of a transmitting system. The Error Rate Meter block provides an error-rate computation and display during the communication system simulation.

The first input port takes the input from the transmitting side. The signal to the first input port must be a scalar. The second input port takes the input from the receivers. The signal to the second port can be a scalar or a vector. In the vector case, the received signal listed in the meter display is named receiver1, receiver2, ... by the order of the position of the signal in the vector. The block counts and then displays the number of symbols transferred and the number of bits transferred. It also makes a comparison between the sending digital signal and the received digital signal at a given sample-time point.

The block can adjust the time delay from the sending side to the receiving side. It assumes that the delay is the same for every receiver when the input for the receiver is a vector. When a difference is found between the sent digit and the received digit, the meter registers an error. At the same time, the bit error is estimated. The symbol-error rate is the total number of symbol errors divided by the total number of symbols transferred. The bit-error rate is the total number of bit error divided by the total number of bits transferred.

A positive integer in the **Number of digits on display** entry forces the meter to display the exact number of rows as specified. The display switches the color between blue and green. When a received digit differs from the sent data, the display on the receiver's line has a red color on display. A 0 or negative number in the **Number of digits on display** entry tells the meter not to display the transferred digital signal.

Dialog Box



Example

This example shows a snap shot of the figure generated by the Error Rate Meter block during a simulation. In the simulation, the input signal to the second input port is a two-element vector signal. There are four bits in each transferred symbol. There are 20 lines on display, which list the most recent 20 digits in the transmitting side and the receiving side. The error rate as well as the sent and received digit are shown in the figure.

Sender	Receiver1	Receiver2
0	3	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	10	2
7	7	7
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	7	6
7	7	7
Symbol Transfered	2428	
Error Number	209	25
Error Rate	0.086079077	0.01029654
Bit Transfered	7284	
Error Number	318	30
Error Rate	0.043657331	0.0041186161

Figure 6-5: Figure Generated by Error Rate Meter in a Simulation

Equivalent M-function

`symerr` and `biterr`

Source Coding

Source coding in communication systems converts arbitrary real-world information to an acceptable representation in communication systems. This section provides some basic techniques as examples of solving the source coding problems using Simulink and MATLAB. This toolbox includes the source coding techniques of signal quantization and differential pulse code modulation (DPCM).

This section also includes compander techniques. Compander is the name for the combination of compressor and expander. Data compression is important for transforming a signals with different power level transformation.

This figure shows the Source Coding Sublibrary:

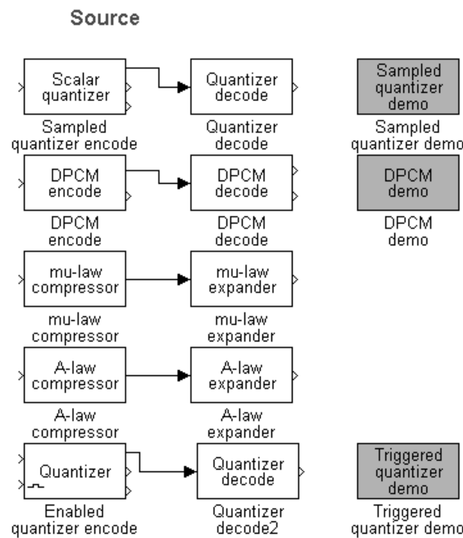


Figure 6-6: Source Coding Sublibrary

Source Coding Reference Table

This table lists the Simulink blocks in the Source Coding Sublibrary. (They are listed alphabetically in this table for your convenience.):

Block Name	Description
A-Law Compressor	Compresses data using an A-law compander
A-Law Expander	Recovers compressed data using an A-law compander
DPCM Decode	Recovers DPCM quantized signals
DPCM Encode	Quantized input data signals
μ-Law Compressor	Compresses data using a μ-law compander
μ-Law Expander	Recovers compressed data using a μ-law compander
Quantization Decode	Recovers signals quantized by the Signal Quantizer or the Triggered Signal Quantizer block
Signal Quantizer	Quantizes an input signal
Triggered Signal Quantizer	Quantizes an input signal when triggered

Signal Quantizer

Scalar quantizer

Catagory Signal Quantization

Location Source Coding Sublibrary

Description The Signal Quantizer block encodes a message signal using scalar quantization. The block uses the finite length of a digit to represent an analog signal. Please refer to chapter 3, the *Tutorial,* for the general principles of quantization computation. Note that you may lose computation accuracy in the quantization processing.

In quantization, the major parameters are **Quantization partition** and **Quantization codebook**. **Quantization partition** is a strict ascending ordered vector, which contains the partition points used in dividing up the input data. **Quantization codebook** is a quantization value vector with length equal to the (length + 1) of the **Quantization partition**. If the input value is less than the `i` th element of **Quantization partition** (and greater than (`i` -1)th element, if any), the quantization value equals to the `i` th element in the **Quantization codebook**.

The figure below shows the quantization process:

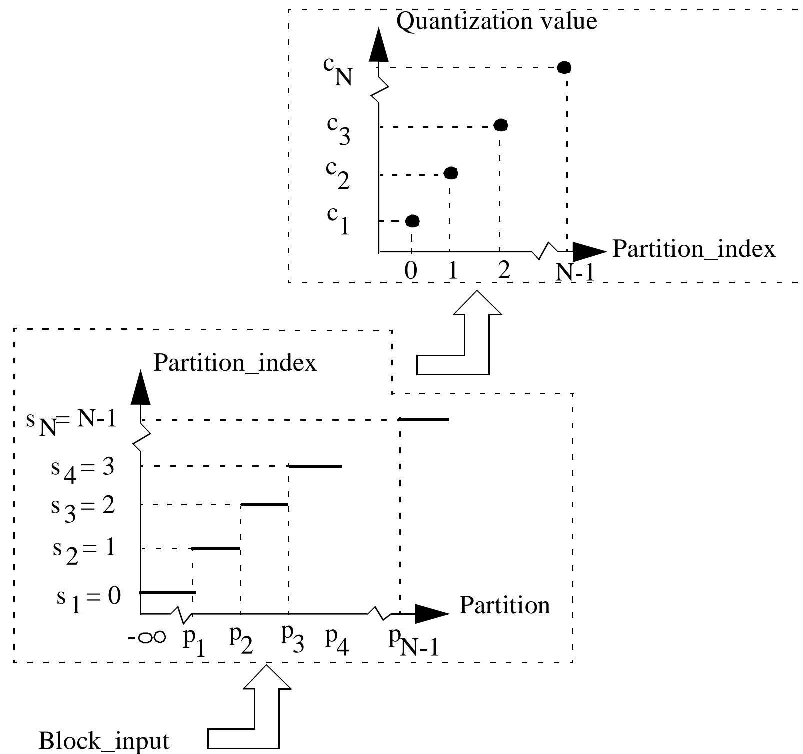
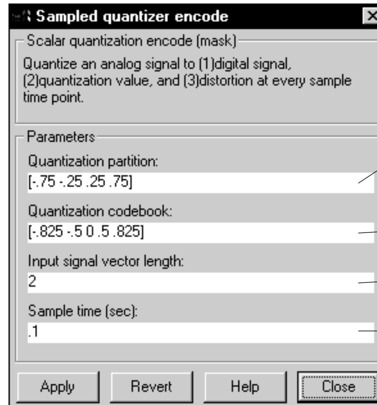


Figure 6-7: Quantization

This block has one input port and three output ports. The input port takes the analog signal. The three output ports output, from top to bottom, the quantization indexFigure 6-7:, the distortion value, and the quantization valueFigure 6-7:. The distortion is a measurement of the quantization error. The vector lengths of all three outputs are equal to the vector length of the input. The quantization block can accept a vector input. When the input is a vector, each output port outputs a vector with the vector length equal to the input vector length. The block processes each element of the vector independently; it performs the quantization at the sample time.

You can use the function lloyds to train the available data to obtain the expected partition and codebook vectors.

Dialog Box



A length N vector, where N is the number of symbols in the symbol set. This must be a strictly ascending ordered vector.

A length N-1 strictly ascending ordered vector.

Specify the length of the input signal.

Specify the sample time. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/3
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Auto
Output Vector Width	Same as the input vector width
Scalar Expansion	N/A
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

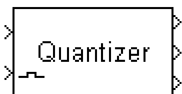
Pair Block

Quantization Decode

Equivalent M-function

quantiz for quantization computation

lloyds for partition and codebook training using the available data



Triggered Signal Quantizer

Catagory Signal Quantization

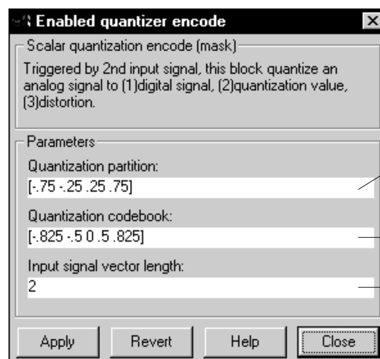
Location Source Coding Sublibrary

Description

The Trigger Signal Quantizer block performs quantization when a trigger signal occurs. This block is similar to the Signal Quantizer block except that the quantization processing is controlled by the second input port of this block, the trigger signal. This block renews its output when the scalar signal from the second input port is a nonzero signal. Please refer to the Signal Quantizer block for a discussion of scalar quantization.

This block has two input ports and three output ports. The quantizer block takes message input from the first input port. It takes the trigger signal from the second input port. The three output ports output quantization index, quantization value, and quantization distortion. When the message input is a vector, the three outputs are also vectors with their vector length equal to the input vector length. Each element in the vector is independently processed.

Dialog Box



A length N vector, where N is the number of partition values. This must be a strictly ascending ordered vector.

A length N-1 strictly ascending ordered vector.

Specify the length of the input signal.

Triggered Signal Quantizer

Characteristics

No. of Input/Outputs	2/3
Vectorized No. 1 Input	Yes
Vectorized No. 2 Input	No
Vectorized Outputs	Yes
No. 1 Input Vector Width	Auto
Output Vector Width	Same as the input vector width
Scalar Expansion	N/A
Time Base	Triggered
States	N/A
Direct feedthrough	Yes

Pair Block

Quantization Decode

Equivalent M-function

`quantiz`

Quantizer decode

Quantization Decode

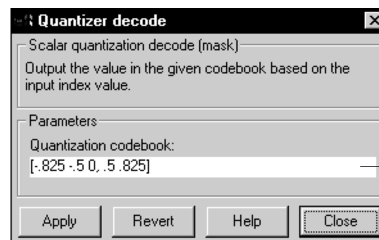
Catagory
Decoding

Location
Source Coding Sublibrary

Description
The Quantization Decode block recovers a message from a quantized signal by finding the quantization value from quantization index. The input of this block is the quantization index, which contains the elements in S = $[s_1, s_2, \ldots s_N]$ = [0, 1, ... N-1]. The output of this block is quantized value, which contains the elements in C = $[c_1, c_2, \ldots, c_N]$. The vector S and C are introduced in the Signal Quantizer block.

This implementation of this block uses a look-up table.

Dialog Box



A length N vector, where N+1 is the number of partitions. The ith element is the quantization output for the (i-1)th quantization index. The default value for the codebook is [-0.825 -0.5 0 0.5 0.825].

Characteristics

No. of Inputs/ Outputs	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Auto
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Blocks
Signal Quantizer, Triggered Signal Quantizer

Equivalent M-function
There is no quantization decode function in this toolbox. For decode computation, use the command:

```
y = codebook(quantiz_index + 1);
```

DPCM Encode

DPCM
encode

Catagory Encoding

Location Source Coding Sublibrary

Description

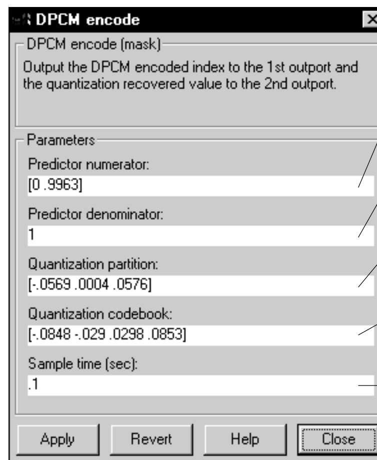
The DPCM (Differential Pulse Code Modulation) Encode block quantizes an input signal. This method uses a predictor to estimate the possible value of the signal at the next step based on the past information into the system. The predictive error is quantized.

This method is specially useful to quantize a signal with a predictable value. This block uses the Signal Quantizer block. Refer to the Signal Quantizer block for a discussion of the codebook and partition concepts.

The predictor in this toolbox is assumed to be a linear predictor. You can use the function `dpcmopt` to train the parameters used in this block: **Predictor numerator, Predictor denominator, Quantization partition**, and **Quantization codebook**. You must input the numerator and denominator of the predictor's transfer function, but the output of `dpcmopt` provides only the numerator. In most DPCM applications, the denominator of predictor transfer function is 1, which means that the predictor is a FIR filter.

When the numerator of the predictor transfer function is a first-order polynomial with the first element (zero-order element) equal to one, the DPCM is a delta modulation.

Dialog Box



Characteristics

No. of Inputs	1/2
Vectorized Inputs/ Outputs	No/No
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

Pair Block

DPCM Decode

Equivalent M-function

`dpcmenco`

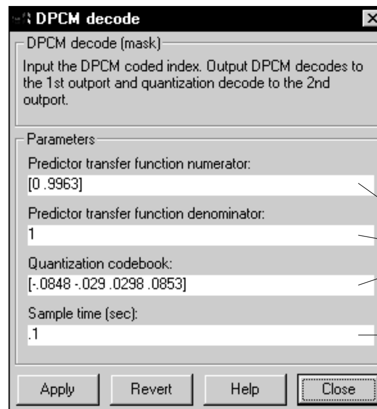
DPCM Decode

Catagory Decoding

Location Source Coding Sublibrary

Description The DPCM (Differential pulse code modulation) Decode block recovers a quantized signal. This block inputs the DPCM encoded index signal and outputs the recovered signal to the first output port and the predictive error to the second output port.

Dialog Box



Match these parameters to the ones used in the corresponding DPCM Encode block.

The calculation sample time. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs	1/2
Vectorized Inputs/ Outputs	No/No
Scalar Expansion	N/A
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

Pair Block DPCM Encode

Equivalent M-function `dpcmdec`

A-Law Compressor

Catagory Data Compression

Location Source Coding Sublibrary

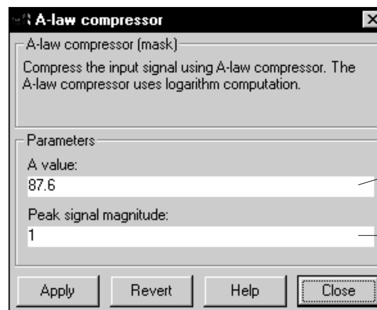
Description

The A-Law Compression block performs data compression. The formula for the A-law compressor is

$$y = \begin{cases} \dfrac{A|x|}{1+\log A}\,\mathrm{sgn}(x) & \text{for } 0 \le |x| \le A/V \\ \dfrac{V(1+\log A|x|/V)}{1+\log A}\,\mathrm{sgn}(x) & \text{for } A/V < |x| \le V \end{cases}$$

The parameters to be specified in the A-law compressor are the A value and the peak magnitude V. The most commonly used A value in practice is 87.6.

Dialog Box



The parameter A in the A-law compressor equation.

Specify the peak value for the input signal. This is the parameter V in the above equation, and the output peak magnitude as well.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block A-Law Expander

Equivalent M-function compand

A-Law Expander

A-law expander

Catagory Data Decompression

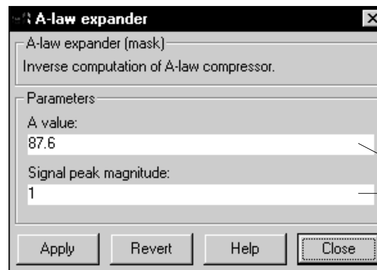
Location Source Coding Sublibrary

Description The A-Law Expander block recovers compressed data. The formula for the A-law expander is the inverse of the compressor function:

$$x = \begin{cases} |y| \dfrac{1 + \log A}{A} \operatorname{sgn}(y) & \text{for } 0 \le |y| \le \dfrac{V}{1 + \log A} \\ e^{|y|(1 + \log A)/V - 1} \dfrac{V}{A} \operatorname{sgn}(y) & \text{for } \dfrac{V}{1 + \log A} < |y| \le V \end{cases}$$

You must specify the A value and the peak magnitude V.

Dialog Box



Match these parameters to the ones used in the corresponding A-Law Compressor block.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block A-Law Compressor

Equivalent M-function compand

mu-law compressor

μ-Law Compressor

Catagory

Data Compression

Location

Source Coding Sublibrary

Description

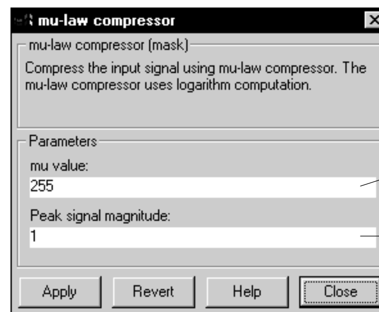
The μ-Law Compressor block performs data compression. The formula for the μ-law compressor is:

$$y = \frac{V\log(1 + \mu|x|/V)}{\log(1 + \mu)}\operatorname{sgn}(x)$$

The parameters to be specified in the μ-law compressor are μ value and the peak magnitude V. The most commonly used μ value in practice is 255.

This block has one input and one output. It takes the x value and outputs the y value described in the above equation.

Dialog Box



The parameter μ in the μ-law compressor equation. The default value is 255.

Specify the peak magnitude of the input signal. This parameter is V in the above equation and the output peak magnitude as well.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block

μ-Law Expander

Equivalent M-function

compand

μ-Law Expander

mu-law expander

Catagory Data Compression

Location Source Coding Sublibrary

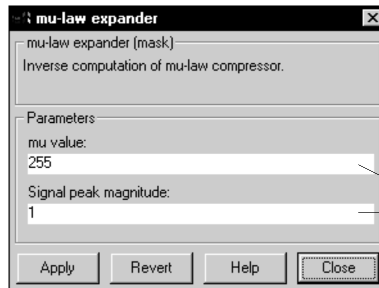
Description The μ-Law Expander block recovers a signal from compressed data. The formula for the μ-law expander is the inverse of the compressor function:

$$x = \frac{V}{\mu}\left(e^{|y|\log(1+\mu)/V} + 1\right)\operatorname{sgn}(y)$$

Same as the μ-law compressor, the parameters to be specified in the μ-law compressor are μ value and the peak magnitude V.

This block takes y as the input and outputs x.

Dialog Box



Match these parameters to the ones used in the corresponding μ-Law Compressor block.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Scalar Expansion	N/A
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block μ-Law Compressor

Equivalent M-function compand

Error-Control Coding

The goal of error-control coding techniques is to reduce the transmission error in a communication system. Error-control coding is an area of very active research in the communications field. This toolbox supports the most commonly used error-control coding techniques.

This toolbox implements two main types of error-control coding techniques: block and convolutional coding. The kinds of block code supported include:

- Hamming code
- BCH code
- Reed-Solomon code
- Cyclic code
- Linear block code

Convolutional code does not break down into different schemes.

There are two main types of decoding used with error-control codes: algebraic and probabilistic. Algebraic decoding is usually used with block codes, and probabilistic decoding is usually used with convolutional codes. For a more detailed discussion of error-control coding and decoding, refer to chapter 3, the *Tutorial*.

This figure shows the Error-Control Coding Sublibrary:Figure 6-8:

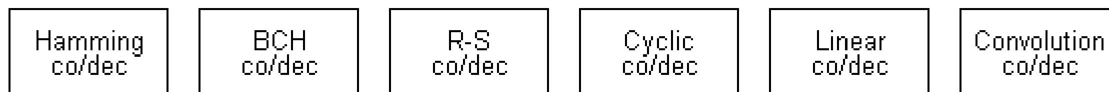


Figure 6-8: Error-Control Coding Sublibrary

Error-Control Coding Reference Table

This table lists the Simulink blocks in the Error-Control sublibrary. (They are listed alphabetically in this table for your convenience.):

Block Name	Description
BCH Code View Table	Generates and displays a table of codeword length, message length, and error-correction capability for BCH codes
BCH Decode Sequence In/Out	Decodes BCH encoded data in the sequential in/out format
BCH Decode Vector In/Out	Decodes BCH encoded data in the vector in/out format
BCH Encode Sequence In/Out	Encodes data using BCH code in the sequential in/out format
BCH Encode Vector In/Out	Encodes data using BCH code in the vector in/out format
Convolutional Decode Sequence In/Out	Decodes convolutionally encoded data in the sequential in/out format
Convolutional Decode Vector In/Out	Decodes convolutionally encoded data in the vector in/out format
Convolutional Encode Sequence In/Out	Encodes data using Convolution code in the sequential in/out format
Convolutional Encode Vector In/Out	Encodes data using Convolution code in the vector in/out format
Cyclic Decode Sequence In/Out	Decodes cyclic encoded data in the sequential in/out format
Cyclic Decode Vector In/Out	Decodes cyclic encoded data in the vector in/out format

Block Name	Description
Cyclic Encode Sequence In/Out	Encodes data using cyclic code in the sequential in/out format
Cyclic Encode Vector In/Out	Encodes data using cyclic code in the vector in/out format
Hamming Decode Sequence In/Out	Decodes Hamming encoded data in the sequential in/out format
Hamming Decode Vector In/Out	Decodes Hamming encoded data in the vector in/out format
Hamming Encode Sequence In/Out	Encodes data using Hamming code in the sequential in/out format
Hamming Encode Vector In/Out	Encodes data using Hamming code in the vector in/out format
Linear Block Decode Sequence In/Out	Decodes linear block encoded data in the sequential in/out format
Linear Block Decode Vector In/Out	Decodes linear block encoded data in the vector in/out format
Linear Block Encode Sequential In/Out	Encodes data using linear block code in the sequential in/out format
Linear Block Encode Vector In/Out	Encodes data using linear block code in the vector in/out format
Reed-Solomon Decode Binary Sequence In/Out	Decodes R-S encoded data in the binary sequential in/out format
Reed-Solomon Decode Binary Vector In/Out	Decodes R-S encoded data in the binary vector in/out format
Reed-Solomon Decode Integer Sequence In/Out	Decodes R-S encoded data in the integer sequential in/out format
Reed-Solomon Decode Integer Vector In/Out	Decodes R-S encoded data in the integer vector in/out format

Block Name	Description
Reed-Solomon Encode Binary Sequence In/Out	Encodes data using R-S code in the binary sequential in/out format
Reed-Solomon Encode Binary VectorIn/Out	Encodes data using R-S code in the binary vector in/out format
Reed-Solomon Encode Integer Sequence In/Out	Encodes data using R-S code in the integer sequential in/out format
Reed-Solomon Encode Integer Vector In/Out	Encodes data using R-S code in the integer vector in/out format

Hamming Encode Vector In/Out

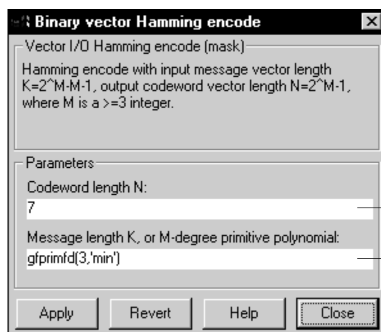
Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Hamming Encode Vector In/Out block encodes a binary message vector into a binary codeword vector. Hamming encoding is a form of linear encoding in which message and codeword lengths are fixed. The codeword length (N) is 2^M-1 and the message length (K) is 2^M-M-1, where M is an integer greater than or equal to 3. Hamming code is a single error correction code.

You can use the `hammgen` function to produce the Hamming code generator matrix and parity-check matrix parameters.

Dialog Box



Codeword length (N).

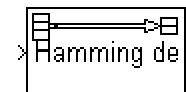
Message length (K). When this entry is a vector, it represents a primitive polynomial in the GF(2) field. You can construct a primitive polynomial

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Row Number of Generator Matrix
Output Vector Width	Column Number of Generator Matrix
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block Hamming Decode Vector In/Out

Hamming Decode Vector In/Out

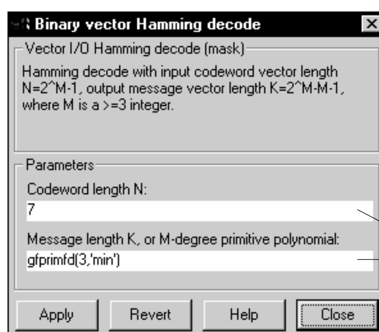


Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description The Hamming Decode Vector In/Out block recovers a message vector from a codeword vector using Hamming code. The input parameters for this block must match the parameters used in the Hamming Encode block in order to obtain the correct result.

Dialog Box

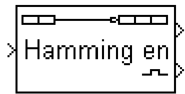


Match the codeword length N and the message length K to the corresponding entries in the Hamming Encode block.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Row Number of Generator Matrix
Output Vector Width	Column Number of Generator Matrix
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block Hamming Encode Vector In/Out



Hamming Encode Sequence In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Hamming Encode Sequence In/Out block encodes a binary message into a binary codeword using a sequential in/out format. Hamming coding is a special case of linear block coding where the message and codeword lengths are fixed. Hamming code is a single error correction code. In the sequential format, the block holds the input message bits in a length K buffer. When the buffer is loaded, the encoding process begins. The block outputs the length N codeword in a bit stream.

The codeword length is always $N=2^M-1$, where M is an integer greater or equal to 3. The message length is $K=2^M-M-1$. You can use the `hammgen` function to construct the Hamming code generator matrix and parity-check matrix parameters.

The relation between the output and input sample times is:

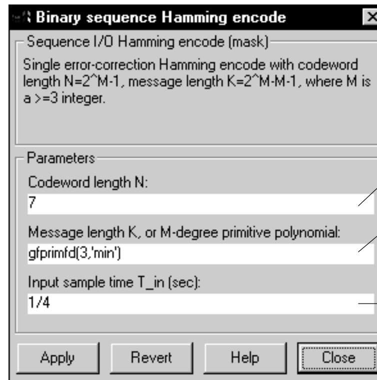
```
output_sample_time=input_sample_time*K/N
```

Because encoding cannot begin until the input buffer is full, this encoding method introduces a delay between the arrival of the first message signal and the outputting of the first codeword. The time delay is:

```
K*Input_sample_time
```

Output port 1 outputs the codeword in binary sequential form. Output port 2 outputs an impulse, that is, it outputs “1” when the output register is full and “0” at all other times.

Dialog Box



Codeword length (N)

Message length. When this entry is a vector, it represents a primitive polynomial in the GF(2) field. You can construct primitive polynomials using the MATLAB M-function `gfprimfd`.

The time interval for each input bit in the sequence.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block

Hamming Decode Sequence In/Out

Hamming Decode Sequence In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description

The Hamming Decode Sequence In/Out block recovers a binary message vector from a binary codeword vector using Hamming decoding technique. The input parameters for this block must match the parameters used in the Hamming Encode block in order to obtain the correct result. In the sequential format, the block holds the input codeword bits in a length N buffer. When the buffer is loaded, the decoding process begins. The block outputs the length K recovered message in a bit stream.

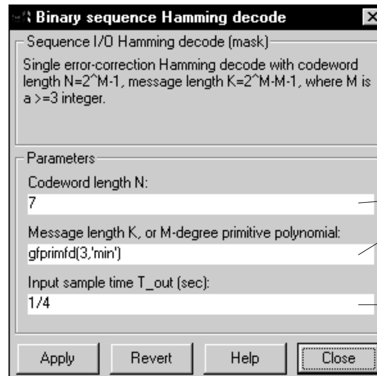
The relation between the input and output sample times is:

```
input_sample_time = output_sample_time*N/K
```

This block introduces a delay between the arrival of the first codeword signal and the outputting of the first recovered message signal. The time delay length is:

```
K*Output_Sample_Time
```

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Hamming Encode block.

The time interval for each input bit in the sequence. The output sample time equals the `input_sample_time*K/N`.

Hamming Decode Sequence In/Out

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time * K

Pair Block

Hamming Encode Sequence In/Out

BCH code view table

BCH Code View Table

Catagory Create a view table for BCH parameters

Location Error-Control Coding Sublibrary

Description The BCH Code View Table block creates a table of codeword length, message length, and error-correction capability for BCH codes. The BCH codeword is of length 2^M-1, where M is a positive integer greater than or equal to 3. The message length depends on the length of the cosets in the Galois field GF(2^M). The error-correction capability is more complicated; there is no closed-formula for deriving the relationship between error-correction capability and the codeword and message lengths.

This block provides a view table for codeword lengths 2^M-1, where M ranges from 3 through 9. The figure on the following page shows the view table as it appears in the BCH Code View Table block. In this table, N is the codeword length, K the valid message length determined by the calculation of cosets, and T is the error-correction capability for the given N and K values. T is determined by an exhaustive search.

BCH Code View Table

N	K	T	N	K	T	N	K	T
7	4	1	255	199	7	511	358	18
15	11	1		191	8		349	19
	7	2		187	9		340	20
	5	3		179	10		331	21
31	26	1		171	11		322	22
	21	2		163	12		313	23
	16	3		155	13		304	25
	11	5		147	14		295	26
	6	7		139	15		286	27
63	57	1		131	18		277	28
	51	2		123	19		268	29
	45	3		115	21		259	30
	39	4		107	22		250	31
	36	5		99	23		241	36
	30	6		91	25		238	37
	24	7		87	26		229	38
	18	10		79	27		220	39
	16	11		71	29		211	41
	10	13		63	30		202	42
	7	15		55	31		193	43
127	120	1		47	42		184	45
	113	2		45	43		175	46
	106	3		37	45		166	47
	99	4		29	47		157	51
	92	5		21	55		148	53
	85	6		13	59		139	54
	78	7		9	63		130	55
	71	9	511	502	1		121	58
	64	10		493	2		112	59
	57	11		484	3		103	61
	50	13		475	4		94	62
	43	14		466	5		85	63
	36	15		457	6		76	85
	29	21		448	7		67	87
	22	23		439	8		58	91
	15	27		430	9		49	93
	8	31		421	10		40	95
255	247	1		412	11		31	109
	239	2		403	12		28	111
	231	3		394	13		19	119
	223	4		385	14		10	121
	215	5		376	15			
	207	6		367	16			

N: code word length; K: message length; T: error-correction capability

Figure 6-9: BCH View Table

For values of M larger than 9, use the bchpoly function.

Equivalent M-function

`bchpoly`

BCH Encode Vector In/Out

Catagory Error-control encoder

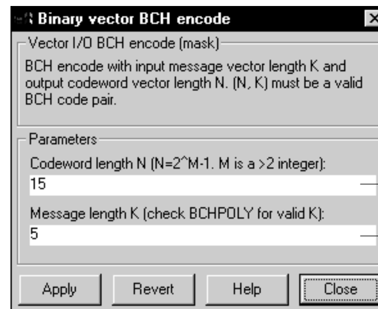
Location Error-Control Coding Sublibrary

Description

The BCH Encode Vector In/Out block encodes a message vector into a codeword vector. BCH coding is a widely used error control coding method for correcting the error induced by random noise in the transmission channel.

The BCH codeword length (N) is 2^M-1, where M is a positive integer greater than or equal to 3. The message length (K) depends upon the size of the cosets of GF(2^M). There is currently no analytic formula to derive the relationship between the error-correction capability and the codeword and message lengths. You can use the BCH Code View Table block or bchpol y function to calculate the error-correction capability and possible message lengths for codeword lengths 3 through 9.

Dialog Box



Codeword length (N).

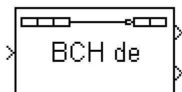
Message length (K) must be less than the codeword length.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Input Vector Width	Message Length
Output Vector Width	Codeword Length
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block BCH Decode Vector In/Out

Equivalent M-function encode



BCH Decode Vector In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description

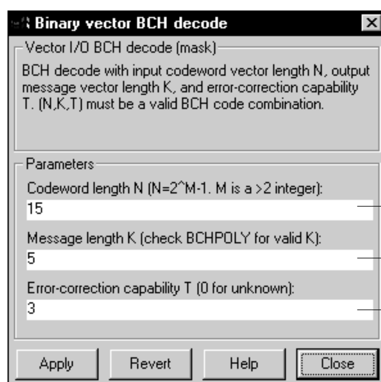
The BCH Decode Vector In/Out block recovers a binary message vector from a codeword vector using BCH coding techniques. The input parameters for this block must match the parameters used in the BCH Encode block in order to obtain the correct result.

Input port 1 accepts the codeword vector and input port 2 accepts a computation flag signal. This signal controls the decode computation. A nonzero input on this port enables the BCH decoding and a zero input disables BCH decoding. This control is necessary because BCH decoding is computationally intensive.

Output port 1 outputs the recovered message. Output port 2 outputs the number of errors detected during the recovery of the message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the selected BCH coding scheme.

Use the BCH Code View Table block or `bchpoly` function to calculate the error-correction capability.

Dialog Box



Codeword length (N).

Message length (K).

Error-correction capability. If this parameter is unknown, entering a 0 forces the block to calculate the error-control capability at initialization.

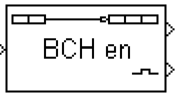
BCH Decode Vector In/Out

Characteristics

No. of Inputs/Outputs	2/2
Vectorized No. 1 Input	Yes
No. 1 Input Vector Length	No
Vectorized No. 2 Input	No
Vectorized No. 1 Output	Yes
No. 1 Output Vector Length	K
Vectorized No. 2 Output	No
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block BCH Encode Vector In/Out

Equivalent M-function decode



BCH Encode Sequence In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The BCH Encode Sequence In/Out block encodes a binary message vector into a codeword using a sequence in/out format. BCH coding is a widely used error control coding method for correcting the error induced by random noise in the transmission channel. In the sequential format, the block holds the input message bits in a length K buffer. When the buffer is loaded, the encoding process begins. The block outputs the length N codeword in a bit stream.

The BCH codeword length (N) is 2^M-1, where M is a positive integer greater than or equal to 3. The message length (K) depends upon the size of the cosets of GF(2^M). There is currently no analytic formula to derive the relationship between the error-correction capability and the codeword and message lengths.

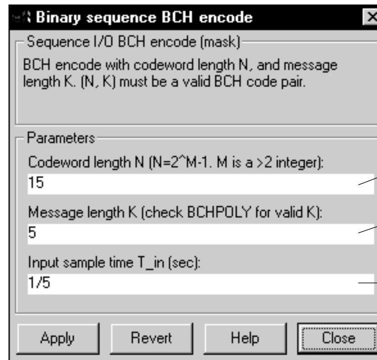
Please refer to the BCH Code View Table block for a list of error correction capabilities and possible message lengths for codeword lengths 3 through 9. You can use the function `bchpoly` to find the codeword, message length and error-correction capability suitable to your application.

The relationship between the output and input sample times is:

```
output_sample_time = input_sample_time*K/N
```

Output port 1 outputs the codeword in binary sequential form. Output port 2 outputs an impulse, that is, it outputs a “1” at the when the input register is full and a “0” at all other times.

Dialog Box



Characteristics

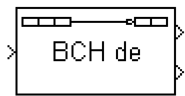
No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time * K

Pair Block

BCH Decode Sequence In/Out

Equivalent M-function

encode



BCH Decode Sequence In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description The BCH Decode Sequence In/Out block recovers a binary message from a codeword using a sequential in/out format. The input parameters for this block must match the parameters used in the BCH encode block to obtain the correct result. In the sequential format, the block holds the input codeword bits in a length N buffer. When the buffer is loaded, the encoding process begins. The block outputs the length K recovered message in a bit stream.

The input port accepts the codeword vector. Output port 1 outputs the recovered message, and output port 2 outputs the number of errors detected during the recovery of the message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the selected BCH coding scheme.

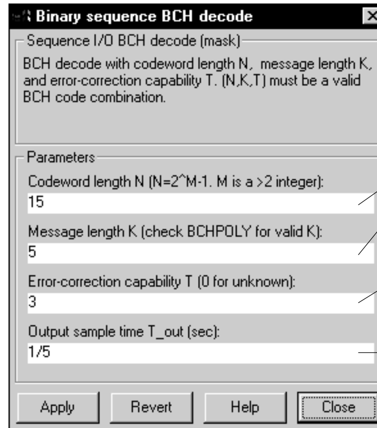
The relationship between the input and output sample times is:

```
input_sample_time = output_sample_time*K/N
```

Because decoding cannot begin until the input buffer is full, this block introduces a time delay. The delay between the arrival of the first codeword and the output of the first recovered message is:

```
K*output_sample_time
```

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the BCH Encode block.

Error-correction capability. If this parameter is unknown, entering a 0 forces the block to calculate the error-control capability at initialization.

The time interval for each input bit in the sequence. The output sample time equals the `input_sample_time*K/N`.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block

BCH Decode Sequence In/Out

Equivalent M-function

encode

Reed-Solomon Encode Integer Vector In/Out

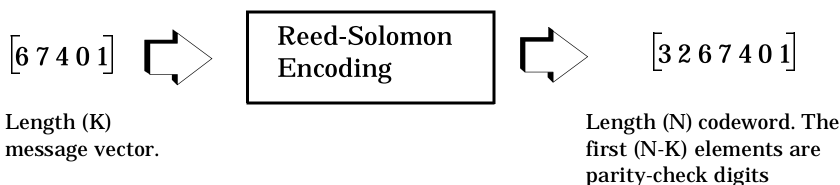
Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Encode Integer Vector In/Out block encodes an integer vector message into a codeword vector using an integer vector in/out format. Reed-Solomon (R-S) code is a burst-error correction code. Please refer to Chapter 3, the *Tutorial*, or reference [28] for a discussion of R-S coding techniques.

The elements in R-S coding are in GF(2^M), a finite field that contains 2^M elements, where M is an integer greater than or equal to 3. This block supports integer signals that range in value from 0 to 2^M-1. An (N,K) Reed-Solomon code has codeword length (N) equal to 2^M-1, message length (K), and parity-check length (N-K). For an efficient Reed-Solomon code, N-K should be an even number.

For example, if M=3, the range of allowed values is 0-7. For a message vector (K) of length 5, the codeword (N) is of length 7 and the number of parity-check digits (N-K) is 2:



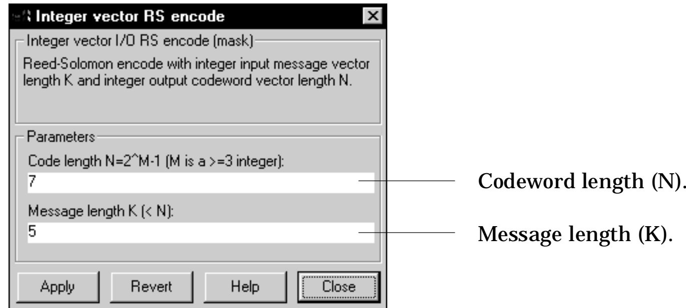
The (N,K) Reed-Solomon coding scheme is an (N∗M,K∗M) bitwise counting, and its error correction capability is:

```
T = floor((N-K)/2)
```

The corrections made in the Reed-Solomon code are integer corrections, which means that every correction is M-bit.

Input port 1 accepts the input message, which must be a length K integer signal. Input port 2 accepts the computation flag signal, which controls the decode computation. A nonzero scalar input to this port enables the R-S encoding and a 0 input disables R-S encoding. This control is necessary because R-S encoding is computationally intensive.

Dialog Box



Characteristics

No. of Inputs/Outputs	2/1
Vectorized No. 1 Input	Yes
No. 1 Input Vector Length	N
Vectorized No. 2 Input	No
Vectorized Outputs	Yes
Output Vector Length	K
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block

Reed-Solomon Encode Integer Vector In/Out

Equivalent M-function

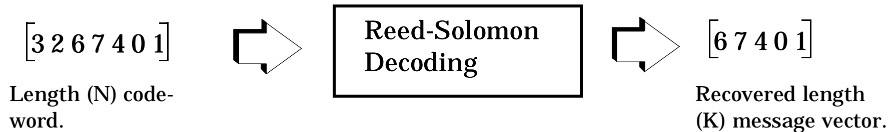
encode

Reed-Solomon Decode Integer Vector In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

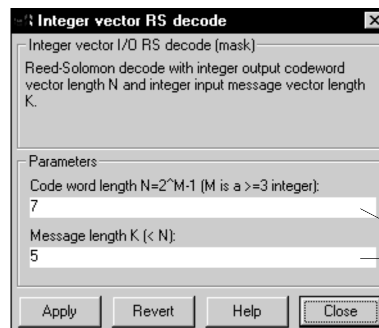
Description The Reed-Solomon Decode Integer Vector In/Out block recovers a message vector from a codeword using an integer vector in/out format. Please read the discussion of Reed-Solomon (R-S) code in Chapter 3, the *Tutorial*, for details on this coding method. The input parameters for this block must match the parameters used in the Reed-Solomon Encode block in order to obtain the correct result.



Input port 1 accepts a length N integer codeword. The second accepts the computation flag signal, which controls the decode computation. A nonzero scalar input to this port enables the R-S decoding and a 0 input disables R-S decoding. This control is necessary because R-S decoding is computationally intensive.

Output port 1 outputs the recovered length K integer vector message. Output port 2 outputs the number of errors detected in the decoded message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the R-S coding scheme.

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Reed-Solomon Encode block.

Reed-Solomon Decode Integer Vector In/Out

Characteristics

No. of Inputs/Outputs	2/2
Vectorized No. 1 Input	Yes
No. 1 Input Vector Length	N
Vectorized No. 2 Input	No
Vectorized Outputs	Yes
Output Vector Length	K
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block

Reed-Solomon Encode Integer Vector In/Out

Equivalent M-function

decode

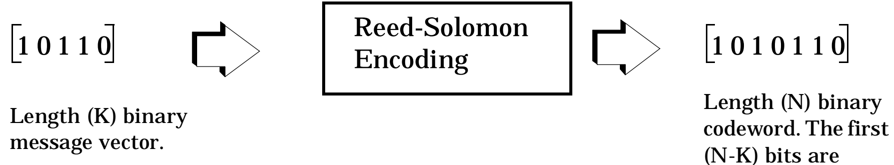
Reed-Solomon Encode Binary Vector In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Encode Binary Vector In/Out block encodes a length K binary vector into length N codeword. Unlike any other coding method introduced in this toolbox, Reed-Solomon code is a burst-error correction code. Please refer to Chapter 3, the *Tutorial*, for a discussion of Reed-Solomon coding techniques. There is also a detailed discussion in reference [28].

In the binary vector format, all input/output signals are vectors with binary elements. The elements in Reed-Solomon (R-S) coding are in GF(2^M), a finite field that contains 2^M elements, where M is an integer greater than or equal to 3. An (N,K) Reed-Solomon code has codeword length N=2^M-1 and message length K that is less than N. For efficient coding the parity-check length (N-K) should be an even integer. For example, if M=3, the codeword length (N) is 7, and to make the parity-check length (N-K) even length (K) can be set to 5:



The (N,K) Reed-Solomon coding scheme is an (N∗M,K∗M) code when counted bitwise. The error correction capability of Reed-Solomon code is:

```
T = floor((N-K)/2)
```

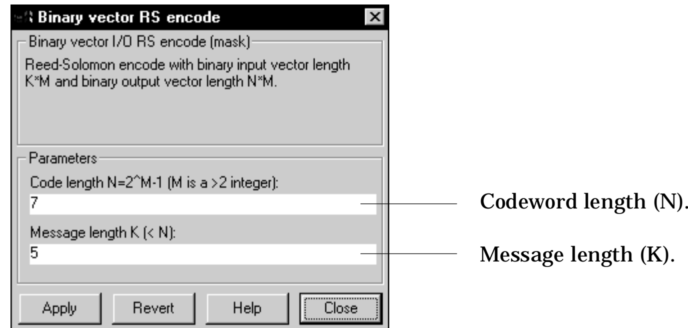
The corrections made in the Reed-Solomon code are integer corrections, which means that every correction is M-bit.

This block encodes a length M∗K binary message into a length M∗N binary codeword. The R-S encode block is length N∗M, the output signal is length K∗M.

Reed-Solomon Encode Binary Vector In/Out

Input port 1 accepts the length N∗K message signal. Input port 2 accepts the computation flag signal that controls the encode computation. A nonzero scalar input to this port enables the R-S encoding and a 0 input disables R-S encoding. This control is necessary because R-S decoding is computationally intensive.

Dialog Box



Characteristics

No. of Inputs/Outputs	2/1
Vectorized No. 1 Input	Yes
No. 1 Input Vector Length	K∗M
Vectorized No. 2 Input	No
Vectorized Outputs	Yes
Output Vector Length	N∗M
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block

Reed-Solomon Decode Binary Vector In/Out

Equivalent M-function

encode

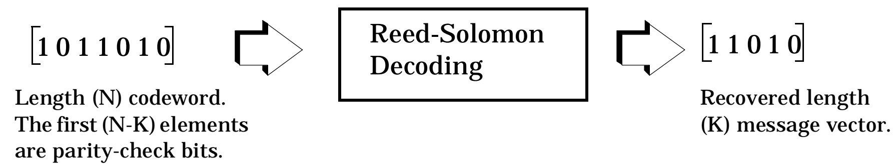
Reed-Solomon Decode Binary Vector In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Decode Binary Vector In/Out block recovers a binary message vector from a binary codeword vector. Please read the discussion of Reed-Solomon (R-S) code in Chapter 3, the *Tutorial*, for details on this coding method. The input parameters for this block must match the parameters used in the Reed-Solomon Encode block in order to obtain the correct result.

This block accepts length N binary vector codewords. Length (N) is equal to 2M-1, where M is an integer greater than or equal to 3. The recovered message length (K) is less than N, typically an odd integer:

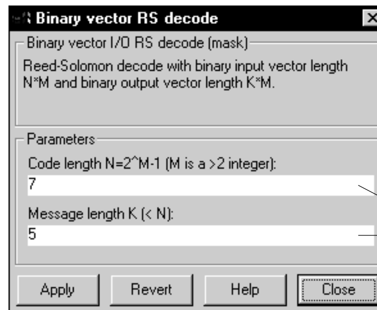


Input port 1 accepts the length N binary codeword. Input port 2 accepts the computation flag signal, which controls the decode computation. A nonzero scalar input to this port enables the R-S decoding and a 0 input disables R-S decoding. This control is necessary because R-S decoding is computationally intensive.

Output port 1 outputs the recovered length K binary vector message. Output port 2 outputs the number of errors detected in the decoded message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the R-S coding scheme.

Reed-Solomon Decode Binary Vector In/Out

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Reed-Solomon Encode block.

Characteristics

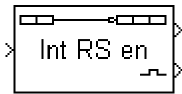
No. of Inputs/Outputs	2/2
Vectorized No. 1 Input	Yes
No. 1 Input Vector Length	K∗M
Vectorized No. 2 Input	No
Vectorized Outputs	Yes
Output Vector Length	N∗M
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block

R-S Encode Binary Vector In/Out

Equivalent M-function

decode



Reed-Solomon Encode Integer Sequence In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Encode Integer Sequence In/Out block encodes an integer message into a codeword using an integer sequential in/out format. Reed-Solomon (R-S) code is a burst-error correction code. Please refer to Chapter 3, the *Tutorial*, or reference [28] for a discussion of Reed-Solomon coding techniques. In the integer sequential format, the block holds the input message integers in a length K integer buffer. When the buffer is loaded, the encoding process begins. The block outputs the integer length N codeword one integer at a time.

The elements in Reed-Solomon coding are in GF(2^M), a finite field that contains 2^M elements. M is an integer greater than or equal to 3. An (N,K) Reed-Solomon code has codeword length N=2^M-1 and message length K. The elements in the codeword and message are from GF(2^M). The (N,K) Reed-Solomon coding scheme is an (N*M,K*M) bitwise counting. The error-correction capability of Reed-Solomon code is:

```
T = floor((N-K)/2)
```

For an efficient Reed-Solomon code, N-K should be an even number. The corrections made in the Reed-Solomon code are integer corrections, which means that every correction is M-bit.

The codeword output sample time is different from the input sample time. The relationship between the output and input sample times is:

```
output_sample_time=input_sample_time*K/N
```

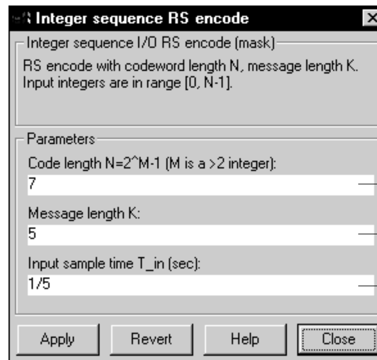
Output port 1 outputs the codeword in integer sequential form. Output port 2 outputs an impulse, that is, it outputs a “1” at the onset of the encoding process and a “0” at all other times.

Because the encoding process cannot begin until the input integer buffer is full, this block introduces a delay between the arrival of the first message signal and the output of the first codeword. The time delay is:

```
K*input_sample_time
```

Reed-Solomon Encode Integer Sequence In/Out

Dialog Box



Characteristics

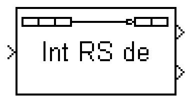
No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block

Reed-Solomon Integer Sequence In/Out

Equivalent M-function

encode



Reed-Solomon Decode Integer Sequence In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description

The Reed-Solomon Decode Integer Sequence In/Out block recovers a message from a codeword using an integer sequential in/out format. Please read the discussion of Reed-Solomon (R-S) code in Chapter 3, the *Tutorial*, for details on this coding method. The input parameters for this block must match the parameters used in the Reed-Solomon Encode block in order to obtain the correct result.

In the sequential format, the block holds the input integer codeword in a length N integer buffer. When the buffer is loaded, the decoding process begins. The block outputs the length K recovered message one integer at a time.

Output port 1 outputs the recovered length K integer sequential message. Output port 2 outputs the number of errors detected in the decoded message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the R-S coding scheme.

This block requires information about the output sample time. The relationship between the input and output sample times is:

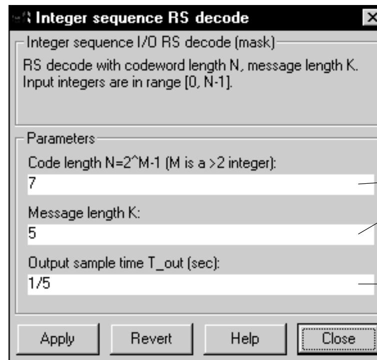
```
input_sample_time=output_sample_time*K/N
```

Because the decoding process cannot begin until the integer input buffer is full, the block introduces a delay between the arrival of the first codeword and the output of the first recovered message. This delay is:

```
K*output_sample_time
```

Reed-Solomon Decode Integer Sequence In/Out

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Reed-Solomon Encode block.

The time interval for each input in the sequence. The output sample time equals the `input_sample_time*K/N`.

Characteristics

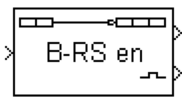
No. of Inputs/Outputs	1/2
Vectorized Input/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time * K

Pair Block

Reed-Solomon Integer Sequence In/Out

Equivalent M-function

decode



Reed-Solomon Encode Binary Sequence In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Encode Binary Sequence In/Out block encodes a message into a codeword using a binary sequential in/out format. Reed-Solomon (R-S) code is a burst-error correction code. Please refer to the *Communications Fundamentals* chapter or reference [28] for a discussion of Reed-Solomon coding techniques.

The elements in Reed-Solomon coding are in GF(2^M), a finite field that contains 2^M elements. M is an integer greater than or equal to 3. An (N,K) Reed-Solomon code has codeword length N=2^M-1 and message length K, where K is an integer less than M. Since the error correction capability of R-S coding is:

```
floor((N-K)/2)
```

it is more efficient to choose K so that (N-K) is an even number.

An (N,K) R-S code refers to the codeword length (N) and message length (K) in integer form. This means that each element of the codeword and message is M-bit, and that the code is actually (N∗M,K∗M) when counted bitwise.

The corrections made in the Reed-Solomon code are integer corrections, which means that every correction is M-bit.The codeword output sample time is different from the input sample time. The relationship between the two is:

```
output_sample_time=input_sample_time*K/N
```

Output port 1 outputs the codeword in integer sequential form. Output port 2 outputs an impulse, that is, it outputs a “1” at the onset of the encoding process and a “0” at all other times.

This block introduces a delay between the arrival of the first message signal and the output of the first codeword. The time delay is:

```
M*K*input_sample_time
```

The sequential in/out format breaks down into five steps:

Reed-Solomon Encode Binary Sequence In/Out

(1) 1 1 0 0 1 0 1 1 ... ⇨ [| | | | | | | | | | | |1|1]

Bits arrive sequentially and are loaded into a length K∗M register. Here, K=5 and M=3.

(2) ... [1|0|1|1] ⇨ [1 1 1 0 0 0 0 1 1 0 0 1 0 1 1]

The block converts the loaded information into a length K∗M vector.

(3) Message vector ⇨ Reed-Solomon Encoding ⇨

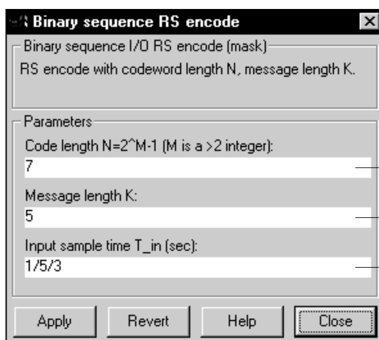
A length N∗M codeword vector. Here N=7, making the length of the codeword equal to 21.

(4) The block loads the length N∗M codeword vector into an output register. ⇨ [1|0|0|1|0|1|1|1|0|0|1|0|1|1|1|0|0|1|0|1|1]

(5) ... [1|0|0|1|0|1|1] ⇨ 1 0 0 1 0 1 1...

The block outputs the bits sequentially, first-in, first-out.

Dialog Box



Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ M ∗ K

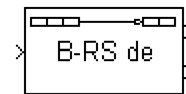
Pair Block

Reed-Solomon Decode Binary Sequence In/Out

Equivalent M-function

encode

Reed-Solomon Decode Binary Sequence In/Out



Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description The Reed-Solomon Decode Binary Sequence In/Out block recovers a message from a codeword using a binary sequential in/out format. Please read the discussion of Reed-Solomon (R-S) code in Chapter 3, the *Tutorial*, for details on this coding method. The input parameters for this block must match the parameters used in the Reed-Solomon Encode block in order to obtain the correct result.

Output port 1 outputs the recovered length K binary sequential message. Output port 2 outputs the number of errors detected in the decoded message. A negative integer indicates that the number of errors detected exceeds the error-correction capability of the R-S coding scheme.

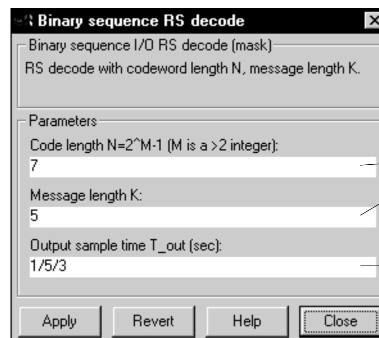
This block requires information about the output sample time. The relationship between the input and output sample times is:

```
input_sample_time = output_sample_time*K/N
```

The block introduces a delay between the arrival of the first codeword and the output of the first recovered message. This delay is:

```
K*M*output_sample_time
```

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Reed-Solomon Encode block.

The time interval for each input bit in the sequence.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ M ∗ K

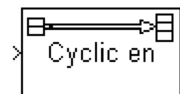
Pair Block

Reed-Solomon Encode Binary Sequence In/Out

Equivalent M-function

decode

Cyclic Encode Vector In/Out



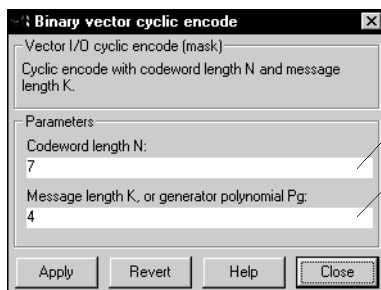
Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Cyclic Encode Vector In/Out block encodes a binary message into a codeword. Cyclic coding is a generic coding technique that requires the use of a cyclic generator polynomial. You can create a cyclic generator polynomial with the `cyclpoly` function. The Cyclic Encode and Cyclic Decode blocks use systematic cyclic coding.

The codeword vector length N must be 2^M-1, where M is an integer greater than or equal to 3. The message vector length K must be less than the codeword length N. The degree of the cyclic polynomial is N-K.

Dialog Box



Codeword length (N).

Generator polynomial defined in the GF(2) field. The degree of the polynomial must be less than the codeword length N. When this entry is a scalar, this block assumes that the entered value is the message length K. The block then generates a default cyclic polynomial using the function `pr=cyclpoly(N,K,'min')`.

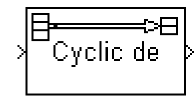
Characteristic

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Input Vector Width	Message Length
Output Vector Width	Codeword Length
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block Cyclic Decode Vector In/Out

Equivalent M-function encode

Cyclic Decode Vector In/Out

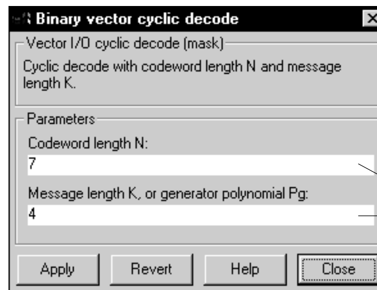


Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description The Cyclic Decode Vector In/Out block recovers a binary message vector from a binary codeword vector using cyclic decoding. The input parameters for this block must match the parameters used in the Cyclic Encode block to obtain the correct result.

Dialog Box

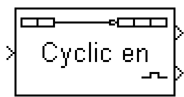


Match the codeword length N and the message length K to the corresponding entries in the Cyclic Encode block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Input Vector Width	Message Length
Output Vector Width	Codeword Length
Time Base	Auto
States	N/A
Direct Feedthrough	YesPair Block
Cyclic Encode Vector In/Out	

Equivalent M-function decode



Cyclic Encode Sequence In/Out

Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Cyclic Encode Sequence In/Out block encodes a binary message into a codeword using a sequential in/out format. Cyclic coding is a generic coding method that requires the use of a cyclic generator polynomial. You can use the function `cyclpoly` to generate a proper cyclic polynomial. Both the Cyclic Encode and Cyclic Decode blocks use systematic cyclic coding. In the sequential format, the block holds the input message bits in a length K buffer. When the buffer is loaded, the encoding process begins. The block outputs the length N codeword in a bit stream.

The codeword vector length N must be 2^M-1, where M is an integer greater than or equal to 3. The message vector length K must be less than the codeword length N. The degree of the cyclic polynomial is N-K.

The relationship between the output and input sample times is:

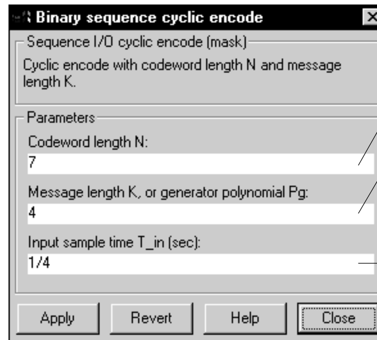
```
output_sample_time=input_sample_time*K/N
```

Output port 1 outputs the codeword in binary sequential form. Output port 2 outputs an impulse, that is, it outputs “1” when the input register is full and “0” at all other times.

Because encoding cannot begin until the input buffer is full, this encoding method introduces a delay between the arrival of the first message signal and the output of the first codeword. The time delay is:

```
K*Input_sample_time.
```

Dialog Box



Codeword length (N).

Generator polynomial defined in the GF(2) field. The degree of the polynomial must be less than the codeword length N. When this entry is a scalar, cyclic encode assumes that the entered value is the message length K. The block then generates a default cyclic polynomial using the function `pr=cyclpoly(N,K,'min')`.

The time interval for each input bit in the sequence. The output sample time equals the `input_sample_time*K/N`.

Characteristics

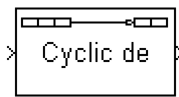
No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-Time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block

Cyclic Decode Sequence In/Out

Equivalent M-function

encode



Cyclic Decode Sequence In/Out

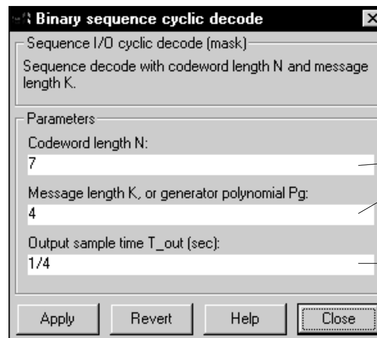
Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description

The Cyclic Decode Sequence In/Out block recovers a binary message from a binary codeword using a sequential in/out format. The input parameters for this block must match the parameters used in the Cyclic Encode block to obtain the correct result. In the sequential format, the block holds the input codeword bits in a length N buffer. When the buffer is loaded, the decoding process begins. The block outputs the length K recovered message in a bit stream.

Dialog Box



Match the codeword length N and the message length K to the corresponding entries in the Cyclic Encode block.

The time interval for each input bit in the sequence. The output sample time equals the `input_sample_time*K/N`

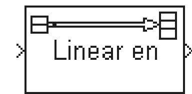
Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time $*$ K

Pair Block Cyclic Encode Sequence In/Out

Equivalent M-function decode

Linear Block Encode Vector In/Out

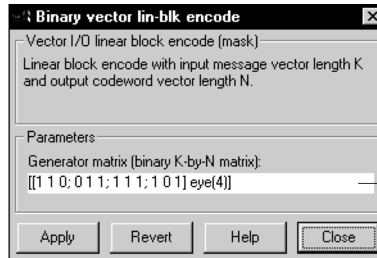


Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Linear Block Encode Vector In/Out block uses linear block encoding to generate a binary codeword vector from a binary message segment. The input message segment can be either a binary vector or a binary sequence. Its length is equal to the number of rows (K) of the generator matrix. The number of elements in the output codeword equals the number of columns (N) in the generator matrix. You can produce an appropriate generator matrix using the `hammgen` or `cyclgen` function. The corresponding decode block requires both this generator matrix and an error correction truth table to decode the output of this block.

Dialog Box



A K-by-N matrix, where K is the message length and the input vector length, and N is the codeword length.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Row Number of Generator Matrix
Output Vector Width	Column Number of Generator Matrix
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block Linear Block Decode Vector In/Out

Linear Block Decode Vector In/Out

Catagory Error-control decoder

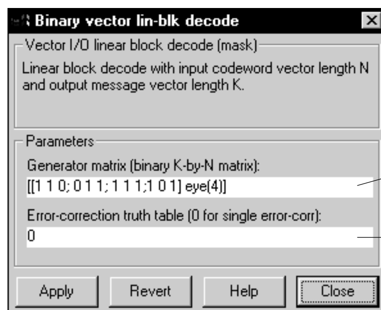
Location Error-Control Coding Sublibrary

Description

The Linear Block Decode Vector In/Out block recovers a binary message vector from a binary codeword vector by using linear block decoding.

Linear block code is a generic coding method that requires the specification of a generator matrix. This block uses the same generator matrix specified in the corresponding encoding block, as well as an error-correction truth table. Generate the truth table using `htruthtb`.

Dialog Box



Generator matrix specified in Linear Block Encode block.

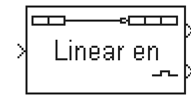
The logic circuit that determines error bit location. The table is a 2^{N-K}-by-N matrix. When 0 is entered, the error-control code defaults to a systematic single error-correction code.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	Yes/Yes
Input Vector Width	Row Number of Generator Matrix
Output Vector Width	Column Number of Generator Matrix
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block Linear Block Encode Vector In/Out

Linear Block Encode Sequence In/Out



Catagory Error-control encoder

Location Error-Control Coding Sublibrary

Description The Linear Block Encode Sequence In/Out block performs error-control encoding using linear block code with a sequential input/sequential output format. The block encodes a binary message into a binary codeword. In the sequential format, the block holds the input message bits in a length K buffer. When the buffer is loaded, the encoding process begins. The block outputs the length N codeword in a bit stream.

The relationship between the two input and output sample times is:

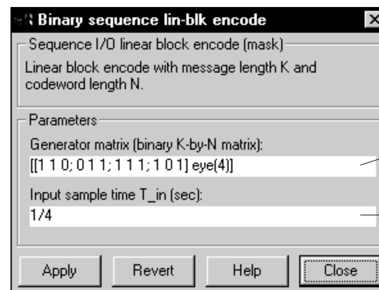
```
output_sample_time=input_sample_time*N/K
```

where N is the codeword length and K is the message length. Because encoding cannot begin until the input buffer is full, there is a time delay introduced by the sequential format. The time delay between the arrival of the first message signal and the output of the first codeword is:

```
K*Input_Sample_Time
```

Output port 1 outputs the codeword in sequential binary form. Output port 2 outputs an impulse. The raising edge of the second output indicates the start of the encode process.

Dialog Box



A K-by-N matrix that generates the codeword signal from the message signal.

The time interval for each input bit in the sequence. The output sample time equals the `input_sample_time*K/N`.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block Linear Block Decode with Sequence In/Out

Linear Block Decode Sequence In/Out

Catagory Error-control decoder

Location Error-Control Coding Sublibrary

Description

The Linear Block Decode Sequence In/Out block recovers a binary message from a codeword using linear block code in a sequential in/out format. The generator matrix used in this block must match the one used in the Linear Block Encode block in order to obtain the correct result. In the sequential format, the block holds the input codeword bits in a length N buffer. When the buffer is loaded, the decoding process begins. The block outputs the length K recovered message in a bit stream.

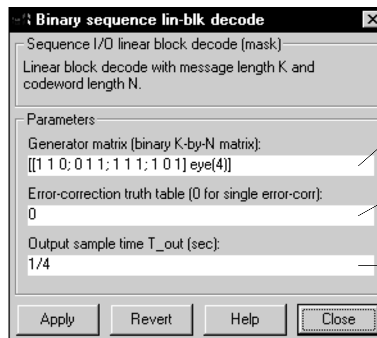
The relationship between the two sample times is:

```
input_sample_time=output_sample_time*N/K
```

where `N` is the codeword length and `K` is the message length. Because decoding cannot begin until the input buffer is full, this block introduces a delay between the arrival of the first codeword signal and the first output. The time delay length is:

```
K*Output_Sample_Time
```

Dialog Box



An K-by-N matrix that generates the codeword signal from message signal.

A 2^{N-K}-by-N matrix that determines error bit location. Set this to 0 for a systematic single error-correction code.

The time interval for each output bit in the sequence.

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs/ Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	N/A
Direct Feedthrough	Yes
Delay	Input_Sample_Time ∗ K

Pair Block Linear Block Decode Sequence In/Out

Convolutional Encode Vector In/Out

Catagory

Error-control encoder

Location

Error-Control Coding Sublibrary

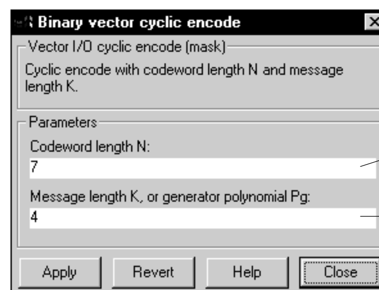
Description

The Convolutional Encode Vector In/Out block encodes a message into a codeword using a vector in/out format. Unlike block codes, convolutional code requires information not only on the incoming message but also on a fixed number of previous messages. Accordingly, convolutional codes are called (N,K,M) codes where N is the codeword length, K the message length, and M the memory length.

Because of the memory length requirement, convolutional code implementation in this toolbox requires a transfer function matrix in the parameter **Transfer function**. An example of how to build a transfer function using Simulink block diagrams is presented in the Generating Tran_func block located in the Convolutional Coding/Decoding Sublibrary. Following the rules outlined in the Generating Tran_func block, you can build your own Simulink block diagram of the transfer function and use the string name of the Simulink model as an input to the dialog box. The block will generate the correct transfer function. You can also choose to input directly a matrix representation of the numerator and denominator of a transfer function.

Input port 1 accepts the message signal, and the input port 2 accepts a trigger signal. The convolutional encoding process begins when the leading edge of the signal crosses the specified threshold value. You can use the Vector Pulse block to generate a trigger signal for this input port.

Dialog Box



Specify a transfer function matrix or the string name of a Simulink block diagram.

Set the trigger signal threshold.

Characteristics

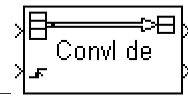
No. of Inputs/Outputs	2/1
Vectorized Input 1/Input 2	Yes/No
Input No. 1 Vector length	K
Vectorized Output	Yes
Output Vector length	N
Time Base	Triggered
States	2M, where M is the number of registers in the convolutional coding structure
Direct Feedthrough	Depends on the transfer function

Pair Block Convolutional Encode Vector In/Out

Equivalent M-files encode

See Also For more information on how to construct transfer functions, see:
sim2gen, sim2tran, sim2logi

Convolutional Decode Vector In/Out



Catagory Error-control decoder

Location Error-Control Coding Sublibrary

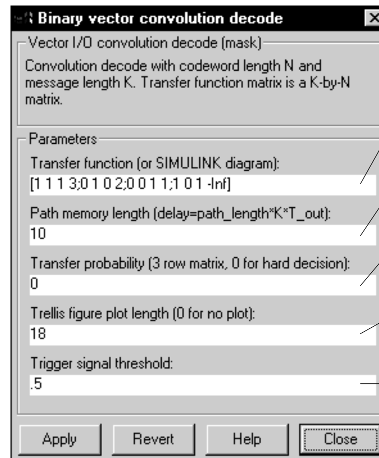
Description

The Convolutional Decode Vector In/Out block recovers a binary codeword vector from a binary message vector using the Viterbi decoding algorithm. Refer to Chapter 3, the Tutorial, for a more detailed discussion of this technique. This block supports both “hard” and “soft” decision decoding. Using the soft decision method requires the definition of transfer probabilities.

Input port 1 inputs a length N binary vector codeword. Input port 2 inputs a scalar computation trigger signal. You can use the Vector Pulse block to generate a trigger signal for this input port.

Output port 1 outputs the recovered message vector. Output port 2 outputs a scalar. This output is a metric value in the case of soft decision decoding, and the accumulative distance in the case of hard decision decoding.

Dialog Box



Match the transfer function to the one used in the Convolutional Encode block.

The delay length. Set this to a number greater than the memory length M of the decoder.

The transfer probability. Set this to 0 for hard decision decoding. Otherwise, enter a K-by-N matrix of probabilities.

Trellis figure plot length. Set this to an integer larger than the memory length M for a decision track. Setting this to 0 suppresses the trellis plot.

Match the trigger signal threshold to that of the Convolutional Encode block.

Characteristics

No. of Inputs/Outputs	2/2
Vectorized No.1/No. 2 Inputs	Yes/No
No. 1 Input Vector length	N
Vectorized No. 1/No. 2 Outputs	Yes/No
No. 1 Output Vector length	K
Time base	Triggered
States	Delay_length $*$ 2^M, where M is the number of registers in the convolutional coding structure
Direct Feedthrough	Yes
Delay	Interval_between_trigger*Memory_length

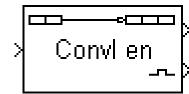
Pair Block

Convolutional Encode Vector In/Out

Equivalent M-function

encode

Convolutional Encode Sequence In/Out



Catagory Error-control encoder

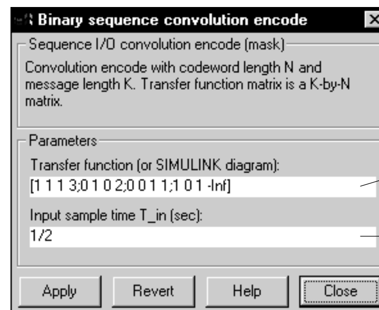
Location Error-Control Coding Sublibrary

Description The Convolutional Encode Vector In/Out block encodes a message into a codeword using a binary sequential in/out format. Unlike block codes, convolutional code requires information not only on the incoming message but also on a fixed number of previous messages. Accordingly, convolutional codes are called (N,K,M) codes where N is the codeword length, K the message length, and M the memory length.

Because of the memory length requirement, convolutional code implementation in this toolbox requires a transfer function matrix in the parameter **Transfer function**. An example of how to build a transfer function using Simulink block diagrams is presented in the Generating Tran_func block located in the Convolutional Coding/Decoding Sublibrary. Following the rules outlined in the Generating Tran_func block, you can build your own block diagram of the transfer function and use the string name of the Simulink model as an input to the dialog box. The block will generate the correct transfer function. You can also choose to directly input a matrix representation of the numerator and denominator of a transfer function.

Input port 1 accepts the message signal, and the input port 2 accepts a trigger signal. The convolutional encoding process begins when the leading edge of the signal crosses the specified threshold value. You can use the Vector Pulse block to generate a trigger signal for this input port.

Dialog Box



Specify a transfer function matrix or the string name of a Simulink block diagram.

Set the input sample time.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-rate
States	2^M, where M is the number of registers in the convolutional coding structure
Direct Feedthrough	Yes
Delay	Input_Sample_Time $*$ K

Pair Block

Convolutional Decode Sequence In/Out

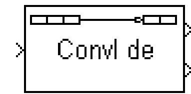
Equivalent M-function

encode

See Also

For more information on how to construct transfer functions, see:
`sim2gen`, `sim2tran`, `sim2logi`

Convolutional Decode Sequence In/Out



Catagory

Error-control encoder

Location

Error-Control Coding Sublibrary

Description

The Convolutional Decode Sequence In/Out block recovers a binary message from a binary codeword using the sequential in/out format. The technique used for decoding is the Viterbi algorithm. Refer to Chapter 3, the *Tutorial*, for a more detailed discussion of this technique.

This block supports both "hard" and "soft" decision decoding. Soft decision decoding requires the definition of a transfer probability matrix.

Dialog Box

Binary sequence convolution decode

Sequence I/O convolution decode (mask)

Convolution decode with codeword length N and message length K. Transfer function matrix is a K-by-N matrix.

Parameters

Transfer function (or SIMULINK diagram):
[1 1 1 3;0 1 0 2;0 0 1 1;1 0 1 -Inf]

Path memory length (delay=path_length*K*T_out):
5

Transfer probability (3 row matrix, 0 for hard decision):
0

Trellis figure plot length (0 for no plot):
10

Output sample time T_out (sec):
1/2

Apply | Revert | Help | Close

Match the transfer function to the one used in the Convolutional Encode block.

The delay length. Set this to a number greater than the memory length M of the decoder.

The transfer probability. Set this to 0 for hard decision decoding. Otherwise, enter a K-by-N matrix of probabilities, where K is the message length and N is the codeword length.

Trellis figure plot length. Set this to an integer larger than the memory length M for a decision track. Setting this to 0 suppresses the trellis plot.

The time interval for each output bit in the sequence. The output_sample_time =input_sample_time*K/N

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete-time, Multi-Rate
States	N/A
Direct Feedthrough	Yes
Delay	In_Sample_Time ∗ K ∗ (Memory_length+1)

Pair Block Convolutional Encode Sequence In/Out

Equivalent M-function decode

Modulation and Demodulation

In communication systems, information is often transmitted by modulating the source signal onto a carrier signal to form a modulated signal. A modulated signal can be transferred a longer distance with considerably less energy compared to an unmodulated signal. The modulated signal can also share the same transmitting media and bandwidth with other modulated signals. Modulation can be achieved by a number of modulation techniques. The Communications Toolbox supports the most commonly used methods for modulation and demodulation.

The Communications Toolbox provides both an analog modulation/demodulation library and a digital modulation/demodulation library. Digital modulation can be divided into two parts: digital signal mapping and modulation. A digital demodulation can also be divided into two parts: demodulation and digital demapping. The division is useful for detailed research of the mapping/demapping techniques and in its combination with modulation/demodulation techniques. This toolbox provides a library for the mapping/demapping sublibraries under the modulation/demodulation category.

This figure shows the Modulation/Demodulation Sublibrary:

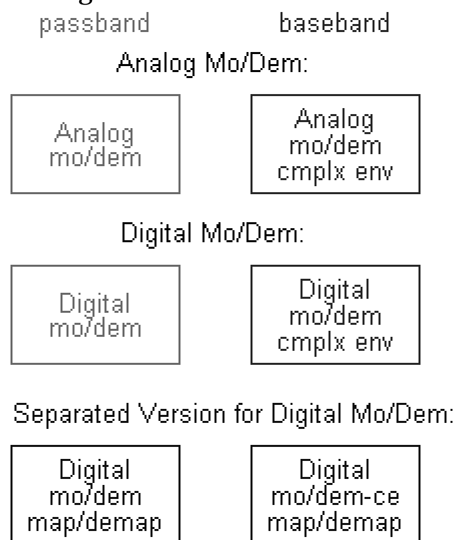


Figure 6-10: Modulation/Demodulation Sublibrary

The first column shows the sublibraries of the passband simulation blocks; the second column shows the sublibraries of the baseband simulation blocks. The first row shows the sublibraries for analog modulation. The second row shows the sublibraries for digital modulation. The third row shows the sublibraries for mapping/demapping in digital modulation/demodulation.

This chapter introduces the blocks by going through each sublibrary. This section is further divided into these five subsections:

- Analog Modulation/Demodulation Method for Passband Simulation
- Analog Modulation/Demodulation Method for Baseband Simulation
- Mapping/Demapping for Digital Modulation/Demodulation
- Digital Modulation/Demodulation Method for Passband Simulation
- Digital Modulation/Demodulation Method for Baseband Simulation

Analog Modulation/Demodulation Methods for Passband Simulation

The Communications Toolbox includes these analog modulation methods for passband simulation.

- Double sideband suppressed-carrier amplitude modulation (DSB-SC AM)
- DSB-SC with carrier amplitude modulation (AM with Carrier)
- Quadrature multiplexing DSB-SC amplitude modulation (QM DSB AM)
- Single sideband amplitude modulation (SSB AM)
- Frequency modulation (FM)
- Phase modulation (PM)

The toolbox also includes demodulation methods for these modulation implementations.

A lowpass filter plays an important role in the demodulation of a modulated signal. It is important that the lowpass filter be well designed. In this toolbox, lowpass filters are not provided as default parameters. You must provide the numerator and denominator of a filter in a demodulation block. However, if you do not know which filter to choose, you can use functions in the Signal Processing Toolbox to find a proper filter for your application. A simple

example of designing a demodulation lowpass filter is to design a Butterworth filter using the command:

```
[num, den] = butter(Ord, Fc * 2 * Sample_time);
```

where `Ord` is the order of the filter; `Fc` is the carrier frequency in the modulation; `Sample_time` is the calculation sampling time; and `num` and `den` are the numerator and denominator of the lowpass filter. All of the examples in this toolbox use this command in computing the lowpass filter.

Demodulation procedures introduce some time delays because of the presence of lowpass filters in the demodulation process. The time delay varies with the filter structure, filter parameters, and sampling time. The toolbox does not provide the estimation of the time delay of the demodulation.

This figure shows the Analog Passband Modulation/Demodulation Sublibrary:

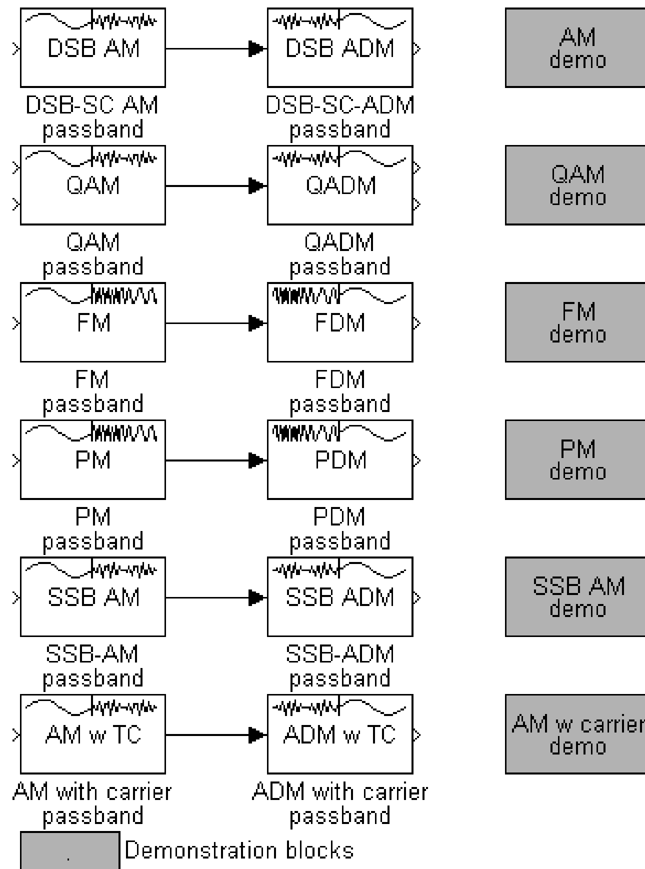


Figure 6-11: Analog Passband Modulation/Demodulation Sublibrary

Analog Modulation/Demodulation Passband Reference Table

This table lists the Simulink blocks in the Analog Mo/Dem Passband sublibrary. (They are listed alphabetically in this table for your convenience.):

Block Name	Description
ADM with Carrier	Amplitude demodulation with carrier
AM with Carrier	Amplitude modulation with carrier
DSB ADM	Double sideband amplitude demodulation
DSB-SC AM	Double sideband amplitude modulation with suppressed carrier
FDM	Frequency demodulation
FM	Frequency modulation
PDM	Phase demodulation
PM	Phase modulation
QADM	Quadrature amplitude demodulation
QAM	Quadrature amplitude modulation
SSB-ADM	Single sideband amplitude demodulation
SSB-AM	Single sideband amplitude modulation

Catagory

Passband modulation

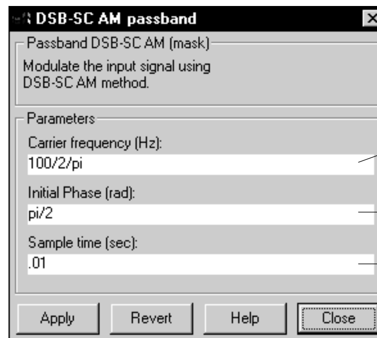
Location

Analog Mo/Dem, Passband Sublibrary

Description

The DSB-SC AM (double sideband suppressed-carrier amplitude modulation) block modulates an input signal with a carrier frequency signal. For any input signal, *u(t)*, and carrier frequency, f_c, the resulting modulated signal, y(t) is $y(t) = A_c u(t)\sin(2\pi f_c t + \varphi)$ where A_c is an amplitude scale factor and φ is the initial phase of the modulation.

Dialog Box



The carrier frequency (Hz). Set this to a number greater than the message signal frequency.

The initial phase (rad).

The sample time (sec).

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

Costas PLL DSB ADM

Equivalent M-function

commod

DSB ADM

DSB ADM

Catagory Passband demodulation

Location Analog Mo/Dem, Passband Sublibrary

Description The DSB ADM (phase-locked loop double sideband suppressed-carrier amplitude demodulation) block recovers a message signal from its modulated form. The DSB ADM method is shown below:

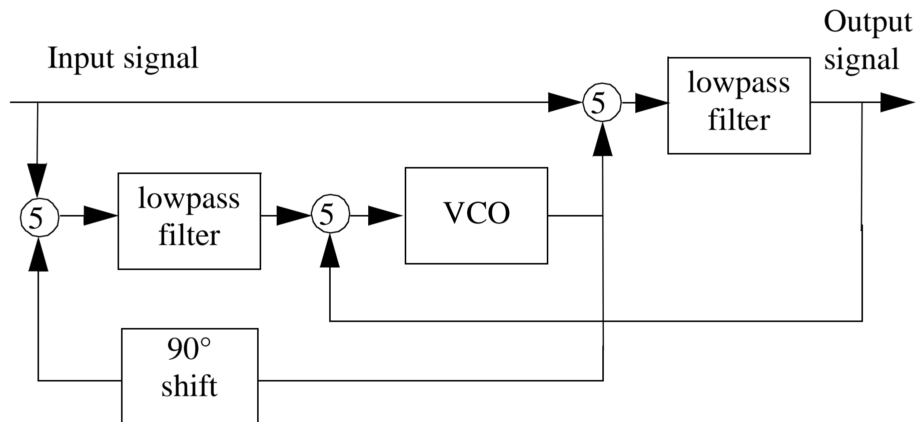
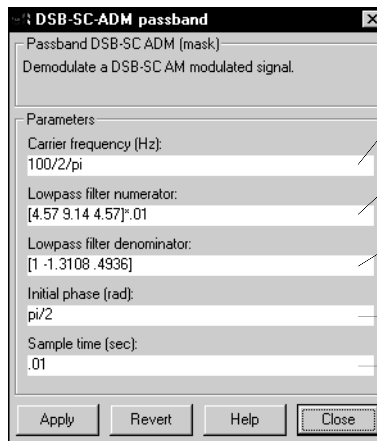


Figure 6-12: DSB Amplitude Demodulation

(VCO -- voltage controlled oscillator).

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the carrier frequency (Hz) to the one used in the corresponding DSB-SC AM block.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase (rad) of the carrier signal.

Match the sample time to the one used in the corresponding DSB-SC AM block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	2 * order_of_the_lowpass_filter
Direct Feedthrough	Yes

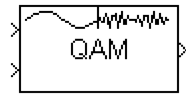
Pair Block

DSB-SC AM

Equivalent M-function

comdemod

QAM



Catagory Passband quadrature modulation

Location Analog Mo/Dem, Passband Sublibrary

Description The QAM (quadrature amplitude modulation) block combines two independent message signals into a single modulated signal. The two independent message signals are combined with two carrier frequencies that are 90° out of phase with each other. The resulting signal components are orthogonal, which simplifies message signal recovery on the receiving side. Quadrature modulation used in QAM is shown below:

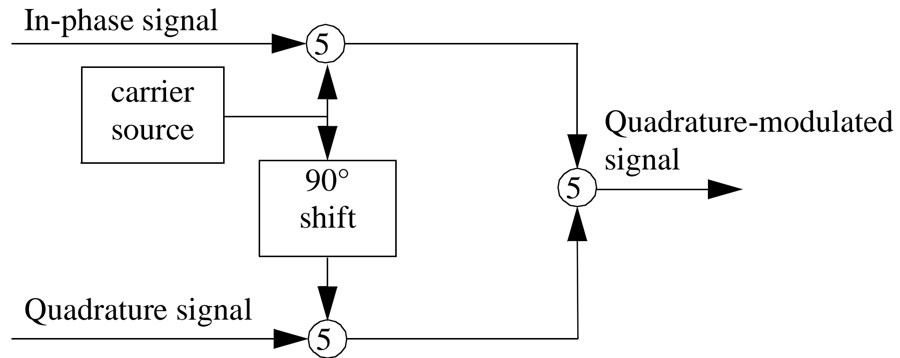
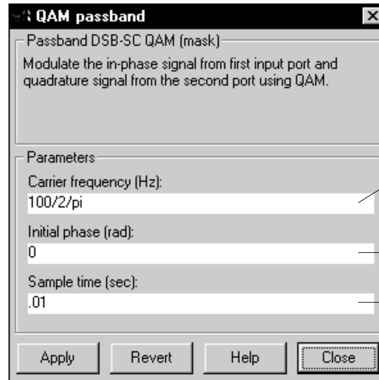


Figure 6-13: Quadrature Amplitude Demodulation

Input port 1 accepts the in-phase signal, and input port 2 accepts the quadrature signal.

Dialog Box



Set the carrier frequency to a number greater than either of the frequencies of the two input message signals.

The initial phase of the carrier signal.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

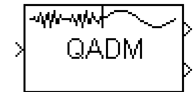
Pair Block

QADM

Equivalent M-function

commod

QADM



Catagory Passband quadrature demodulation

Location Analog Mo/Dem, Passband Sublibrary

Description The QMADM (quadrature amplitude demodulation) block recovers a message signal from a quadrature modulated carrier signal. This block recovers both the in-phase and quadrature message signals. The figure below shows the demodulation process:

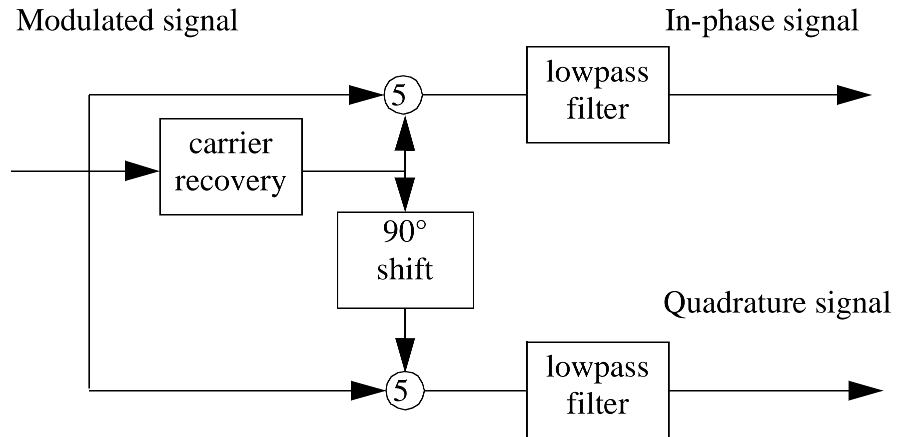
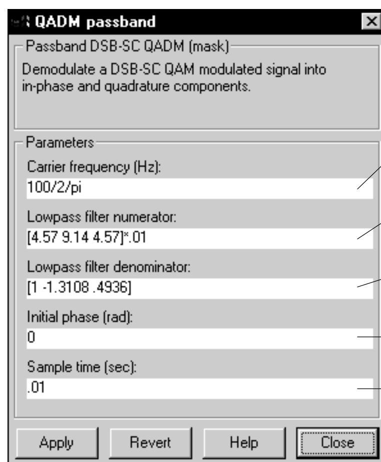


Figure 6-14: Quadrature Amplitude Demodulation

Output port 1 outputs the recovered in-phase signal. Output port 2 outputs the recovered quadrature signal.

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the carrier frequency (Hz) to that of the corresponding QAM block.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase (rad) of the carrier signal.

Match the sample time to the one used in the corresponding QAM block.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	2 * Order_of _the_lowpass_filter
Direct Feedthrough	Yes

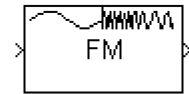
Pair Block

QAM

Equivalent M-function

comdemod

FM



Catagory Passband modulation

Location Analog Mo/Dem, Passband Sublibrary

Description

The FM (frequency modulation) block modulates a message signal using frequency modulation. In this technique, the signal frequency varies with the amplitude of the input message signal. Let $u(t)$ be the input message signal, A_c the amplitude scale factor; K_c the modulation sensitivity constant, and ϕ be the initial phase. The the output of this block is:

$$y(t) = A_c \cos(2\pi\theta t) + \varphi)$$

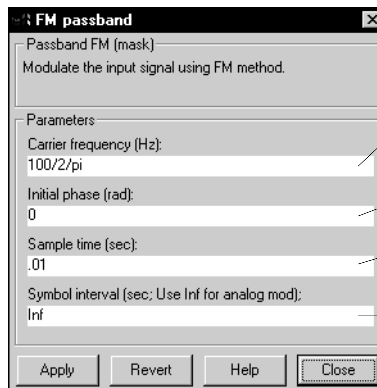
where

$$\theta(t) = f_c(t) + K_c \int_0^t u(t)\,dt$$

and f_c is the carrier frequency.

It is possible to model FSK (frequency shift keying) in this block by setting the symbol interval parameter. This parameter, when set to a value other than Inf, resets the integrator in the above equation. To model FSK, set the value to the length required to transmit an information bit. This forces an integrator reset at the onset of each bit transmission.

Dialog Box



Set the carrier frequency to a number greater than either of the frequencies of the two input message signals.

The initial phase of the carrier signal.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

For analog modulation, set this parameter to Inf. For FSK modeling, set it to the length of time required to transmit a single information bit.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	1
Direct Feedthrough	No

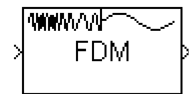
Pair Block

FDM

Equivalent M-function

commod

FDM



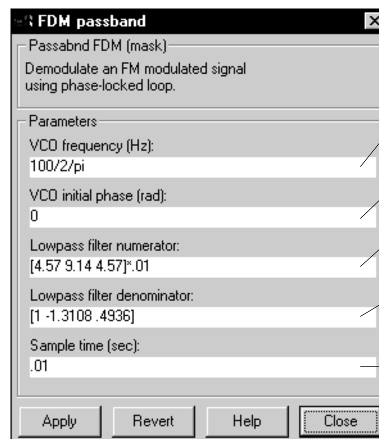
Catagory Passband demodulation

Location Analog Mo/Dem, Passband Sublibrary

Description The FDM (frequency demodulation) block recovers a message signal from a frequency modulated carrier signal.

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the carrier frequency to that of the corresponding FM block.

VCO initial phase (rad).

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

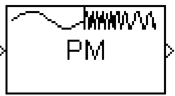
Match the sample time to that of the corresponding FM block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter + 1
Direct Feedthrough	Yes

Pair Block FM

Equivalent M-function comdemod



Catagory Passband modulation

Location Analog Mo/Dem, Passband Sublibrary

Description The PM (phase modulation) block converts an input message signal into a modulated signal using phase modulation. In this technique, the phase of the carrier frequency is varied in proportion to the amplitude of the incoming message signal. Let $u(t)$ be the input message signal, A_c be the amplitude scale factor, and ϕ be the initial phase. The output of this block is:

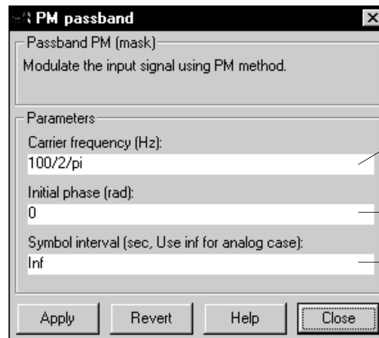
$$y(t) = A_c \cos(2\pi\theta(t) + \varphi)$$

where

$$\theta(t) = f_c(t) + u(t)$$

and f_c is the carrier frequency. If the input signal is not continuous, the modulated output signal may be discontinuous.

Dialog Box



The carrier frequency must be greater than the frequency bandwidth of either of the input message signals.

The initial phase of the carrier signal.

For analog modulation, set this parameter to Inf.

Characteristics

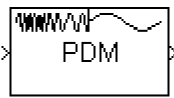
No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block

PLL PDM

Equivalent M-function

commod



PDM

Catagory

Passband demodulation

Location

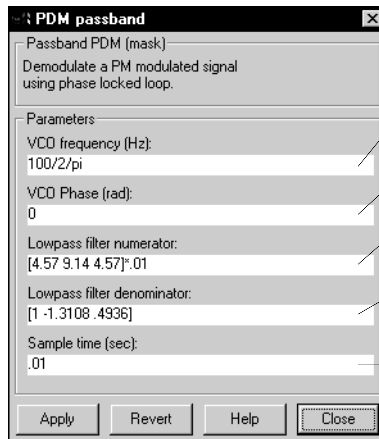
Analog Mo/Dem, Passband Sublibrary

Description

The PDM (phase demodulation) block recovers a message signal from a phase-modulated carrier signal.

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the carrier frequency to that of the corresponding PM block.

The initial phase of the carrier frequency.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter + 2
Direct Feedthrough	Yes

Pair Block

PM

Equivalent M-function

comdemod

SSB-AM

Catagory Passband modulation

Location Analog Mo/Dem, Passband Sublibrary

Description

The SSB-AM (single sideband amplitude modulation) block modulates an input signal with a given carrier frequency using SSB-AM with a Hilbert filter. For details on Hilbert transforms, refer to Chapter 3, the *Tutorial.*

SSB-AM transmits either the lower or upper sideband signal, but not both as is the case is DSB-AM.

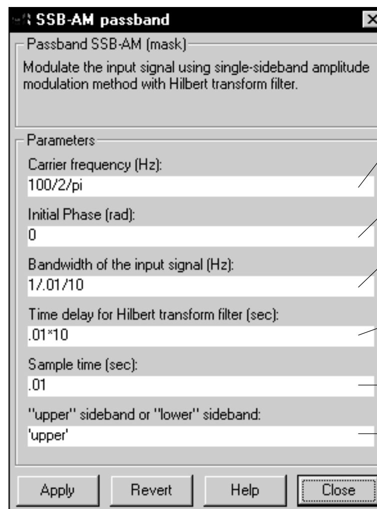
Set the bandwidth of the input signal parameter to the estimated highest frequency component in the input message signal. This is used to design a compensator with the Hilbert filter. This forces the message signal amplitude to remain within the assigned range. If assigned to 0 or a value larger than 1/(2*sample_time), the block does not generate a compensator.

This block requires a time delay for the Hilbert filter. Set this parameter to

```
(N+0.5)*sample_time
```

where N is a positive integer. This block uses the function `hilbiir` to design the Hilbert filter.

Dialog Box



The carrier signal (Hz) must be greater than the frequency bandwidth of the input signal.

The initial phase (rad) of the carrier frequency.

Set the bandwidth of the input signal to 0 for no compensation. Otherwise, set it to the highest frequency component of the message signal.

Set the time delay to (N + 0.5) * Sample_time, where N is a positive integer.

Specify the sample time.

Specify whether to use the upper or lower sideband.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	ceil(Time_delay/Sample_time) + 5
Direct Feedthrough	Yes

Pair Block SSB ADM

Equivalent M-function commod

SSB ADM

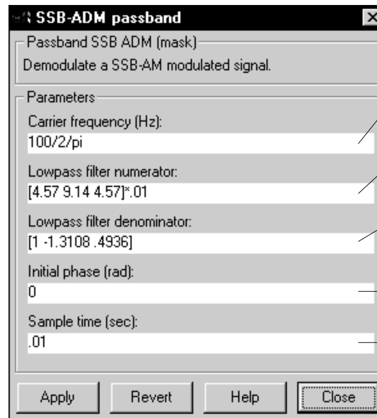
Catagory Passband demodulation

Location Analog Mo/Dem, Passband Sublibrary

Description The SSB ADM (single sideband amplitude demodulation) recovers a message signal from a signal modulated using the SSB AM technique.

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the carrier frequency to that of the corresponding SSB AM block.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase of the carrier frequency.

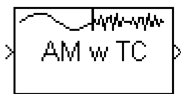
Match the sample time to that of the corresponding SSB AM block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter
Direct Feedthrough	Yes

Pair Block SSB AM

Equivalent M-function comdemod



AM With Carrier

Catagory Passband modulation

Location Analog Mo/Dem, Passband Sublibrary

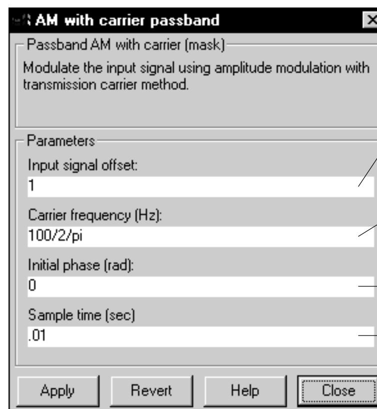
Description

The AM With Carrier block modulates the input signal with a given carrier frequency using the amplitude modulation with carrier technique. In this method, the modulated output signal is given by

$$y(t) = (u(t) + k)\sin(2\pi f_c t)$$

where *y(t)* is the output signal, *u(t)* the input signal to be modulated, f_c is the carrier frequency, and k an offset value that is added to the signal in this modulation technique. It is common to set the value of k to the maximum absolute value of the negative part of the input signal *u(t)*. The output signal is sinusoidal with its envelope having the shape of the input signal.

Dialog Box



The offset factor in the above AM equation. This value should be larger than or equal to the absolute value of the minimum value of the message signal *u(t)*.

The carrier frequency must be greater than the frequency bandwidth of either of the input message signals.

The initial phase of the carrier signal.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block ADM with Carrier

Equivalent M-function commod

ADM with Carrier

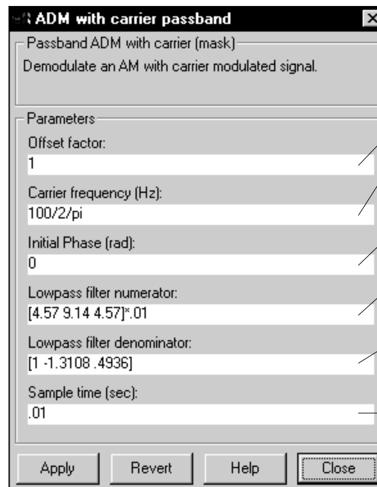
Catagory Passband demodulation

Location Analog Mo/Dem, Passband Sublibrary

Description The ADM with Carrier block recovers a message signal from an AM with carrier modulated signal using the envelope detection method. The parameters set in this block must match those used in the AM With Carrier block.

Lowpass filters play an important role in the demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Block



Match the offset factor and the carrier frequency to those in the corresponding AM with Carrier block.

The initial phase of the carrier frequency.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	2 * Order_of_the_lowpass_filter
Direct Feedthrough	Yes

Pair Block AM with Carrier

Equivalent M-function comdemod

Analog Modulation/Demodulation Methods for Baseband Simulation

Prior to using the blocks in this section, you should read through the "Baseband Simulation" section in chapter 3, the *Tutorial*.

A modulation block in baseband simulation takes a real signal and outputs a complex signal. A demodulation block in baseband simulation takes a complex signal and outputs a real signal. In the Simulink block library of this toolbox, a complex number is represented by a length two vector with the first element being the real part and the second element being the image part of a complex number.

The baseband block library is analogous to the passband simulation. For every modulation/demodulation block in baseband, you can find a passband counterpart in the passband and vice versa. The simulation results of these two techniques should be very close to each other. The advantage of using the baseband simulation is that the speed for baseband simulation can be much faster than the passband simulation since the sample frequency for passband simulation must be at least twice as the carrier frequency. There is no such limit for the baseband simulation.

To match the result between the baseband and passband simulation, this toolbox uses lowpass filters in the baseband simulation. Note that the sampling frequency for the passband simulation may be different from the sampling frequency for the baseband simulation. To obtain a similar feature of a filter under a different sampling time, use the functions `c2dm` and `d2cm` in the Control System Toolbox to convert the filters from one sample frequency to another, for example, if your filter has the numerator, `num1`, and denominator, `den1`, at the sampling time `ts1`. To get a filter with the same frequency response at sampling time `ts2`, you can use the following commands to do the conversion.

```
[num, den] = d2cm(num1, den1, ts1);
[num2, den2] = c2dm(num, den, ts2);
```

The resulting new filter has numerator `num2`, denominator `den2`, and sampling time `ts2`. The numerator `num2` and denominator `den2` with sampling time `ts2` have the similar frequency response as the pair `num1` and `den1` with sampling time `ts1`.

The carrier frequency is no longer needed as a parameter in the baseband modulation/demodulation simulation. If you don't care about the delay factors, you can directly assign both numerator and denominator of the filter to be 1.

This figure shows the Analog Modulation/Demodulation, Baseband Sublibrary:

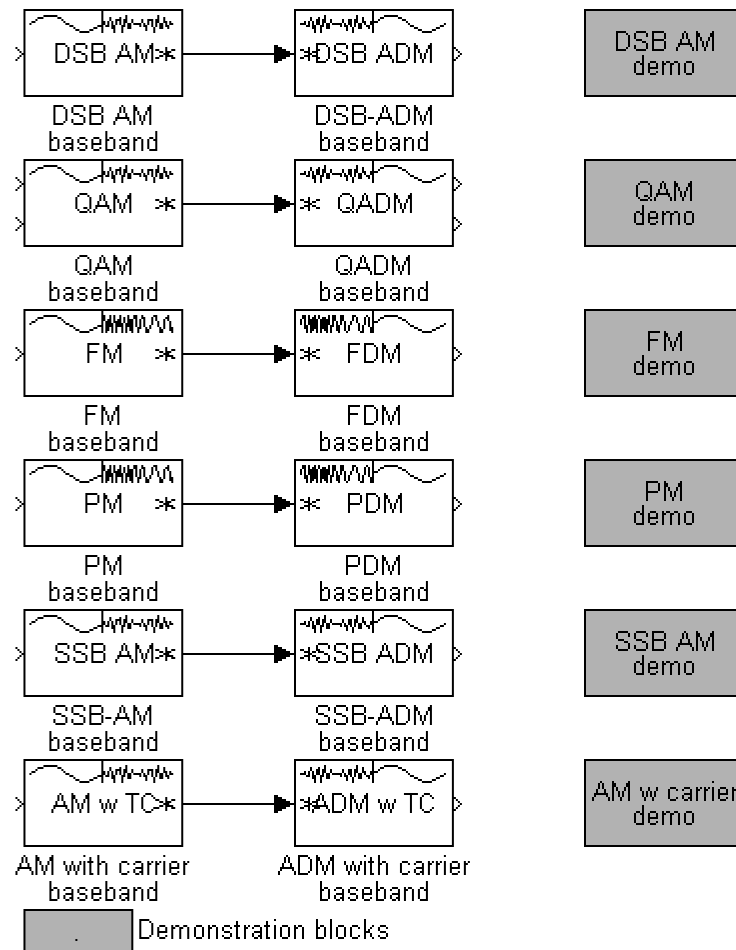


Figure 6-15: Analog Modulation/Demodulation, Baseband Sublibrary

Analog Modulation/Demodulation Baseband Reference Table

This table lists the Simulink blocks in the Analog Modulation/Demodulation Baseband Sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
ADM with Carrier CE	Amplitude demodulation with carrier
AM with Carrier CC	Amplitude modulation with carrier
DSB ADM CE	Double sideband amplitude demodulation
DSB-SC AM CE	Double sideband amplitude modulation with suppressed carrier
FDM CE	Frequency demodulation
FM CE	Frequency modulation
PDM CE	Phase demodulation
PM CE	Phase modulation
QADM CE	Quadrature amplitude demodulation
QAM CE	Quadrature amplitude modulation
SSB-ADM CE	Single sideband amplitude demodulation
SSB-AM CE	Single sideband amplitude modulation

Catagory Baseband modulation

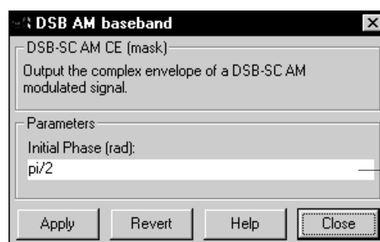
Location Analog Mo/Dem, Baseband Sublibrary

Description The DSB-SC AM CE (double sideband-suppressed carrier amplitude modulation complex envelope) block modulates an input signal with a carrier frequency. The input signal, *u(t)*, is a real signal. The output signal, *y(t)*, is a complex signal represented by a length 2 vector. The first index is the real part of the modulated signal, the second index is the complex part. The output of this block is:

$$y(t) = u(t)e^{j\theta}$$

where θ is the phase shift of the signal.

Dialog Box



The initial phase of the carrier frequency.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block DSB ADM CE

Equivalent M-function `amodce`

DSB ADM CE

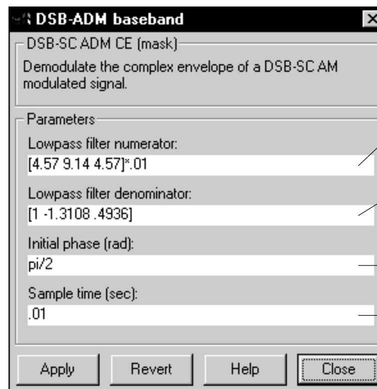
Catagory Baseband demodulation

Location Analog Mo/Dem, Baseband Sublibrary

Description The DSB ADM CE (amplitude demodulation complex envelope) block recovers a message signal from the input modulated signal. The input to this block is a complex signal, represented in Simulink as a length 2 vector. The first element is the real part of the signal, and the second is the imaginary part. The output of this block is the recovered signal, which is a real scalar.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase of the carrier frequency.

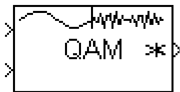
Match the sample time to that of the corresponding DSB-SC AM block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	2 * order_of_the_lowpass_filter
Direct Feedthrough	Yes

Pair Block DSB-SC AM

Equivalent M-function `ademodce`



Catagory Baseband modulation

Location Analog Mo/Dem, Baseband Sublibrary

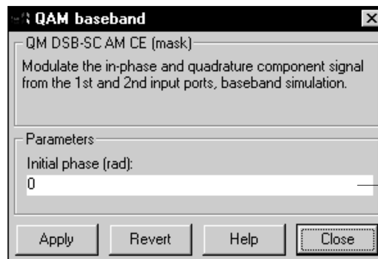
Description The QAM CE (quadrature amplitude modulation complex envelope) block modulates two independent input signals with a single carrier frequency signal. The two input signals are real, and the output modulated signal is complex. Simulink represents complex numbers as length 2 vectors. The first element is the real part of the signal, and the second element is the imaginary part. The mathematical operation in this block is:

$$y(t) = (u_1(t) + u_2(t))e^{j\theta}$$

where $y(t)$ is the complex output signal, $u_1(t)$ and $u_2(t)$ are the real input signals, and θ is the phase shift of the carrier frequency.

Input port 1 accepts an in-phase signal $u_1(t)$ and input port 2 accepts a quadrature signal $u_2(t)$. The output port outputs $y(t)$, the complex envelope of the modulated signal.

Dialog Box



The initial phase in the carrier signal.

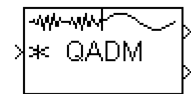
Characteristics

No. of Inputs/Outputs	2/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block QADM CE

Equivalent M-function amodce

QADM CE



Catagory Baseband demodulation

Location Analog Mo/Dem, Baseband Sublibrary

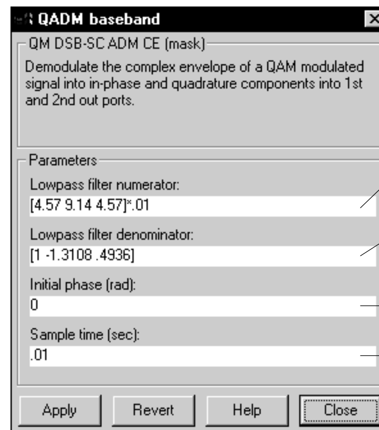
Description

The QADM CE (quadrature amplitude demodulation complex envelope) block recovers a message signal from a modulated input signal. This block accepts a complex signal and outputs a two-part recovered message signal. The first signal is in-phase part of the message signal, and the second is the quadrature part.

The input port accepts the complex modulated signal. Output 1 outputs the in-phase part of the message signal, and output port 2 outputs the quadrature part.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase of the carrier frequency.

By the Nyquist sampling theorem, 1/sample_time > 2 * carrier_frequency.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	2 * order_of_the_lowpass_filter
Direct Feedthrough	Yes

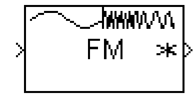
Pair Block

QM DSB AM CE

Equivalent M-function

`ademodce`

FM CE



Catagory Baseband modulation

Location Analog Mo/Dem, Baseband Sublibrary

Description The FM CE (frequency modulation complex envelope) block modulates a message signal using the frequency modulation technique. In FM, the amplitude of the input signal directly affects the frequency of the modulated signal.

The input to this block is a real signal. The output signal is a complex number, which is the complex envelope of the frequency modulation of the input signal. The mathematical relationship between the input and output signals is:

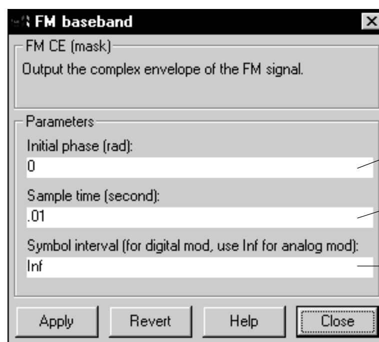
$$y(t) = e^{j\left(\theta + \int_0^t u(\tau)\,d\tau\right)}$$

where *u(t)* is the input signal, *y(t)* the output signal, and θ the phase-shift in the modulation.

The complex output is represented by a length 2 vector. The first element is the real part of the signal, and the second element is the imaginary part.

It is possible to model FSK (frequency shift keying) in this block by setting the **symbol interval** parameter. This parameter, when set to a value other than Inf, resets the integrator in the above equation. To model FSK, set the value to the length required to transmit an information bit. This forces an integrator reset at the onset of each bit transmission.

Dialog Box



The initial phase of the carrier signal.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

For analog modulation, set this parameter to Inf. For FSK modeling, set it to the length of time required to transmit a single information bit.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	1
Direct Feedthrough	No

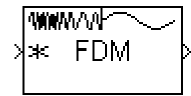
Pair Block

FDM CE

Equivalent M-function

`amodce`

FDM CE



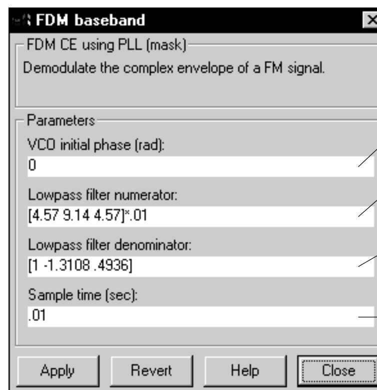
Catagory Baseband demodulation

Location Analog Mo/Dem, Baseband Sublibrary

Description The PLL FDM CE (phase-locked loop frequency demodulation complex envelope) block recovers a message signal from a modulated input signal. The input signal to this block is complex, and this block represents complex signals as length 2 vectors. The first element is the real part of the signal, and the second is the imaginary part.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



The initial phase of the carrier signal.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

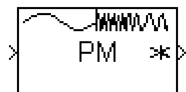
By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter + 1
Direct Feedthrough	No

Pair Block FM CE

Equivalent M-function `ademodce`



Catagory Baseband modulation

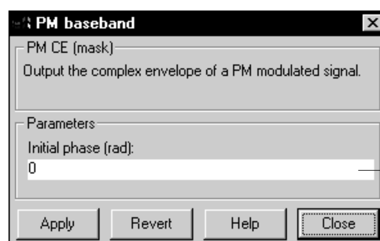
Location Analog Mo/Dem, Baseband Sublibrary

Description The PM CE (phase modulation complex envelope) block modulates an input message signal using phase modulation technique. The input to this block is a real signal. The output is a complex number represented by a length 2 vector. The first element is the real part of the signal, and the second is the imaginary part. This complex number is the complex envelope of the phase modulation. The mathematical relationship between the input, *u(t)*, and the output, *y(t)*, is:

$$y(t) = e^{j(\theta + u(t))}$$

where θ is the phase shift in the modulation.

Dialog Box



The initial phase of the carrier frequency.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block PDM CE

Equivalent M-function `amodce`

PDM CE

Catagory

Baseband demodulation

Location

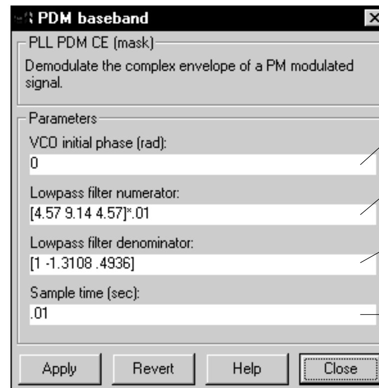
Analog Mo/Dem, Baseband Sublibrary

Description

The PM CE (phase modulation complex envelope) block recovers a message signal from a modulated input signal. The input to this block is a complex signal. The output is the recovered real signal.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



The initial phase of the carrier signal.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter + 2
Direct Feedthrough	No

Pair Block

PM CE

Equivalent M-function

`ademodce`

Catagory

Baseband modulation

Location

Analog Mo/Dem, Baseband Sublibrary

Description

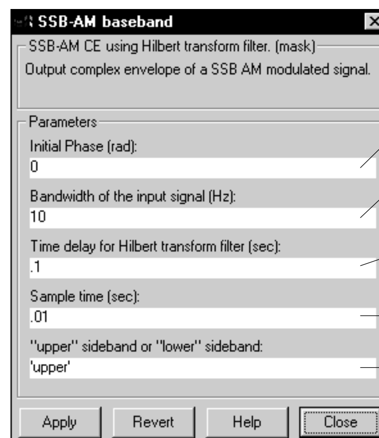
The SSB-AM CE (single sideband-amplitude modulation complex envelope) block modulates an input message signal using amplitude modulation with a Hilbert filter. The single sideband method requires the transmission of either the lower or upper sideband signal, but not both. This limitation means that it is not possible to transmit a DC component of the signal. For details on this method, and on Hilbert transforms, refer to chapter 3, the *Tutorial*.

The input to this block is a real signal. The output is a complex signal. Simulink represents a complex number by a length 2 vector. The first element is the real part of the signal, and the second element is the imaginary part. The mathematical relationship between the input, $u(t)$, and the output, $y(t)$, is:

$$y(t) = (u(t) + j\hat{u}(t))e^{j\theta}$$

where θ is the phase shift of the signal and $\hat{u}(t)$ is the Hilbert transform of the input $u(t)$.

Dialog Box



The initial phase (rad) of the carrier frequency.

Set the bandwidth of the input signal to 0 for no compensation. Otherwise, set it to the highest frequency component of the message signal.

Set the time delay to `(N + 0.5)*Sample_time`, where N is a positive integer.

The sample time of the calculation.

Specify whether to use the upper or lower sideband.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	ceil(time_delay/sample_time) +5
Direct Feedthrough	Yes

Pair Block

SSB ADM

Equivalent M-function

amodce

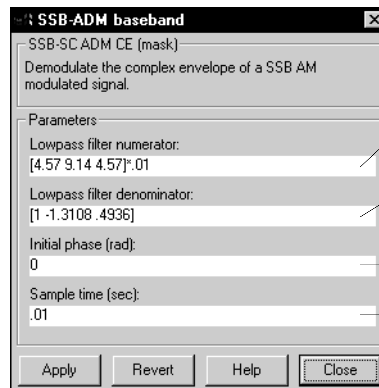
Catagory Baseband demodulation

Location Analog Mo/Dem, Baseband Sublibrary

Description The SSB ADM (single sideband amplitude demodulation) block recovers a message signal from a modulated input signal. The input to this block is a complex signal represented by a length 2 vector. The first element is the real part of the signal, and the second is the imaginary part. The output of this block is a real signal.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase of the carrier signal.

By the Nyquist sampling theorem, 1/sample_time > 2 * carrier_frequency.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	Order_of_the_lowpass_filter
Direct Feedthrough	Yes

Pair Block SSB AM CE

Equivalent M-function `ademodce`

AM With Carrier CE

Catagory Baseband modulation

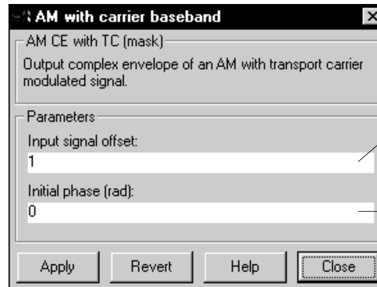
Location Analog Mo/Dem, Baseband Sublibrary

Description The AM with Carrier CE (amplitude modulation with carrier complex envelope) block modulates an input message signal using amplitude modulation. The input to this block is a real signal, and the output is a complex signal. Simulink represents a complex number by a length 2 vector. The first element is the real part, and the second is the imaginary part. The mathematical relationship between the input, *u(t)*, and the output, *y(t,)* is:

$$y(t) = (u(t) + k)e^{j\theta}$$

where q is the phase shift of the signal and k is the initial signal offset.

Dialog Box



The offset factor in the above AM equation. This value should be larger than or equal to the absolute value of the minimum value of the message signal *u(t)*.

The initial phase of the carrier signal.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Pair Block ADM CE with Carrier

Equivalent M-function amodce

ADM with Carrier CE

Catagory Baseband demodulation

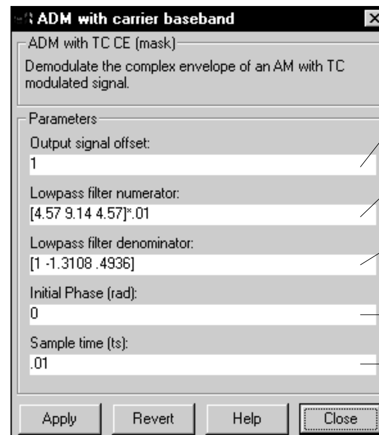
Location Analog Mo/Dem, Baseband Sublibrary

Description

The ADM CE with Carrier (amplitude demodulation complex envelope with carrier) block recovers a message signal from a modulated input signal. The input to this block is a complex signal. Simulink represents complex numbers by length 2 vectors. The first element of the vector is the real part of the complex signal, and the second element is the imaginary part. The output is a real signal.

Lowpass filters play an important role in demodulation. Refer to chapter 3, the *Tutorial*, for information on how to select the numerator and denominator of the lowpass filter transfer function.

Dialog Box



Match the offset fact to the value used in the corresponding AM with Carrier CE block.

The numerator of the lowpass filter transfer function used in the demodulation.

The denominator of the lowpass filter transfer function. For an FIR filter, set this parameter to 1.

The initial phase of the carrier signal.

By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	2 * order_of_the_lowpass_filter
Direct Feedthrough	Yes

Pair Block

AM With Carrier CE

Equivalent M-Function

`ademodce`

Digital Modulation/Demodulation Methods for Passband Simulation

A digital modulation block is the combination of a mapping block and an analog modulation block. A digital demodulation block is the combination of an analog demodulation block and a demapping block. The digital mapping/demapping and modulation/demodulation blocks are discussed separately in the "Passband Digital Modulation/Demodulation" sublibrary section. The blocks in this "Digital Modulation/Demodulation" sublibrary combine the mapping and modulation functions to make ready-to-use digital modulation and demodulation blocks.

This sublibrary uses a default Butterworth lowpass filter in MASK and QASK demodulation blocks. The default filter is designed by using the MATLAB command:

```
[num,den] = butter(Ord,Fc*2*Sample_time)
```

where `Ord` is the order of the filter, `Fc` is the carrier frequency in the modulation, `Sample_time` is the calculation sampling time, and `num` and `den` are the numerator and denominator of the lowpass filter. In this sublibrary, the default lowpass filter has order 5. If you prefer to use your own filter, use separate demapping and demodulation blocks from the "Passband Digital Modulation/Demodulation" sublibrary, where you can customize your own lowpass filters.

The figure below shows the Digital Mod/Demod Mapping/Demapping Passband Sublibrary:

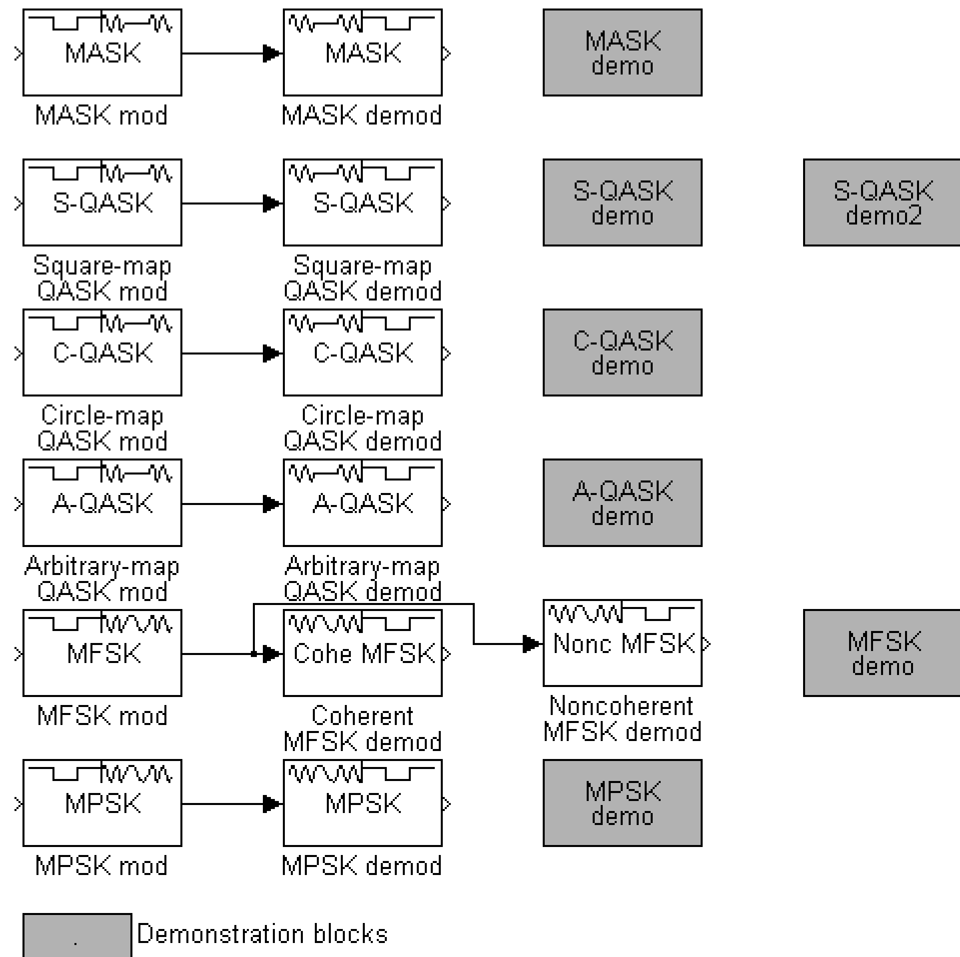


Figure 6-16: Digital Modulation/Demodulation Passband Sublibrary

Digital Modulation/Demodulation Passband Reference Table

This table lists the Simulink blocks in the Digital Modulation/Demodulation Passband Sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
Coherent MFSK Demod	Coherent multiple frequency shift-keying (MFSK) demodulation followed by min/max demapping
MASK Demod	Double sideband suppressed carrier amplitude demodulation (DSB-SC ADM) followed by multiple amplitude shift-keying (MASK) demapping
MASK Mod	M-ary amplitude shift-keying (MFSK) mapping followed by double sideband suppressed carrier amplitude (DSB-SC AM) modulation
MFSK Mod	Multiple frequency shift-keying (MFSK) mapping followed by frequency modulation (FM)
MPSK Mod	Multiple phase shift-keying (MPSK) mapping followed by phase modulation
MPSK Demod	Coherent multiple phase shift-keying (MPSK) correlation demodulation followed by min/max demapping
Noncoherent MFSK Demod	Noncoherent multiple frequency shift-keying (MFSK) demodulation followed by min/max demapping

Block Name	Description
QASK Demod Arbitrary Constellation	Quadrature amplitude demodulation (QADM) followed by quadrature amplitude shift-keying (QASK) demapping using an arbitrary constellation
QASK Demod Circle Constellation	Quadrature amplitude demodulation (QADM) followed by quadrature amplitude shift-keying (QASK) demapping using a circle constellation
QASK Demod Square Constellation	Quadrature amplitude demodulation (QADM) with quadrature shift-keying (QASK) demapping using a square constellation
QASK Mod Arbitrary Constellation	Quadrature amplitude shift-keying mapping using an arbitrary constellation followed by quadrature amplitude modulation (QAM)
QASK Mod Circle Constellation	Quadrature amplitude shift-keying mapping using a circle constellation followed by quadrature amplitude modulation (QAM)
QASK Mod Square Constellation	Quadrature amplitude shift-keying mapping using a square constellation followed by quadrature amplitude modulation (QAM)

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

Description

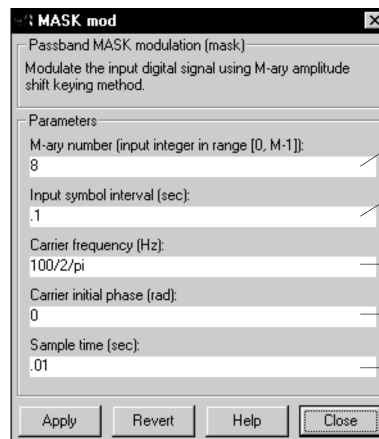
The MASK (multiple amplitude shift-keying) Mod block maps an input signal and then modulates the mapped signal. This block is a combination of the MASK Map and DSB-SC AM blocks. Refer to the individual reference pages for these blocks for discussions of the techniques involved.

MASK Mod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. Typically, M equals 2^K, where K is a positive integer. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Specify the input digit range, [0, M-1].

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value.

The frequency of the carrier signal.

The initial phase of the carrier signal.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

MASK Mod

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MASK Demod

Equivalent M-function

dmod

Catagory

Passband Demodulation/Demapping

Location

Digital Mo/Dem, Passband Sublibrary

Description

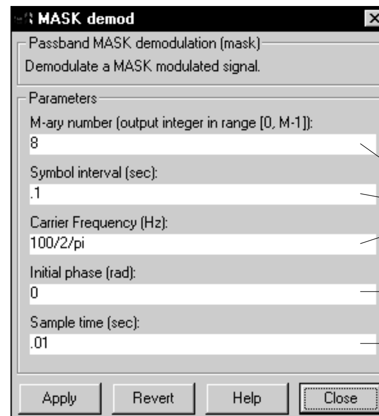
The MASK Demod block recovers a message signal from a modulated and mapped input signal. This block is a combination of the Costas PLL DSB ADM and the MASK Demap blocks. Refer to the reference pages for these blocks for discussions of the techniques involved.

This block uses a default 5th order Butterworth filter, which it generates by using the command:

```
[num, den] = butter(5, Fc*2*Ts);
```

The input to this block is an analog modulated signal with a maximum amplitude equal to one. It outputs integers in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MASK Mod block.

The initial phase of the carrier signal.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics	No. of Inputs/Outputs	1/1
	Vectorized Inputs/Outputs	No/Yes
	Time Base	Discrete time
	States	N/A
	Direct Feedthrough	Yes

Pair Block MASK Mod

Equivalent M-function `ddemod`

QASK Mod Square Constellation

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

Description

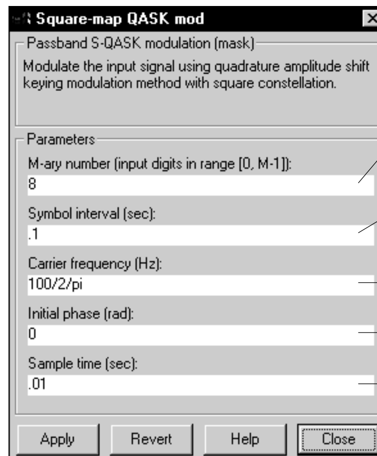
The QASK (quadrature amplitude shift-keying) Mod Square Constellation block maps an input signal into a square constellation signal space and then modulates the mapped signal. This block is a combination of the QASK Map Square Constellation and QM DSB AM blocks. Refer to their individual reference pages for descriptions of the techniques involved.

QASK Mod Square Constellation block is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the square constellation and typically is equal to 2^K, where K is a positive integer. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Specify the input range. The input digit is in the integer range [0,M-1]. Generally $M=2^K$, where K is a positive integer.

The sample time of the input digit. When this parameter is a length 2 vector, the second element is the offset value.

The carrier frequency.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block QASK Demod Square Constellation

Equivalent M-function dmod for modulation

QASK Demod Square Constellation

Catagory Passband Demodulation/Demapping

Location Digital Mo/Dem, Passband Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Demod Square Constellation block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the QM DSB-SC ADM and QASK Demap Square Constellation blocks. Refer to their individual reference pages for descriptions of the techniques involved.

The QASK Demod Square Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

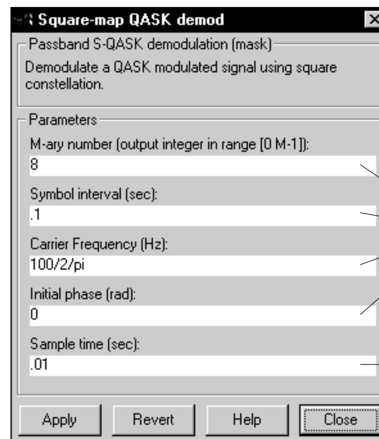
```
Td > 1/fc > 2*Ts
```

This block uses a default 5th order Butterworth lowpass filter, which it generates using the command:

```
[num, den] = butter(5, Fc*2*Ts);
```

This block accepts a modulated analog signal with a maximum amplitude equal to one. The output of this block is an integer in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding QASK Mod Square Constellation block. The offset value in **Symbol interval** can be different.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Mod Square Constellation

Equivalent M-function

ddemod for demodulation

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

Description The QASK (quadrature amplitude shift-keying) Mod Circle Constellation block maps an input signal into a circle constellation signal space and then modulates the mapped signal. This block is a combination of the QASK Map Circle Constellation and QM DSB AM blocks. Refer to their reference pages for descriptions of the techniques involved.

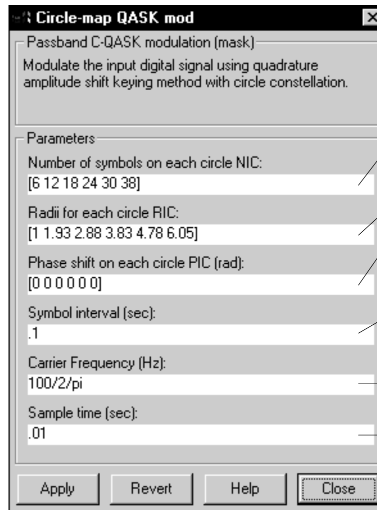
QASK Mod Circle Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the circle constellation. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

The modulation initial phase defaults to zero in this block. If you want to use a different value, use the QASK Map Circle Constellation and QM DSB AM blocks separately.

Dialog Box



The number of symbols on each circle. The elements of this vector must be positive integers.

The radius and phase shift for each circle. The length of each of these vectors must equal the length of the **Number of symbols on each circle** vector.

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value.

The signal's carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demod Circle Constellation

Equivalent M-function

dmod for modulation

QASK Demod Circle Constellation

Catagory Passband Demodulation/Demapping

Location Digital Mo/Dem, Passband Sublibrary

Description

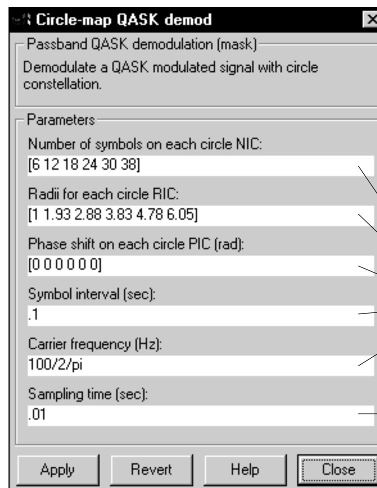
The QASK (quadrature amplitude shift-keying) Demod Circle Constellation block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the QM DSB-SC ADM and QASK Demap Circle Constellation blocks. Refer to their reference pages for descriptions of the techniques involved.

This block uses a default 5th order Butterworth filter, which it generates by using the command:

```
[num, den] = butter(5, Fc*2*Ts);
```

The input to this block is an analog modulated signal with a maximum amplitude equal to one. It outputs integers in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MASK Mod block.The offset in **Symbol interval** may be different.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Mod Circle Constellation

Equivalent M-function

`ddemod` for demodulation computation

QASK Mod Arbitrary Constellation

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

Description

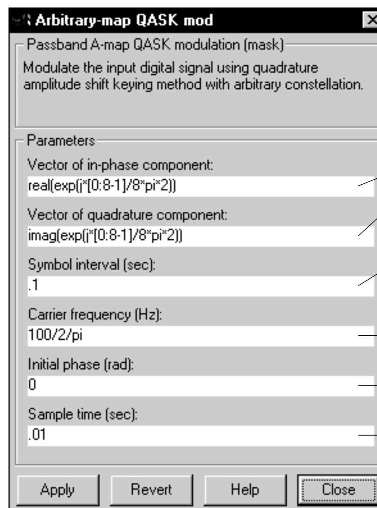
The QASK (quadrature amplitude shift-keying) Mod Arbitrary Constellation block maps an input signal into a user-defined constellation signal space and then modulates the mapped signal. This block is a combination of the QASK Map Arbitrary Constellation and QM DSB AM blocks. Refer to their reference pages for descriptions of the techniques involved.

QASK Mod Arbitrary Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the user-defined arbitrary constellation. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Define the in-phase and quadrature components of each point in the constellation.

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value

The carrier frequency.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demod Arbitrary Constellation

Equivalent M-function

dmod for modulation

QASK Demod Arbitrary Constellation

Catagory

Passband Demodulation/Demapping

Location

Digital Mo/Dem, Passband Sublibrary

Description

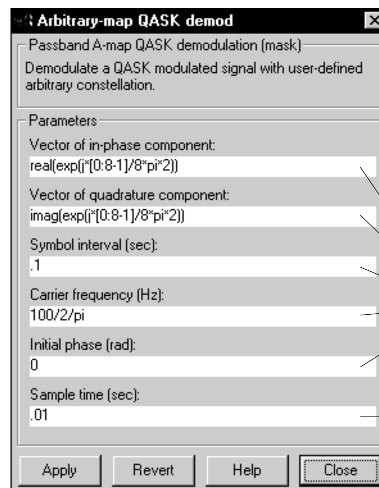
The QASK (quadrature amplitude shift-keying) Demod Arbitrary Constellation block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the QM DSB-SC ADM and QASK Demap Arbitrary Constellation blocks. Refer to their reference pages for descriptions of the techniques involved.

This block uses a default 5th order Butterworth filter, which it generates by using the command:

```
[num, den] = butter(5, Fc*2*Ts);
```

The input to this block is an analog modulated signal with a maximum amplitude equal to one. It outputs integers in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the constellation; it is also equal to the length of the evocator of in-phase components.

Dialog Box



Match these parameters to the ones used in the corresponding QASK Mod block. The offset value in **Symbol interval** can be different.

The calculation sample time. By the Nyquist sampling theorem, 1`/sample_time > 2*carrier_frequency`.

Characteristics	No. of Inputs/Outputs	1/1
	Vectorized Inputs/Outputs	No/Yes
	Time Base	Discrete time
	States	N/A
	Direct Feedthrough	Yes

Pair Block QASK Mod Arbitrary Constellation

Equivalent M-function `ddemod` for demodulation computation

MFSK Mod

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

Description

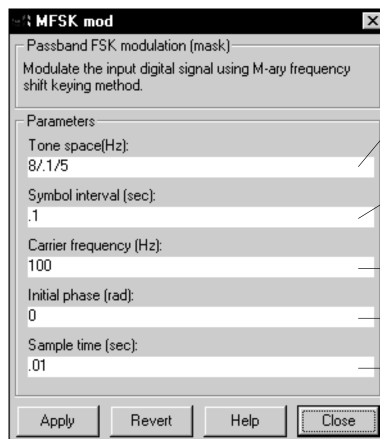
The MFSK (multiple frequency shift-keying) Mod block maps an input signal and then modulates the mapped signal. This block is a combination of the MFSK Map and FM blocks. Refer to the individual reference pages for these blocks for descriptions of the techniques involved.

MFSK Mod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



The frequency separation between two neighboring input integers. The tone space is also known as the *frequency separation*.

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value

The carrier frequency.

The initial phase of the carrier frequency.

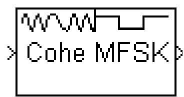
The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MFSK Demodulation

Equivalent M-function dmod



Coherent MFSK Demod

Catagory

Passband Demodulation/Demapping

Location

Digital Mo/Dem, Passband Sublibrary

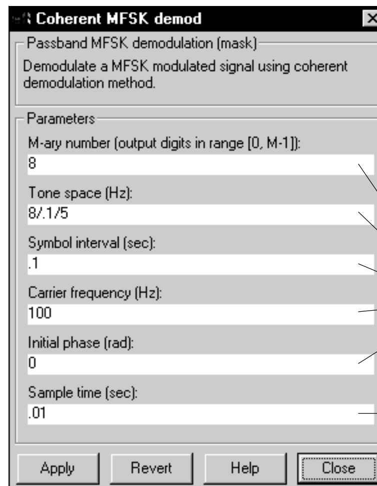
Description

The Coherent MFSK (multiple frequency shift-keying) Demod block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the Coherent MFSK Corr Demap and Min/Max Demap blocks. Refer to their individual reference pages for descriptions of the techniques involved.

Coherent MFSK Demod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

Dialog Box



Match these parameters to the ones used in the corresponding MFSK Mod block. The offset value in **Symbol interval** can be different.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics	No. of Inputs/Outputs	1/1
	Vectorized Inputs/Outputs	No/Yes
	Time Base	Discrete time
	States	N/A
	Direct Feedthrough	Yes

Pair Block MFSK Mod

Equivalent M-function demod

Noncoherent MFSK Demod

Catagory

Passband Demodulation/Demapping

Location

Digital Mo/Dem, Passband Sublibrary

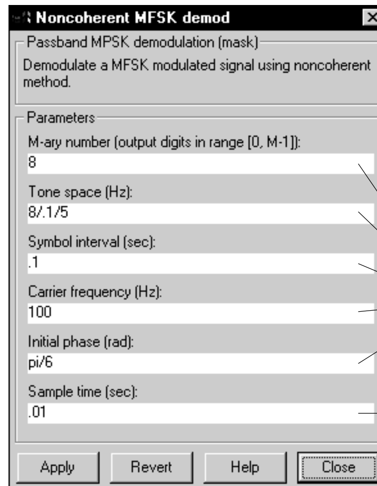
Description

The Noncoherent MFSK (multiple frequency shift-keying) Demod block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the Noncoherent MFSK Corr Demap and Min/Max Demap Arbitrary Constellation blocks. Refer to their individual reference pages for descriptions of the techniques involved.

Noncoherent MFSK Demod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

Dialog Box



Match these parameters to the ones used in the corresponding MFSK Mod block. The offset value in **Symbol interval** can be different.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MFSK Mod

Equivalent M-function `ddemod`

Catagory Passband Mapping/Modulation

Location Digital Mo/Dem, Passband Sublibrary

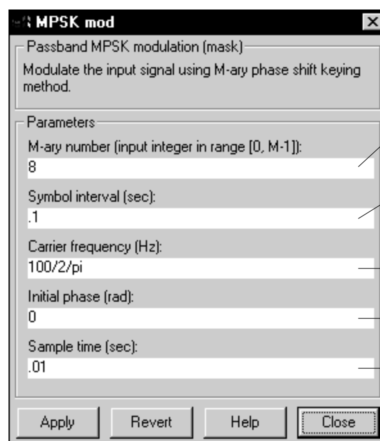
Description The MPSK (multiple phase shift-keying) Mod block maps an input signal and then modulates the mapped signal. This block is a combination of the MPSK Map and PM blocks. Refer to their individual reference pages for descriptions of the techniques involved.

MPSK Mod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

The input signal to this block is an integer in the range [0, M-1], where M is the multiple number. The output is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Specify the range of the input digits. The range is [0, M-1].

The sample time of the transmitted digit symbol. When this parameter is a two-element vector, the second element is the offset value.

The frequency of the carrier signal.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MPSK Demod

Equivalent M-function dmod

Catagory Passband Demodulation/Demapping

Location Digital Mo/Dem, Passband Sublibrary

Description

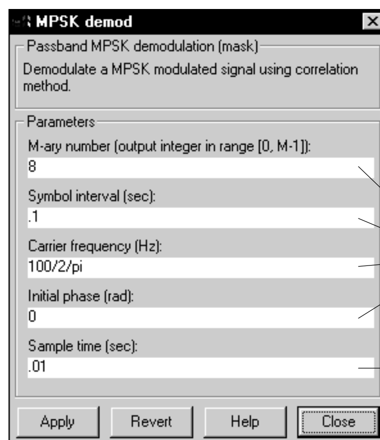
The MPSK (multiple phase shift-keying) Demod block demodulates an input signal and then demaps the demodulated signal. This block is a combination of the MPSK Corr Demap and Min/Max Demap blocks. Refer to their individual reference pages for descriptions of the techniques involved.

MPSK Demod is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

The input signal to this block is a modulated analog signal with a maximum amplitude equal to one. The output is an integer in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MPSK Mod block. The offset value in **Symbol interval** may be different.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MPSK Mod

Equivalent M-function `ddemod`

Digital Modulation/Demodulation Methods for Baseband Simulation

The blocks included in the Digital Modulation/Demodulation, Baseband Sublibrary are the baseband counterparts of the blocks introduced in the Digital Modulation/Demodulation, Passband Simulation Sublibrary.

You must specify the numerators and denominators of the lowpass filter's transfer function when using the MASK and QASK demodulation blocks. Read the introduction of the last section for possible choices. You can simply set both the numerator and denominator to 1 for your simulation if you choose not to use a filter.

The figure below shows the Digital Modulation/Demodulation Baseband Sublibrary:

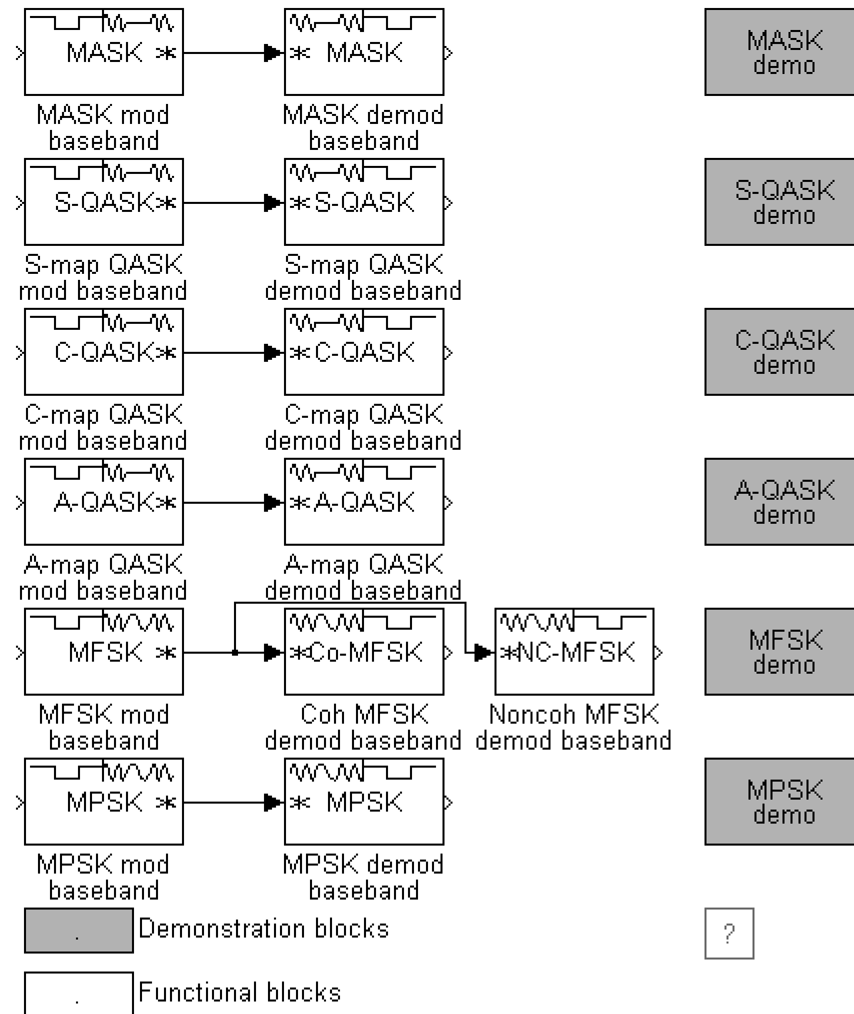


Figure 6-1: The Digital Modulation/Demodulation Baseband Sublibrary

Digital Modulation/Demodulation Baseband Reference Table

This table lists the blocks in the Digital Mod/Demod Mapping/Demapping Baseband sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
Coherent MFSK Demod CE	Multiple frequency shift-keying (MFSK) baseband coherent demodulation followed by min/max demapping
MASK Demod CE	Double sideband amplitude demodulation (DSB ADM) followed by multiple amplitude shift-keying (MASK) baseband demodulation
MASK Mod CE	Multiple amplitude shift-keying (MASK) baseband mapping followed by double sideband amplitude modulation (DSB AM)
MFSK Mod CE	Multiple frequency shift-keying (MFSK) baseband mapping followed by frequency modulation (FM)
MPSK Demod CE	Multiple phase shift-keying (MPSK) baseband demodulation followed by min/max demapping
MPSK Mod CE	Multiple phase shift-keying baseband mapping followed by phase modulation (PM)
Noncoherent MFSK Demod CE	Multiple frequency shift-keying baseband noncoherent demodulation followed by min/max demapping

Block Name	Description
QASK CE Demod Arbitrary Constellation	Double sideband suppressed carrier amplitude demodulation (DSB-SC ADM) followed by quadrature amplitude shift-keying (QASK) demodulation using an arbitrary constellation
QASK Demod CE Circle Constellation	Double sideband suppressed carrier amplitude demodulation (DSB-SC ADM) followed by quadrature amplitude shift-keying (QASK) demodulation using a circle constellation
QASK Demod CE Square Constellation	Double sideband suppressed carrier amplitude demodulation (DSB-SC ADM) followed by quadrature amplitude shift-keying (QASK) demodulation using a square constellation
QASK Mod CE Arbitrary Constellation	Quadrature amplitude shift-keying (QASK) modulation using an arbitrary constellation followed by double sideband suppressed carrier amplitude modulation (DSB AM)
QASK Mod CE Circle Constellation	Quadrature amplitude shift-keying (QASK) modulation using a circle constellation followed by double sideband suppressed carrier amplitude modulation (DSB AM)
QASK Mod Square Constellation	Quadrature amplitude shift-keying (QASK) modulation using a square constellation followed by double sideband suppressed carrier amplitude modulation (DSB AM)

Catagory

Baseband Mapping/Modulation

Location

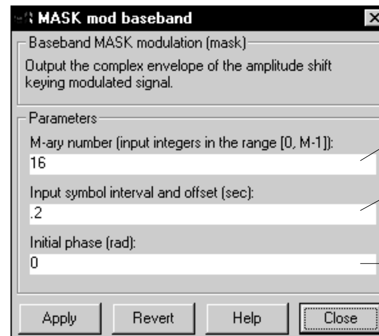
Digital Mo/Dem, Baseband Sublibrary

Description

The MASK (multiple amplitude shift-keying) Mod CE block maps an input signal and then modulates the mapped signal with a complex envelope. This block is a combination of the MASK Map and DSB-SC AM CE blocks. Refer to the reference pages for these blocks for a discussion of the techniques involved.

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. Typically, M equals 2^K, where K is a positive integer. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Specify the input digit range, [0, M-1].

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

The initial phase of the carrier signal.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Complex
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MASK Demod CE

Equivalent M-function

`dmodce`

MASK Demod CE

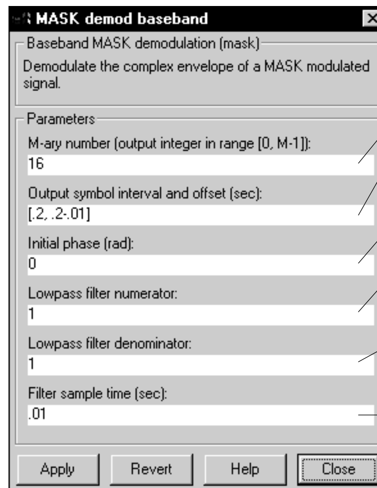
Catagory Baseband Demodulation/Demapping

Location Digital Mo/Dem, Baseband Sublibrary

Description The MASK (multiple amplitude shift-keying) Demod CE block demodulates with complex envelope an input signal and then the demaps the demodulated signal. This block is a combination of the Costas PLL DSB ADM CE and the MASK Demap blocks. Refer to the individual reference pages for these blocks for discussions of the techniques involved.

This block accepts a modulated complex signal with its maximum amplitude equal to one. It outputs integers in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MASK Mod Ce block. The offset value in **Symbol interval** can be different.

The initial phase of the carrier signal.

Specify the numerator of the lowpass filter used in the demodulation. Set this parameter to 1 if you do not need a lowpass filter.

The denominator of the lowpass filter. If the filter is FIR or if you do not need a lowpass filter, set this parameter to 1.

The filter sample time. The sample time of the filter is also the block calculation sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MASK Mod CE

Equivalent M-function

`ddemodce`

QASK Mod CE Square Constellation

Catagory Baseband Mapping/Modulation

Location Digital Mo/Dem, Baseband Sublibrary

Description

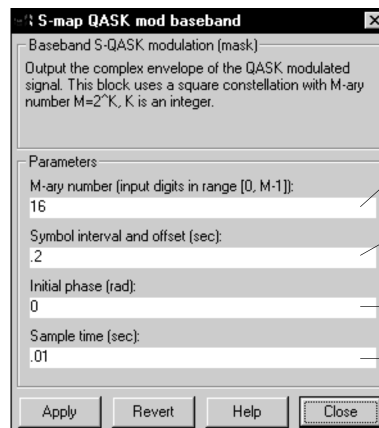
The QASK (quadrature amplitude shift-keying) Mod Square CE Constellation block maps an input signal into a square constellation signal space and then modulates the mapped signal with complex envelope. This block is a combination of the QASK Map Square Constellation and QM DSB AM CE blocks. Refer to their individual reference pages for descriptions of the techniques involved.

QASK Mod CE Square Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the square constellation and typically is equal to 2^K, where K is a positive integer. The output of this block is a modulated analog complex signal with a maximum amplitude of both the in-phase and quadrature components equal to one.

Dialog Box



Specify the input range. The input digit is in the integer range [0,M-1].

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Complex
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

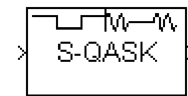
Pair Block

QASK Demod CE Square Constellation

Equivalent M-function

dmodce

QASK Demod CE Square Constellation



Catagory Baseband Mapping/Modulation

Location Digital Mo/Dem, Baseband Sublibrary

Description

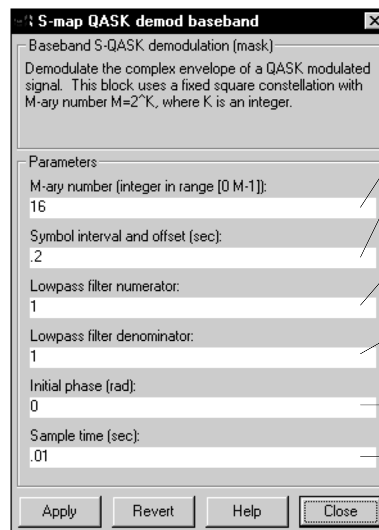
The QASK (quadrature amplitude shift-keying) Demod CE Square Constellation block demodulates a complex input signal and then demaps the demodulated signal using QASK. This block is a combination of the QM DSB-SC ADM CE and QASK Demap Square Constellation blocks. Refer to their individual reference pages for descriptions of the techniques involved.

QASK Demod Square Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a modulated complex analog signal with a maximum amplitude equal to one in both the in-phase and quadrature components. The output of this block is an integer in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding QASK Mod CE Square Constellation block. The offset value in **Symbol interval** can be different.

Specify the numerator of the lowpass filter used in the demodulation. Set this parameter to 1 if you do not need a lowpass filter.

The denominator of the lowpass filter. If the filter is FIR or if you do not need a lowpass filter, set this parameter to 1.

The initial phase of the carrier signal.

The filter sample time, which also specifies the calculation sample time.

Characteristics	No. of Inputs/Outputs	1/1
	Vectorized Inputs/Outputs	Complex/No
	Time Base	Discrete time
	States	N/A
	Direct Feedthrough	Yes

Pair Block QASK Mod CE Square Constellation

Equivalent M-function `ddemodce`

QASK Mod CE Circle Constellation

Catagory Baseband Mapping/Modulation

Location Digital Mo/Dem, Baseband Sublibrary

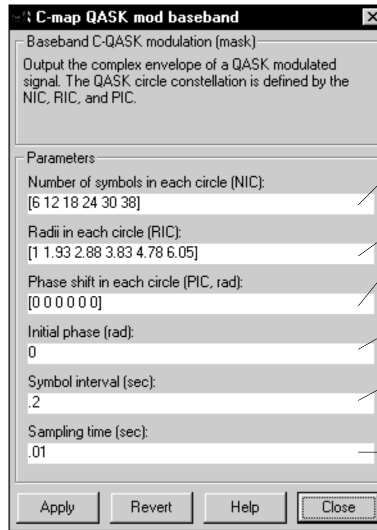
Description The QASK (quadrature amplitude shift-keying) Mod CE Circle Constellation block maps a complex input signal into a circle constellation signal space and then modulates the mapped signal. This block is a combination of the QASK Map Circle Constellation and QM DSB AM CE blocks. Refer to their reference pages for descriptions of the techniques involved.

QASK Mod CE Circle Constellation is a passband simulation block. There are three timing variables in this block: duration of digit (T_d), carrier frequency (f_c), and sample time (T_s). You must select values for these variables so that they satisfy the mathematical relations below:

```
Td > 1/fc > 2*Ts
```

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the circle constellation. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



The number of symbols on each circle. The elements of this vector must be positive integers.

The radius and phase shift for each circle. The length of this vector must equal the length of the **Number of symbols in each circle** vector.

The initial phase of the carrier signal.

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value.

The calculation sampling time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Complex
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demod CE Circle Constellation

Equivalent M-function

`dmodce`

QASK Demod CE Circle Constellation

Catagory Baseband Demodulation/Demapping

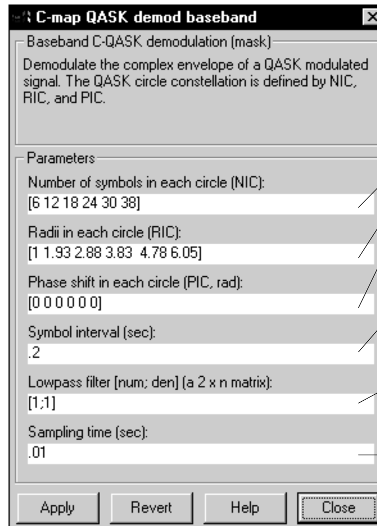
Location Digital Mo/Dem, Baseband Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Demod CE Circle Constellation block demodulates a complex input signal and then demaps the demodulated signal. This block is a combination of the QM DSB-SC ADM CE and QASK Demap Circle Constellation blocks. Refer to their individual reference pages for descriptions of the techniques involved.

The input to this block is an analog modulated signal with a maximum amplitude equal to one. It outputs integers in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MASK Mod CE Circle Constellation block. The offset value in **Symbol interval** can be different.

Specify the numerator of the lowpass filter used in the demodulation. Set this parameter to 1 if you do not need a lowpass filter.

The denominator of the lowpass filter. If the filter is FIR or if you do not need a lowpass filter, set this parameter to 1.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Mod CE Circle Constellation

Equivalent M-function

`ddemodce`

QASK Mod CE Arbitrary Constellation

Catagory Baseband Mapping/Modulation

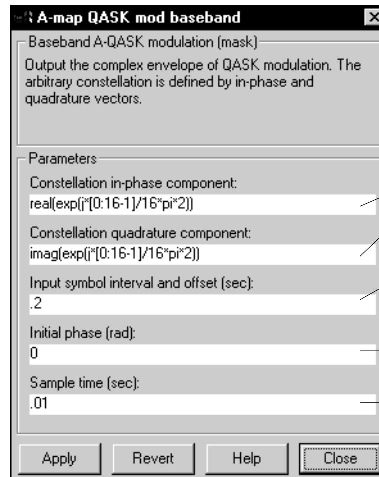
Location Digital Mo/Dem, Baseband Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Mod CE Arbitrary Constellation block maps a complex input signal into a user-defined constellation signal space and then modulates the mapped signal. This block is a combination of the QASK Map Arbitrary Constellation and QM DSB AM CE blocks. Refer to their individual reference pages for descriptions of the techniques involved.

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the user-defined arbitrary constellation. The output of this block is a modulated analog signal with a maximum amplitude equal to one.

Dialog Box



Define the in-phase and quadrature components of each point in the constellation.

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Complex
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

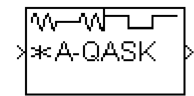
Pair Block

QASK Demod CE Arbitrary Constellation

Equivalent M-function

`dmodce`

QASK Demod CE Arbitrary Constellation



Catagory Baseband Demodulation/Demapping

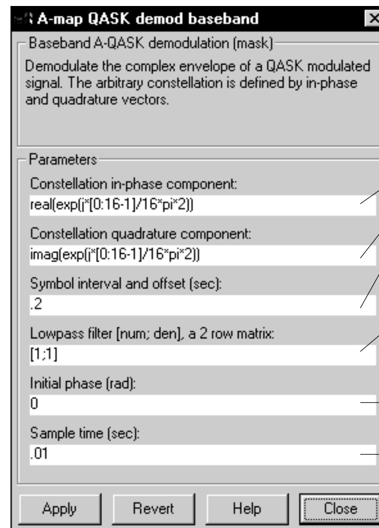
Location Digital Mo/Dem, Baseband Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Demod CE Arbitrary Constellation block demodulates a complex input signal and then demaps the demodulated signal. This block is a combination of the QM DSB-SC ADM CE and QASK Demap Arbitrary Constellation blocks. Refer to their reference pages for descriptions of the techniques involved.

The input to this block is an analog modulated signal with a maximum amplitude equal to the element of maximum absolute value in the vector of in-phase component. It outputs integers in the range [0, M-1], where M is the multiple number. The multiple number equals the total number of points in the constellation; it is also equal to the length of the vector of in-phase components.

Dialog Box



Match these parameters to the ones used in the corresponding QASK Mod CE Arbitrary Constellation block. The offset value in **Symbol interval** can be different.

Specify the numerator and denominator of the lowpass filter used in the demodulation. Set this parameter to 1 if you do not need a lowpass filter.If the filter is FIR set the denominator to 1.

The initial phase of the carrier signal.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Mod CE Arbitrary Constellation

Equivalent M-function

`ddemodce`

MFSK Mod CE

Catagory

Baseband Mapping/Modulation

Location

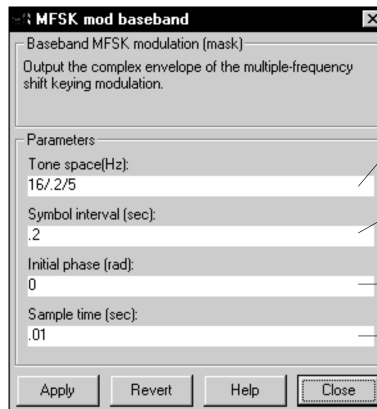
Digital Mo/Dem, Baseband Sublibrary

Description

The MFSK (multiple frequency shift-keying) Mod CE block maps a complex input signal and then modulates the mapped signal. This block is a combination of the MFSK Map and FM CE blocks. Refer to the individual reference pages for these blocks for descriptions of the techniques involved.

This block accepts a scalar input in the range [0, M-1], where M is the multiple number. The output of this block is a modulated complex analog signal with a maximum amplitude equal to one.

Dialog Box



The frequency separation between two neighboring input integers. The tone space is also known as the ***frequency separation***.

The sample time of the input digit. When this parameter is a two-element vector, the second element is the offset value

The initial phase of the carrier signal.

The calculation sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Complex
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

Coherent MFSK Demod CE
Noncoherent MFSK Demod CE

Equivalent M-function

`dmodce`

Coherent MFSK Demod CE

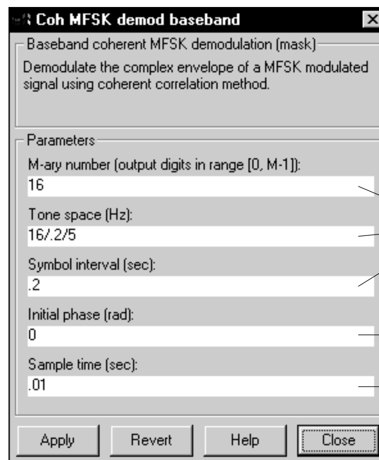
Catagory Baseband Demodulation/Demapping

Location Digital Mo/Dem, Baseband Sublibrary

Description The Coherent MFSK (multiple frequency shift-keying) Demod CE block demodulates a complex input signal and then demaps the demodulated signal. This block is a combination of the Coherent MFSK Corr Demap CE and Min/Max Demap blocks. Refer to their individual reference pages for descriptions of the techniques involved.

The input to this block is a modulated complex analog signal. The output is an integer in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MFSK Mod CE block. The offset value in **Symbol interval** may be different.

The initial phase of the carrier signal.

The calculation sample time.

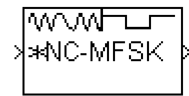
Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MFSK Mod CE

Equivalent M-function `ddemodce`

Noncoherent MFSK Demod CE

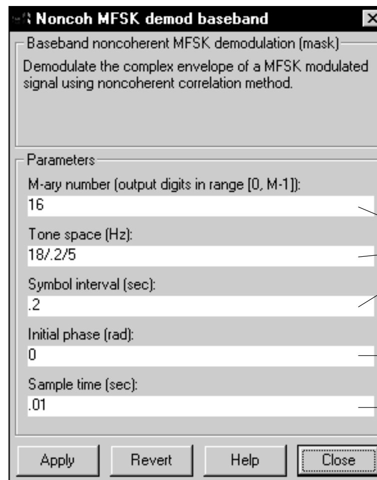


Catagory Baseband Demodulation/Demapping

Location Digital Mo/Dem, Baseband Sublibrary

Description The Noncoherent MFSK (multiple frequency shift-keying) Demod CE block demodulates a complex input signal and then demaps the demodulated signal. This block is a combination of the Noncoherent MFSK Corr Demap CE and Min/Max Demap Arbitrary Constellation blocks. Refer to their individual reference pages for descriptions of the techniques involved.

Dialog Box



Match these parameters to the ones used in the corresponding MFSK Mod CE block. In **Symbol interval**, the offset value may be different.

The initial phase of the carrier signal.

The calculation sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MFSK Mod CE

Equivalent M-function `ddemodce`

MPSK Mod CE

Catagory

Baseband Mapping/Modulation

Location

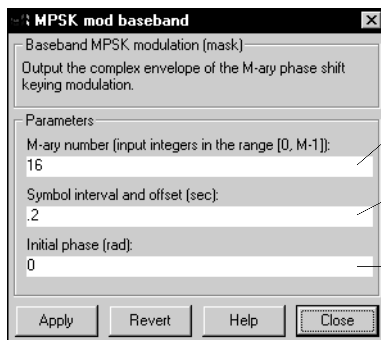
Digital Mo/Dem, Baseband Sublibrary

Description

The MPSK (multiple phase shift-keying) Mod CE block maps an input signal and then modulates the mapped signal with a complex envelope. This block is a combination of the MPSK Map and PM CE blocks. Refer to their individual reference pages for descriptions of the techniques involved.

The input signal to this block is an integer in the range [0, M–1], where M is the multiple number. The output is a modulated complex analog signal with a maximum amplitude equal to one.

Dialog Box



Specify the range of the input digits. The range is [0, M-1].

The sample time of the transmitted symbol. When this parameter is a two-element vector, the second element is the offset value.

The initial phase of the carrier frequency.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MPSK Demod CE

Equivalent M-function

`dmodce`

MPSK Demod CE

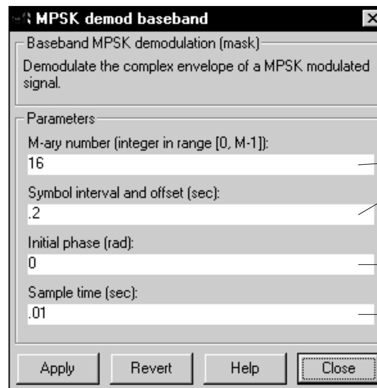
Catagory Baseband Demodulation/Demapping

Location Digital Mo/Dem, Baseband Sublibrary

Description The MPSK (multiple phase shift-keying) Demod CE block demodulates a complex input signal and then demaps the demodulated signal. This block is a combination of the MPSK Corr Demod CE and Min/Max Demap blocks. Refer to their individual reference pages for descriptions of the techniques involved.

The input signal to this block is a modulated complex analog signal with a maximum amplitude equal to one. The output is an integer in the range [0, M-1], where M is the multiple number.

Dialog Box



Match these parameters to the ones used in the corresponding MPSK Mod CE block. The offset value in **Symbol interval** can be different.

The initial phase of the carrier signal.

The calculation sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MPSK Mod CE

Equivalent M-function `ddemodce`

Mapping/Demapping for Digital Modulation/ Demodulation

Digital signal mapping codes an input digital signal into a signal ready for modulation. Digital signal demapping decodes a demodulated signal back into a digital signal. A digital modulation block can be divided into two major parts: digital-to-analog mapping and analog modulation. A digital demodulation block can also be divided into two major parts: analog demodulation and analog-to-digital demapping.

The mapping and demapping blocks are in two sublibraries: the Digital Mo/Dem Map/Demap Sublibrary and Digital Mo/Dem-CE Map/Demap Sublibrary. This section discusses both baseband and passband simulation sublibraries because these two sublibraries share quite a few blocks.

Note: The mapping blocks used in both baseband and passband have no functionality differences, so each of these mapping blocks is documented only for passband simulation. There are no differences in how you use these blocks in baseband.

The figure below shows the Passband Digital Mod/Dem Map/Demap Sublibrary:

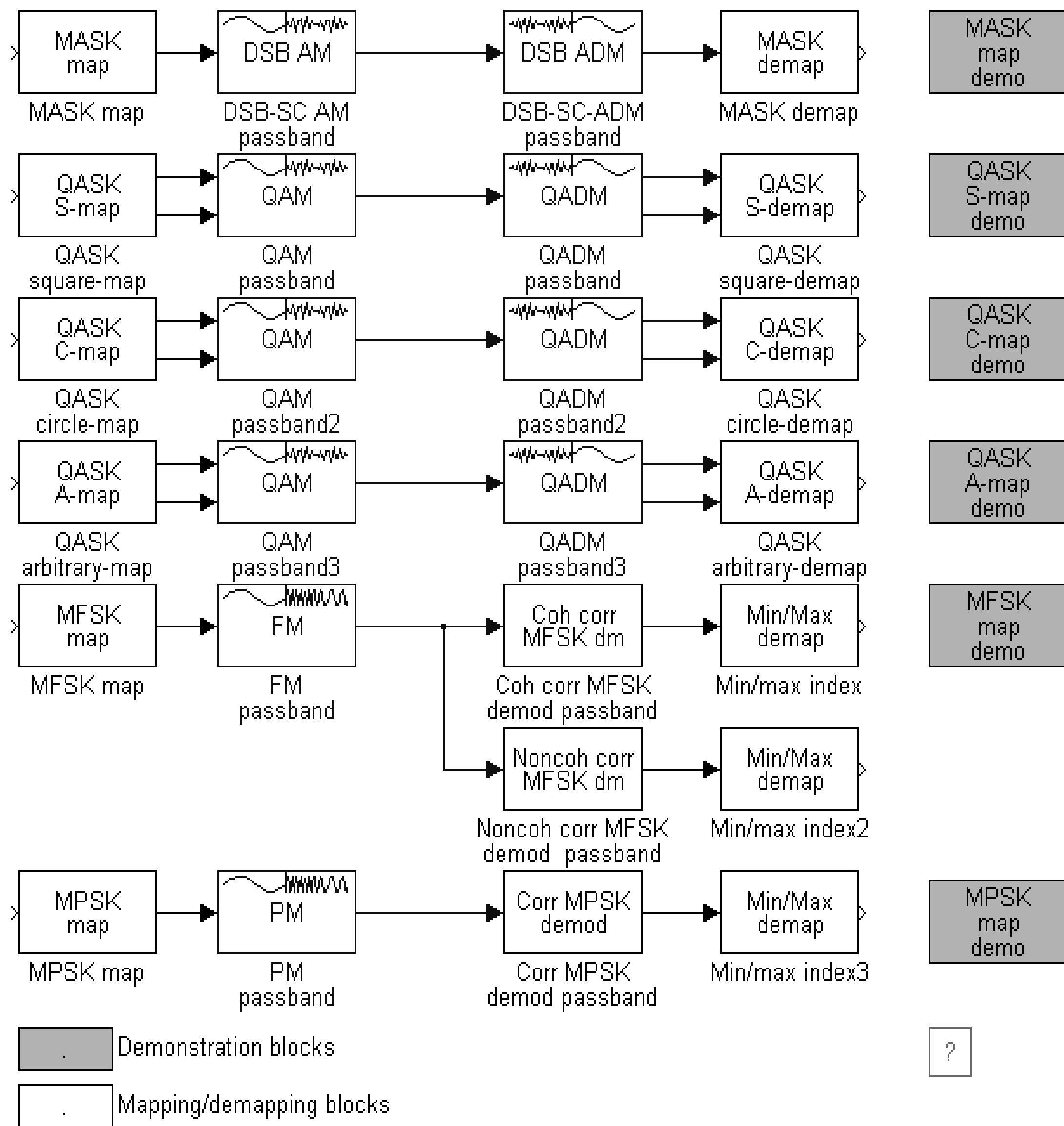


Figure 6-17: The Passband Digital Modulation/Demodulation Mapping/Demapping Sublibrary

The figure below shows the Baseband Digital Mod/Demod Mapping/Demapping Sublibrary:

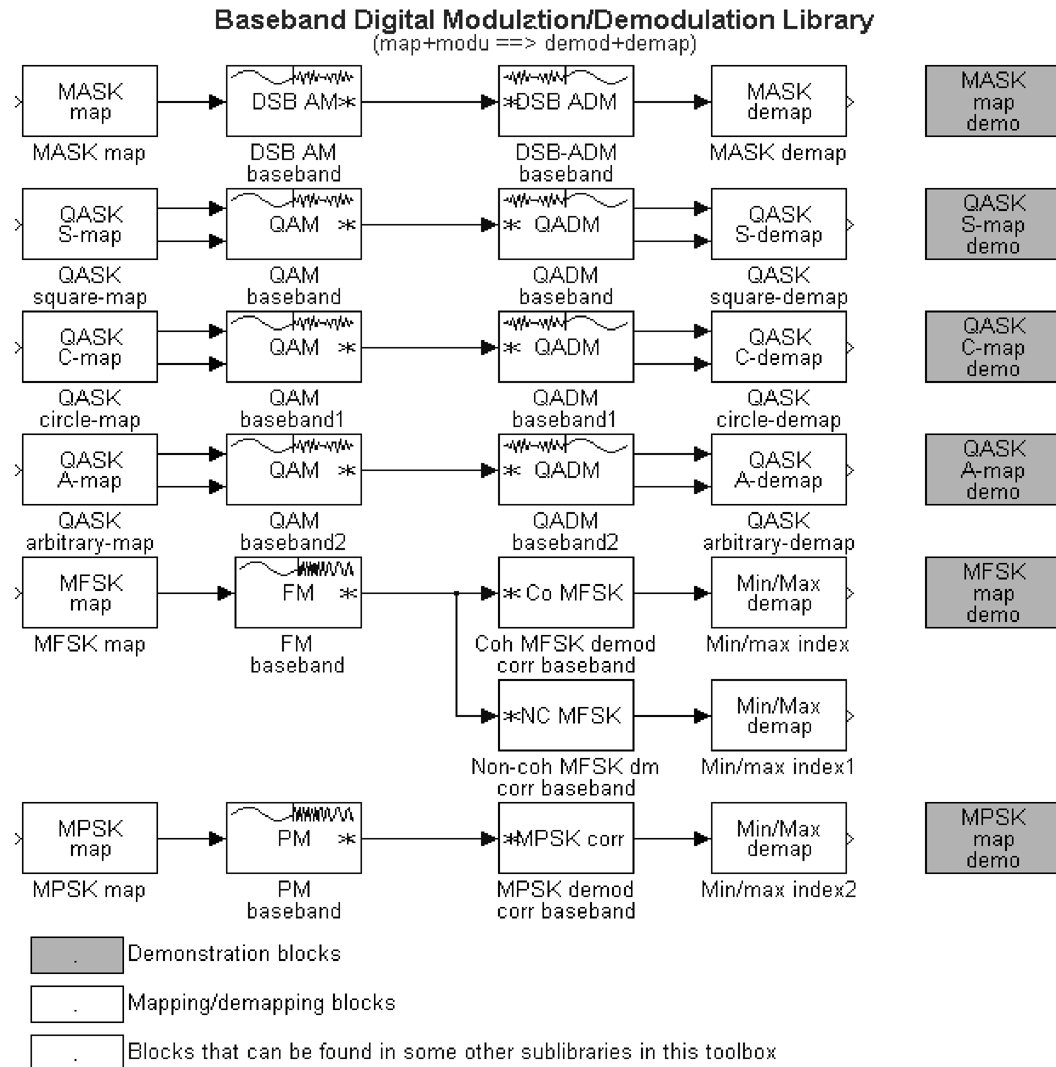


Figure 6-18: The Baseband Digital Modulation/Demodulation Mapping/Demapping Sublibrary

This toolbox uses the term *symbol interval* in digital mapping/demapping as well as digital modulation/demodulation. This term refers to the digital sampling time. Please note that the symbol interval for the transmitting side and the receiving side should be exactly the same to obtain the correct results. The parameter **Symbol interval** can be a two-element vector, in which case the second element is either:

- the transmitting offset time for a mapping block
- the decision offset time for a demapping block

Usually, the transmitting offset is different from the decision offset. The difference may be caused by distortion, interference, filtering and other reasons in the transmitting and receiving process. In the figure below, the upper graph is the transmitted signal; the lower graph is the received signal. Note that the decision offset is determined based on the characteristics of the transmitting system. The eye-pattern plot is useful in determining the decision offset. The unit for both symbol interval and offset is in seconds.

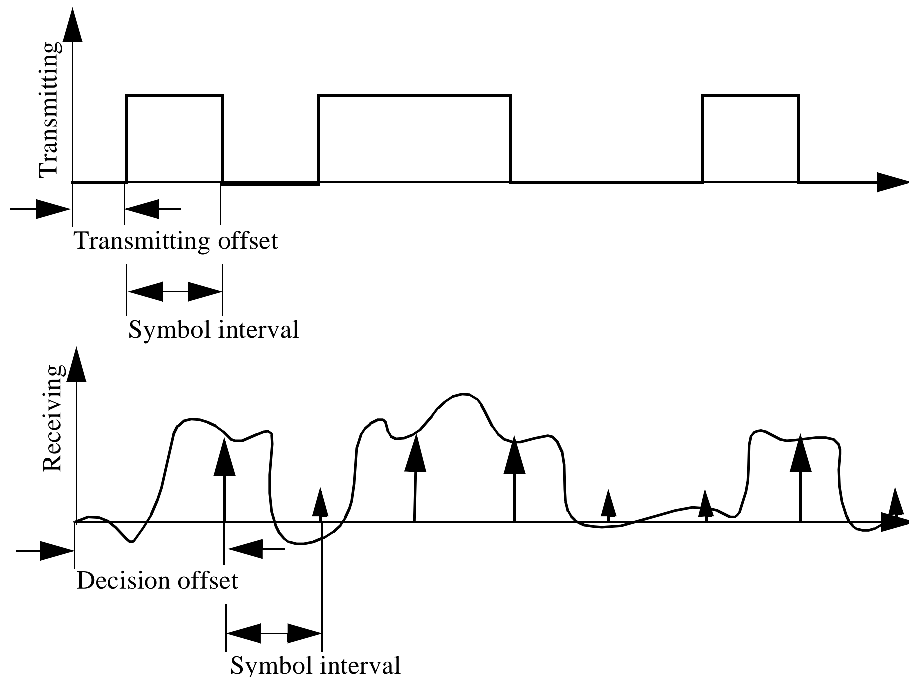


Figure 6-19: The Symbol Interval and Its Offset in Transmission and Reception

This toolbox assumes the digits in the transmitting are non-negative integers. The integers in the transmitting can be 0, 1, 2,..., M-1. Under these assumptions, M is called the *M-ary number*.

Passband and Baseband Digital Mapping/Demapping Reference Table

This table lists the Simulink mapping and demapping blocks used in both the passband and baseband versions of the “Digital Mod/Demod Mapping/ Demapping” sublibraries. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
MASK Demap	Multiple amplitude shift-keying (MASK) demapping
MASK Map	Multiple amplitude shift-keying (MASK) mapping
MFSK Map	Multiple frequency shift-keying (MFSK) mapping
Min/Max Demap	Minimum/maximum demapping
QASK Demap Arbitrary Constellation	Quadrature amplitude shift-keying (QASK) demapping using an arbitrary constellation
QASK Demap Circle Constellation	Quadrature amplitude shift-keying (QASK) demapping using a circle constellation
QASK Demap Square Constellation	Quadrature amplitude shift-keying (QASK) demapping using a square constellation
QASK Map Arbitrary Constellation	Quadrature amplitude shift-keying (QASK) mapping using an arbitrary constellation

Block Name	Description
QASK Map Circle Constellation	Quadrature amplitude shift-keying (QASK) mapping using a circle constellation
QASK Map Square Constellation	Quadrature amplitude shift-keying (QASK) mapping using a square constellation

MASK
map

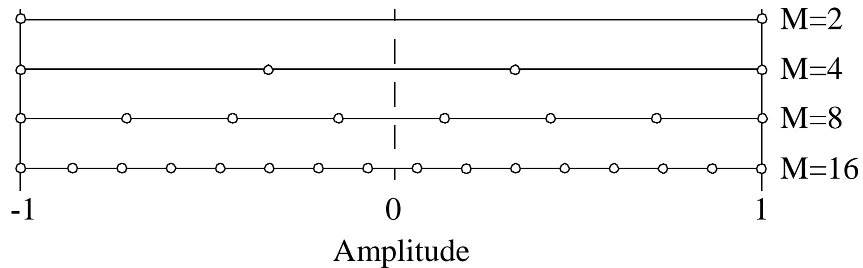
MASK Map

Catagory Modulation Map

Location Digital Mo/Dem Map/Demap Sublibrary

Description

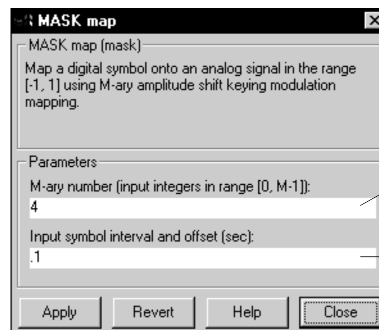
The MASK (multiple amplitude shift-keying) Map block modulates an input message signal. MASK is a one-dimensional coding scheme that codes integers as amplitude shifts. A signal set for M-ary numbers 2, 4, 8, and 16 is:



This block assumes a maximum amplitude normalized to one; if you require a different maximum value, place a gain block after this block.

This block is a discrete-time block; it codes the input signal only at the sampling time point. There is, however, a mechanism provided to convert digital signals to analog signals. On account of this, the output signal of this block is ready to use with analog modulation blocks.

Dialog Box



Specify the number of input symbols. The input symbols are in the integer range [0,M-1]. Generally $M=2^K$, where K is a positive integer.

The sample time each input symbol. When this parameter is a two-element vector, the second element is the offset value.

MASK Map

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MASK Demap

Equivalent M-function

`modmap`

MASK Demap

Catagory Modulation Demap

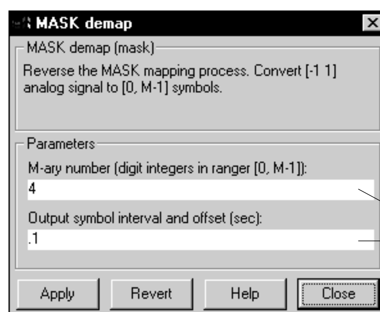
Location Digital Mo/Dem Map/Demap Sublibrary

Description The MASK (multiple amplitude shift-keying) Demap block recovers a message signal from a modulated input signal. The parameters set in the dialog box of this block must match those used in the corresponding MASK Map block to obtain the correct result.

This block is the inverse of the MASK Map block. It accepts a signal in the [-1, 1] range and outputs a number in the [0, M-1] range, where M is the M-ary number that was used in the MASK Map block.

The MASK Demap block is a discrete-time block and decodes the input signal only at the sampling time points, but it has a mechanism to convert analog signals to digital signals. On account of this, it is possible to connect this digital block to analog blocks.

Dialog Block



Match these parameters to the ones in the corresponding MASK Map block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block MASK Map

Equivalent M-function demodmap

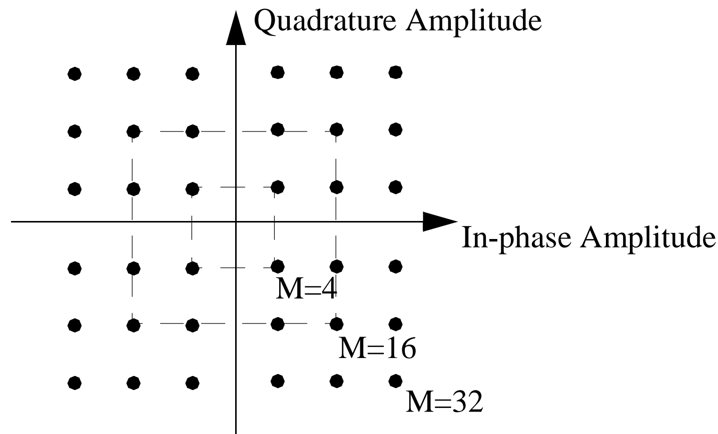
QASK S-map

QASK Map Square Constellation

Catagory Modulation Map

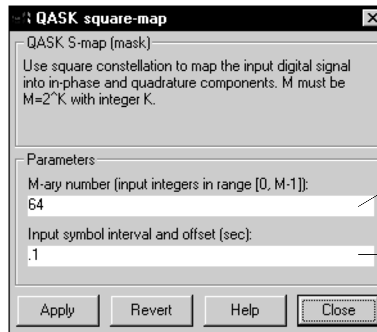
Location Digital Mo/Dem Map/Demap Sublibrary

Description The QASK (quadrature amplitude shift-keying) Map Square Constellation block converts an input message signal into a two dimensional coding signal. For M-ary number M equal to 4, 16, and 32, the square QASK constellation is:



The two axes are the in-phase and quadrature components of the mapped signal. The mapping from the input integer to the components of the two axes is a one-to-one mapping. The maximum amplitude for both components is 1; use a gain block after this block if you require a different maximum amplitude. You can use the function `qaskenco` to plot square constellations.

Dialog Box



Specify the input range. The input symbols are in the integer range [0,M-1]. Generally $M=2^K$, where K is a positive integer.

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demap Square Constellation

Equivalent M-function

`qaskenco` for QASK square constellation plot

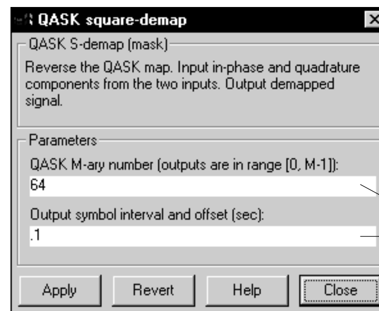
QASK Demap Square Constellation

Catagory Demodulation Demap

Location Digital Mo/Dem Map/Demap Sublibrary

Description The QASK (quadrature amplitude shift-keying) Demap Square Constellation block recovers a message signal from two input signals received from the corresponding QASK Map Square Constellation block.

Dialog Box



Match these parameters to the ones in the corresponding QASK Map block.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A

Pair Block MASK Map Square Constellation

Equivalent M-function demodmap

QASK Map Circle Constellation

Catagory

Modulation Map

Location

Digital Mo/Dem Map/Demap Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Map Circle Constellation block converts an integer message signal into a two dimensional coding signal. The constellation of the signal set is a sequence of one or more concentric circles.

Define the signal set by specifying the **Number of symbols** on each circle, the **Radii** for each circle, and the **Phase shift** for each circle in a set of three equal-length vectors.

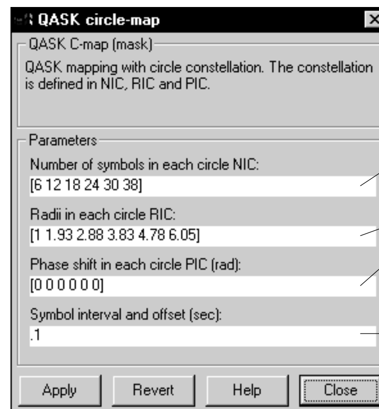
You can generate a plot of the circle constellation using the function `modmap`. For example, define the three vectors:

```
Number_of_symbol = [4 12];
Radii= [1 2];
Phase_shift = [pi/4,0];
```

Using `modmap`, generate the plot of the constellation:

```
modmap('qask/circle',Number_of_symbols,...Radii,Phase_shift);
```

Dialog Box



The number of symbols on each circle. The elements of this vector must be positive integers.

The radius and phase shift for each circle. The length of each of these vectors must equal the length of the **Number of symbols in each circle** vector.

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demap Circle Constellation

Equivalent M-function

`modmap` for mapping computation and constellation plots

QASK Demap Circle Constellation

Catagory

Demodulation Demap

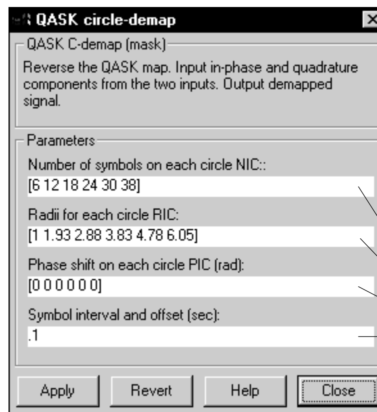
Location

Digital Mo/Dem Map/Demap Sublibrary

Description

The QASK (quadrature amplitude shift-keying) Demap Circle Constellation block recovers a message signal from two input signals received from the corresponding QASK Map Circle Constellation block.

Dialog Box



Match these parameters to the ones in the corresponding QASK Map block. The offset for the **Symbol interval** may differ.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MASK Map Circle Constellation

Equivalent M-function

demodmap

QASK Map Arbitrary Constellation

Catagory Modulation Map

Location Digital Mo/Dem Map/Demap Sublibrary

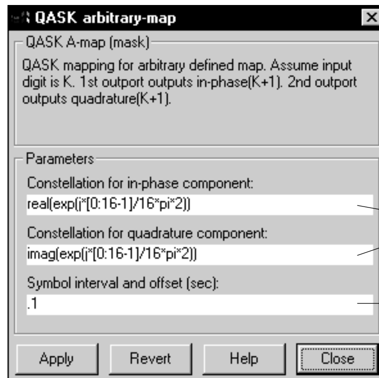
Description The QASK (quadrature amplitude shift-keying) Arbitrary Constellation block converts an input message into a two dimensional coding signal. The structure of the constellation is user-defined.

Define the points of the constellation by using two equal length vectors; the first vector defines the in-phase component of each constellation point, and the second defines the quadrature component. The M-ary number M is the total number of points defined in the constellation. M is equal to the length of either of the two vectors.

You can generate a plot of the arbitrary constellation by using the function `modmap`:

```
modmap('qask/arb', x, y);
```

Dialog Box



Define the in-phase and quadrature components of each point in the constellation.

The sample time of each input symbol. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

QASK Demap Arbitrary Constellation

Equivalent M-function

`modmap`

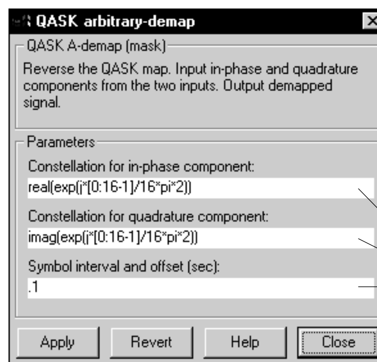
QASK Demap Arbitrary Constellation

Catagory Demodulation Demap

Location Digital Mo/Dem Map/Demap Sublibrary

Description The QASK (quadrature amplitude shift-keying) Demap Arbitrary Constellation block recovers a message signal from two input signals received from the corresponding QASK Map Arbitrary Constellation block.

Dialog Box



Match these parameters to the ones in the corresponding QASK Map block. The offset value in **Symbol interval** may differ.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block QASK Map Arbitrary Constellation

Equivalent M-function demodmap

MFSK
map

MFSK Map

Catagory

Modulation Map

Location

Digital Mo/Dem Map/Demap Sublibrary

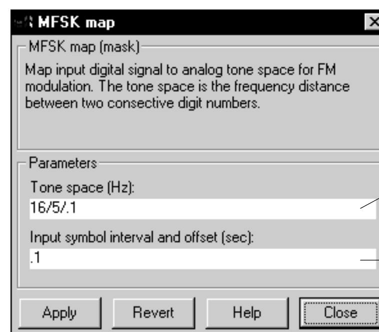
Description

The MFSK (multiple frequency shift-keying) Map block codes an input signal using frequency shift-keying.

The discrete input signal is an integer in the range [0, M-1], where M is the M-ary number. This block requires the definition of a **Tone space**, which is a scalar that specifies the frequency separation between two neighboring input integers. The modulated signal has a frequency in the range [Fc, Fc+B], where Fc is the carrier frequency and B it the bandwidth of MFSK. Note that

```
B=(M-1)*tone_space
```

Dialog Box



The frequency separation between two neighboring input integers. The **Tone space** is also known as the *frequency separation.*

The sample time of the input symbol. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MFSK Correlation Demod and Min/Max Demap

Equivalent M-function

`modmap`

Min/Max Demap

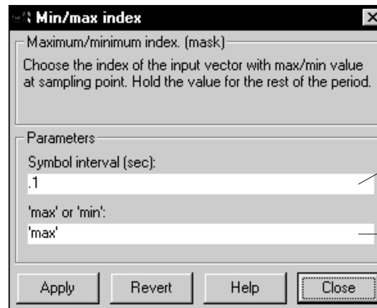
Min/Max
demap

Catagory Demodulation Demap

Location Digital Mo/Dem Map/Demap Sublibrary

Description The Min/Max Demap block finds the minimum or maximum element of an input vector and outputs (`index-1`), where `index` is the index of the element. You can use this block for both MFSK and MPSK mapping.

Dialog Box



The sample time for the output digit. When this parameter is a two-element vector, the second element is the offset value.

A string entry. Choose whether to evaluate for the maximum or minimum element.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Coherence MFSK dm

Coherent MFSK Corr Demod

Catagory Correlation Calculation

Location Digital Mo/Dem Map/Demap Sublibrary

Description The Coherent MFSK (multiple frequency shift-keying) Corr Demod block calculates the correlation between the input signal and a vector of sinusoidal signals using the coherent recovery method.

This block calculates the correlation between the input signal and a vector of sinusoidal signals of various frequencies. Each sinusoidal in the vector corresponds to a possible demapping value. The figure below show the coherent correlation MFSK demodulation method:

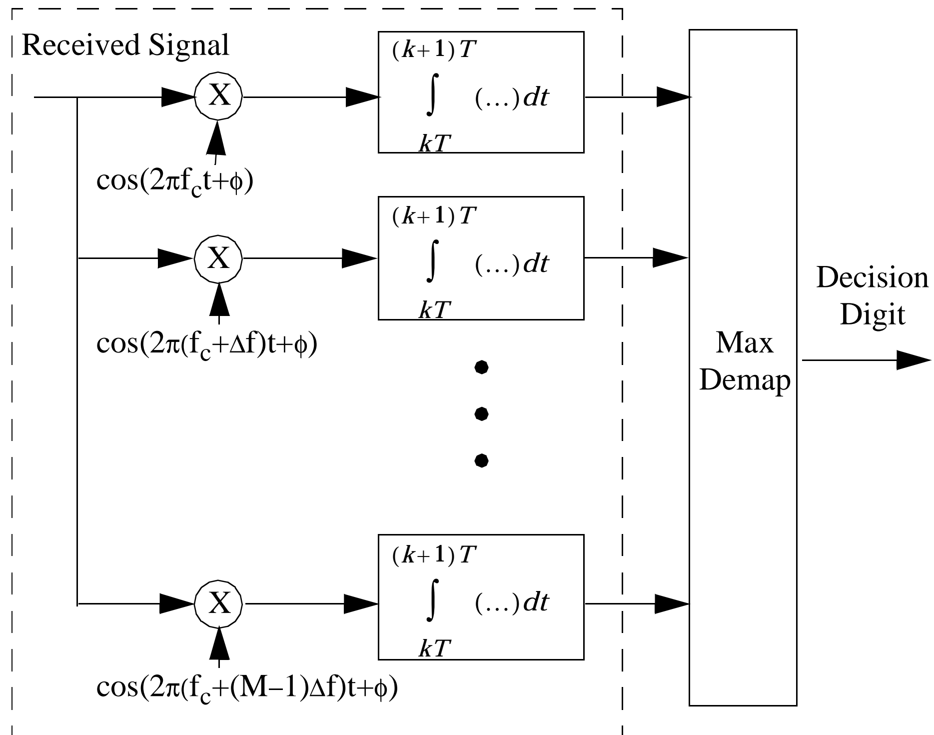
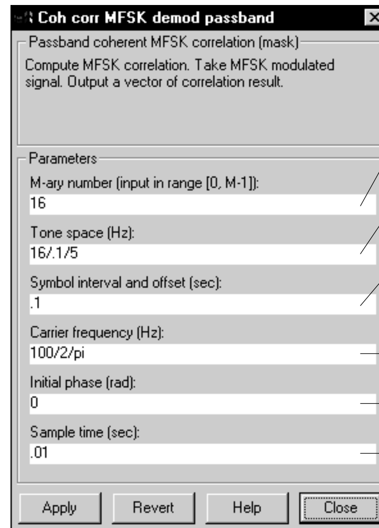


Figure 6-20: Block Diagram of Coherent MFSK Correlation Demodulation

Coherent MFSK Corr Demod

The functionality of this block is inside the dashed box. Parameter f_c is the carrier frequency, Δf is the tone space, and ϕ is the initial phase of the demodulation. M is the M-ary number, and T is the integration time interval.

Dialog Box



Specify the possible range of the input symbols. The range is [0, M-1].

The frequency separation between neighboring input symbols.

The sample time of the transmitted symbol. When this parameter is a two-element vector, the second element is the offset value. This parameter determines the integration time interval T of the correlation function.

The frequency of the carrier signal.

The initial phase of the carrier signal.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

FM with its signal input from MFSK Map

Equivalent M-function

comdemod

Noncohe
MFSK dm

Noncoherent MFSK Corr Demod

Catagory Correlation Calculation

Location Digital Mo/Dem Map/Demap Sublibrary

Description The Noncoherent MFSK Corr Demod (multiple frequency shift-keying correlation demodulation) block calculates the correlation between the input signal and a vector of sinusoidal signals using the noncoherent method.

This block calculates the correlation between the input signal and a vector of sinusoidal signals of various frequencies. Unlike the Coherent MFSK Correlation method, this method is insensitive to initial phase variations. The trade-off is that the calculation of the noncoherent correlation takes about twice as much time as that in the coherent demodulation block.

The diagram below shows the noncoherent correlation MFSK demodulation method:

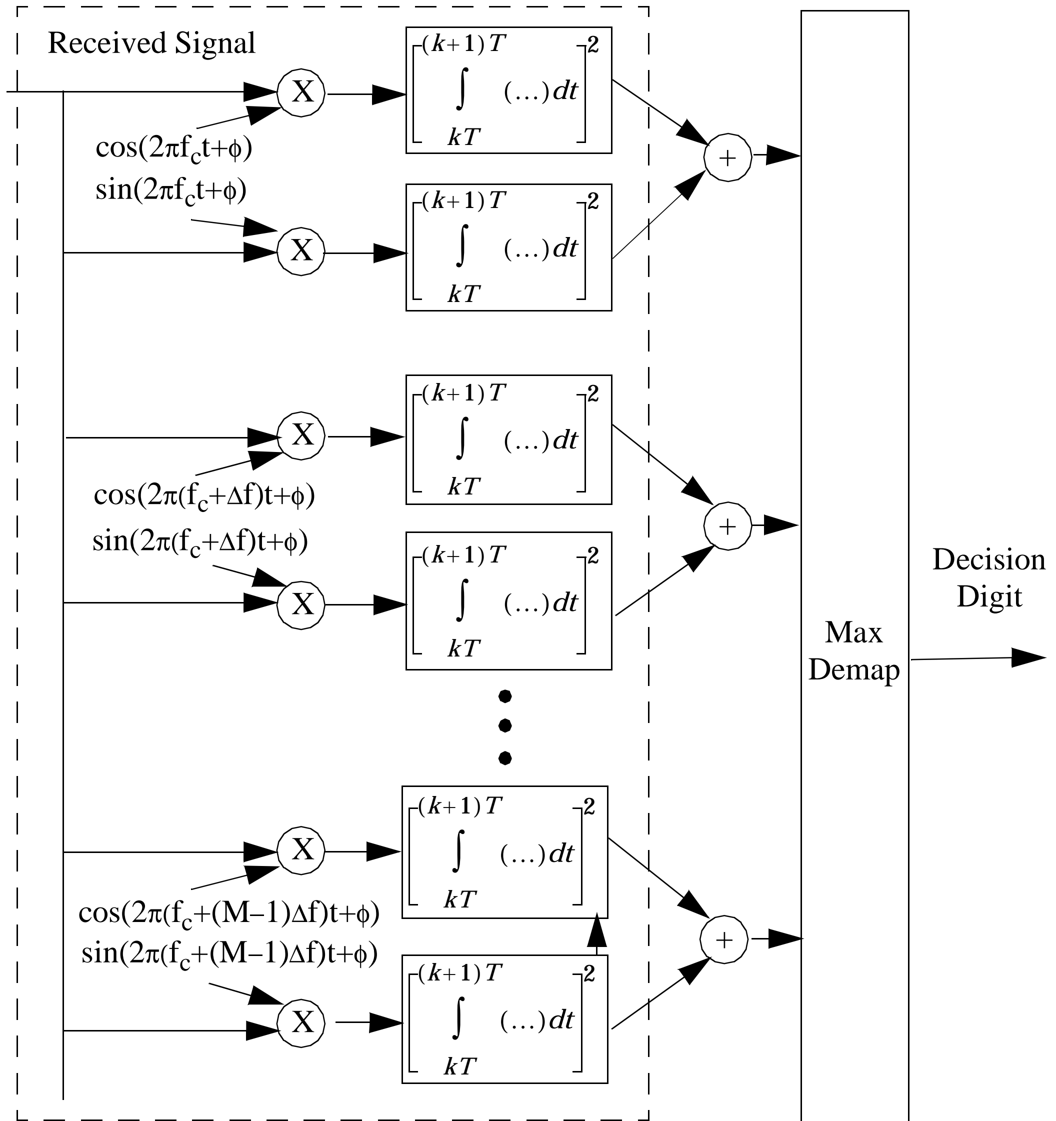
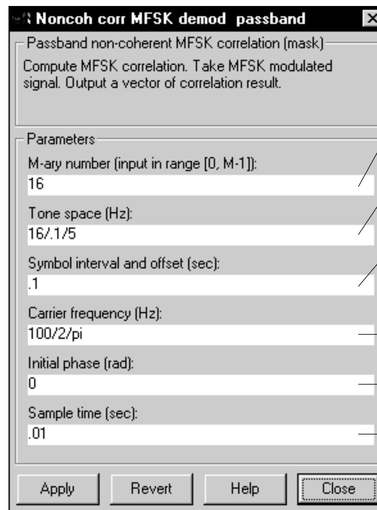


Figure 6-21: Block Diagram of Noncoherent MFSK Correlation Demodulation

The functionality of the block is inside the dashed box. Parameter f_c is the carrier frequency, Δf is the tone space, and ϕ the initial phase in the demodulation. M is the M-ary number, and T is the integration time interval.

This block accepts a scalar input and outputs a vector of correlation values.

Dialog Box



Specify the possible range of the input symbols. The range is [0, M-1].

The frequency separation between neighboring input symbols.

The sample time of the transmitted symbol. When this parameter is a two-element vector, the second element is the offset value. This parameter determines the integration time interval T of the correlation function.

The frequency of the carrier signal.

The initial phase of the carrier signal.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

FM block and MFSK Map block

Equivalent M-function

comdemod

MPSK Map

Catagory

Modulation Map

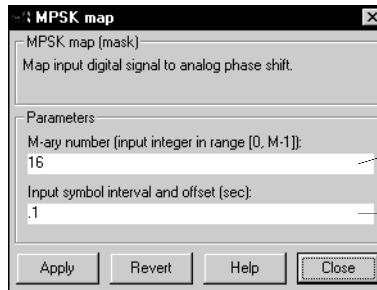
Location

Digital Mo/Dem Map/Demap Sublibrary

Description

The MPSK (multiple phase shift-keying) Map block converts an integer signal into a coded signal using the MPSK method. The discrete input signal must be in the range [0, M–1], where M is the M-ary number. The phase shift for input symbol i is $2\pi i / M$

Dialog Box



Specify the range of the input symbols. The range is [0, M-1].

The sample time of the transmitted digit symbol. When this parameter is a two-element vector, the second element is the offset value.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MPSK Correlation Demod block and Min/Max Demap block

Equivalent M-function

`modmap`

MPSK
dm corr

MPSK Correlation Demodulation

Catagory Correlation Calculation

Location Digital Mo/Dem Map/Demap Sublibrary

Description The MPSK (multiple phase shift-keying) Correlation Demodulation block calculates the correlation between an input signal and a vector of carrier frequency sinusoidal signals, all of which have the same frequency but vary in phase. The figure below shows the MPSK correlation demodulation method:

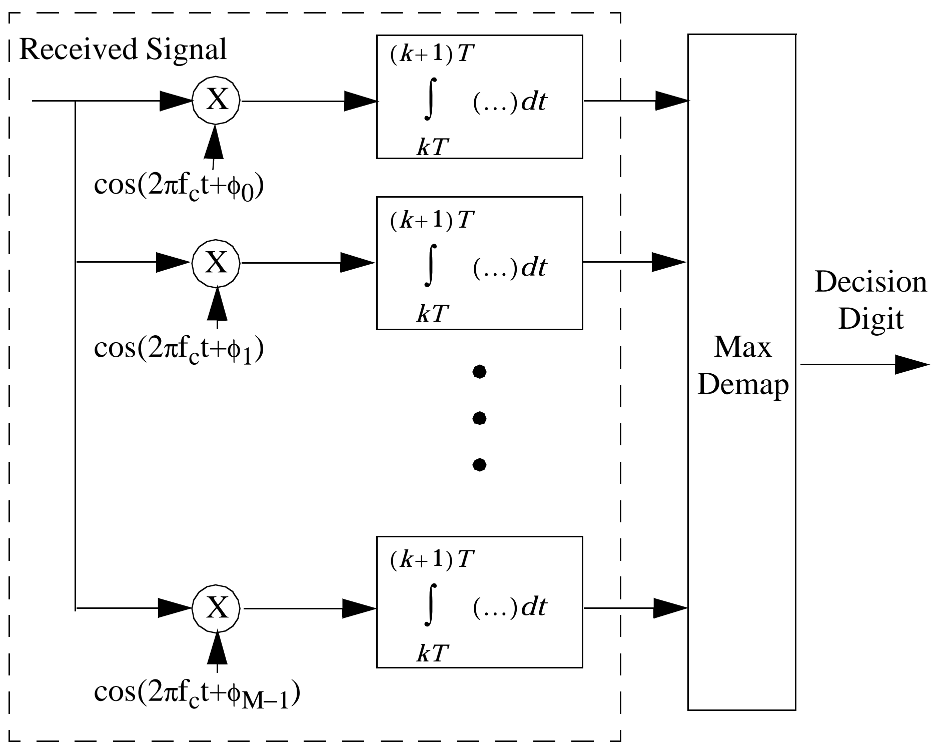
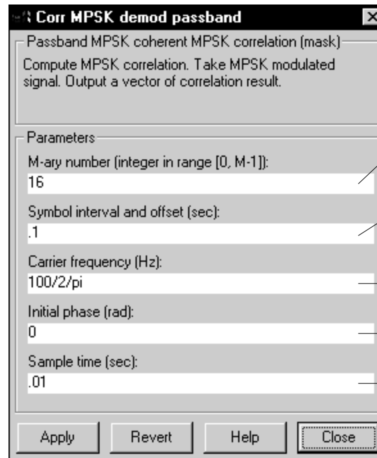


Figure 6-22: Block Diagram of MPSK Correlation Demodulation

The functionality of the block is inside the dashed box. Parameter f_c is the carrier frequency, and ϕ_i is the initial phase for digit i.

This block accepts a scalar input and outputs a vector of correlation values.

Dialog Box



Specify the possible range of the input symbols. The range is [0, M-1].

The sample time of the transmitted digit symbol. When this parameter is a two-element vector, the second element is the offset value.

The frequency of the carrier signal.

The initial phase of the carrier frequency.

The calculation sample time. By the Nyquist sampling theorem, `1/sample_time > 2*carrier_frequency`.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

MFSK Correlation Demod block and Min/Max Demap block

Equivalent M-function

`modmap`

Signal Multiple Access

Most modern communication transmitting resources are shared by multiple users. Signal multiple access is a technique to share fixed communication resources by multiple users. A typical multiple access communication system is shown in the figure below. The multiple access block on the transmitting side collects all information to be transferred and sends the information through the transmitting channel. The multiple access block on the receiving side recovers the individual signals and divides the signals into the individual receivers.

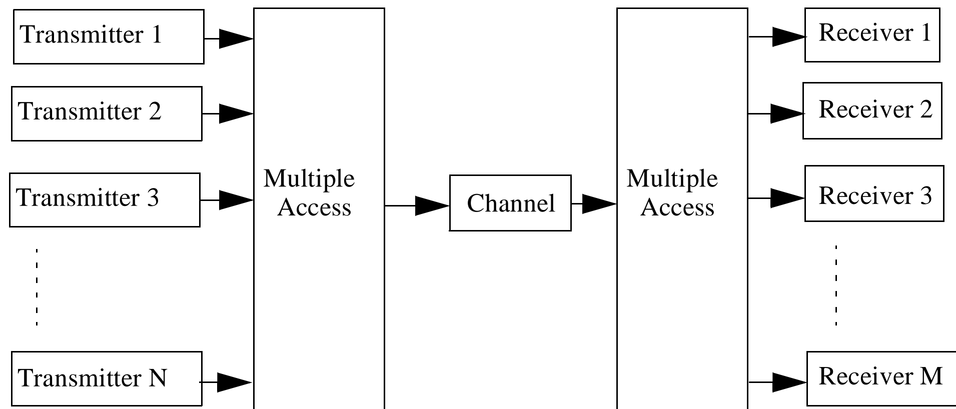


Figure 6-23: A Typical Multiple Access Communication System

The toolbox includes four different multi-access techniques:

- Time-share multiplex
- TDMA (time division multiple access)
- FDMA (frequency division multiple access)
- CDMA (code division multiple access)

The Time-sharing multiplex technique is used for analog signals. TDMA is used with discrete information sequences. The Vector Distributor block is provided as a basic utility for constructing TDMA blocks.

FDMA uses modulation techniques. As discussed in the “Modulation and Demodulation” section of this chapter, there is more than one way to modulate

a signal. The multiple access sublibrary gives an example for the FDMA technique. You can use the example to implement your own FDMA systems. The same is true for the CDMA. A frequency hopping CDMA example is provided in the multiple access sublibrary to illustrate how to use the available blocks to build a transmitting system with CDMA.

This figure shows the Multiple Access Sublibrary:

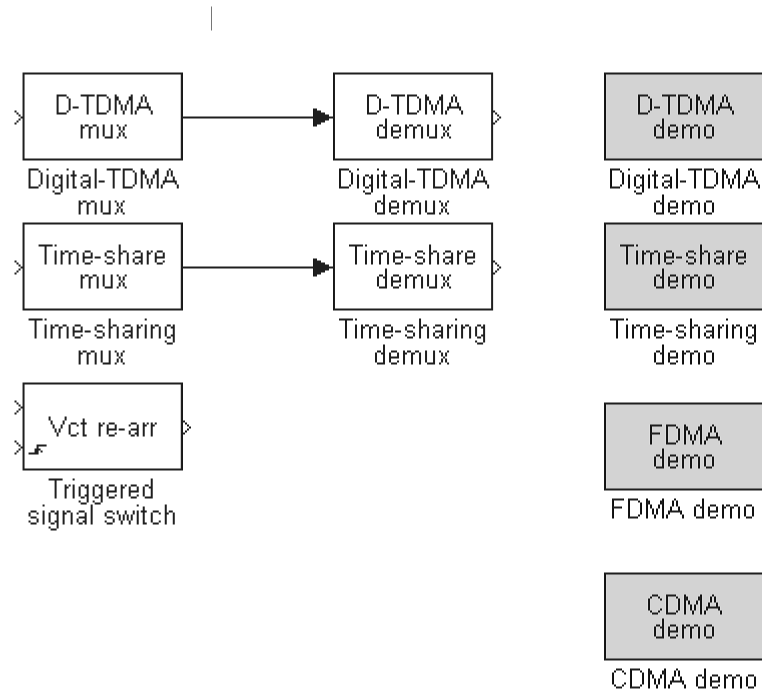


Figure 6-24: Multiple Access Sublibrary

Multiple Access Reference Table

This table lists the Simulink blocks in the Multiple Access Sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
Time-Share Demux	Time-Sharing demultiplexing recovers N analog signals (N≥1) from one analog input signal
Time-Share Mux	Time-Sharing multiplexing routes a single analog input to N (N≥1) analog output signals
D-TDMA Demux	Digital time division multiple access demultiplexing recovers N digital signals (N≥1) from one digital input signal
D-TDMA Mux	Digital time division multiple access multiplexing routes a single digital input to N (N≥1) digital output signals
Vector Redistributor	Redistributes elements from input vectors to output vectors in a specified rearrangement scheme

D-TDMA Mux

Catagory Time Division Multiple Access

Location Multiple Access Sublibrary

Description The D-TDMA (digital time division multiple access) Mux block takes N (N>1) signals and generates a signal output signal using multiplexing. It divides a single transmission time period into time-sharing sections. Each transmitted signal takes one of the time-sharing sections in the transmission. During the time in which one signal has access to the multiplexer, the other signals are denied access.

This block accepts a length N vector input (N>1) and outputs a scalar signal. Each input signal shares a sample time t_s. The D-TDMA Mux block samples each input signal in turn during the t_s time interval. The output has sampling time t_s/N.

The figure below depicts the D-TDMA Multiplex process:

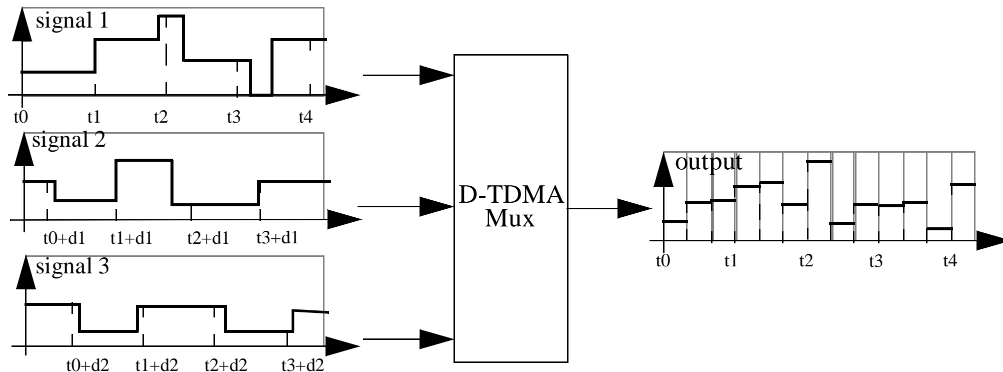
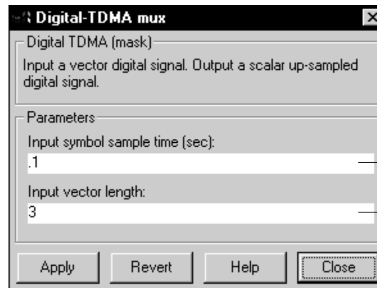


Figure 6-25: Digital TDMA Multiplexing

In this example, the D-TDMA multiplexer samples signals 1, 2, and 3 in turn and outputs one composite signal. The block samples the first element of the input vector at time t_0, the second element at time ($t_0 + t_s/3$), and the third element at time ($t_0 + 2*t_s/3$).

Dialog Box



Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

D-TDMA Demux

D-TDMA Demux

D-TDMA mux

Catagory Time Division Multiple Access

Location Multiple Access Sublibrary

Description The D-TDMA (digital time division multiple access) Demux block recovers N (N>1) digital signals from one input digital signal using demultiplexing. This block assumes that the input signal is the output of the D-TDMA Mux block.

This blocks accepts a scalar input and outputs N digital signals. Each input signal shares a sample time t_s/N. The D-TDMA Demux block samples each input signal in turn during the t_s time interval. The output has sampling time ts.

The figure below depicts the D-TDMA Demultiplex process:

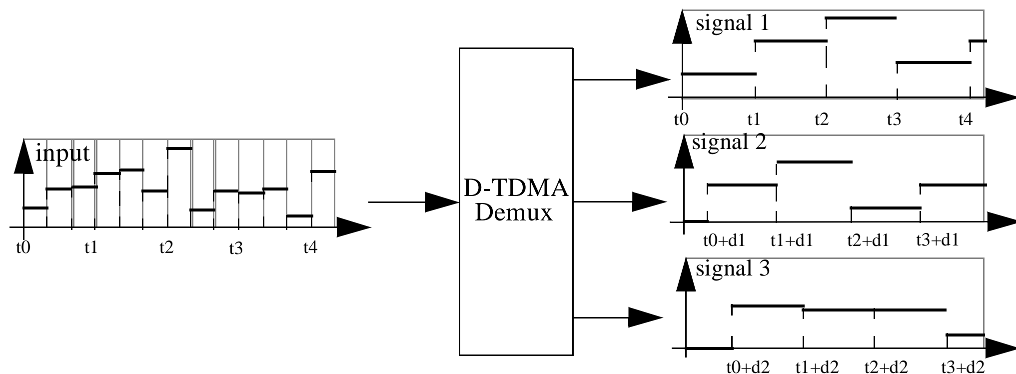
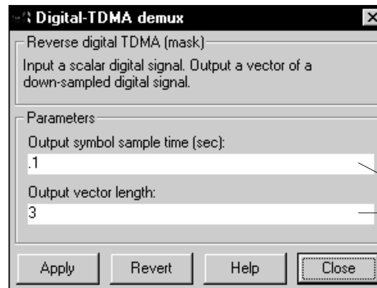


Figure 6-26: Digital TDMA Demultiplexing

Each recovered signal is a sampled output from the input signal. d1 and d2 are offset values; here d1 = t_s/3 and d2 = 2∗t_s/3. In general, di = i∗t_s/N, where i ranges from 1 to N–1.

Dialog Box



Match these parameters to the ones used in the corresponding D-TDMA Mux block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Discrete time
States	N/A
Direct Feedthrough	Yes

Pair Block

D-TDMA Mux

Time-Share Mux

Time-share mux

Catagory Time Division Multiple Access

Location Multiple Access Sublibrary

Description The A-TDMA (analog time division multiple access) Mux block takes N (N>1) analog signals and generates a single output analog signal using multiplexing. It divides a single transmission time period into time-sharing section; each transmitted signal takes one of the time-sharing sections in the transmission in turn.

This blocks accepts a length N vector input (N>1) and outputs a scalar signal. Each input signal shares a sample time t_s. The D-TDMA Mux block samples each input signal in turn during the t_s time interval. The output has sampling time t_s/N.

The figure below depicts the A-TDMA Multiplex process:

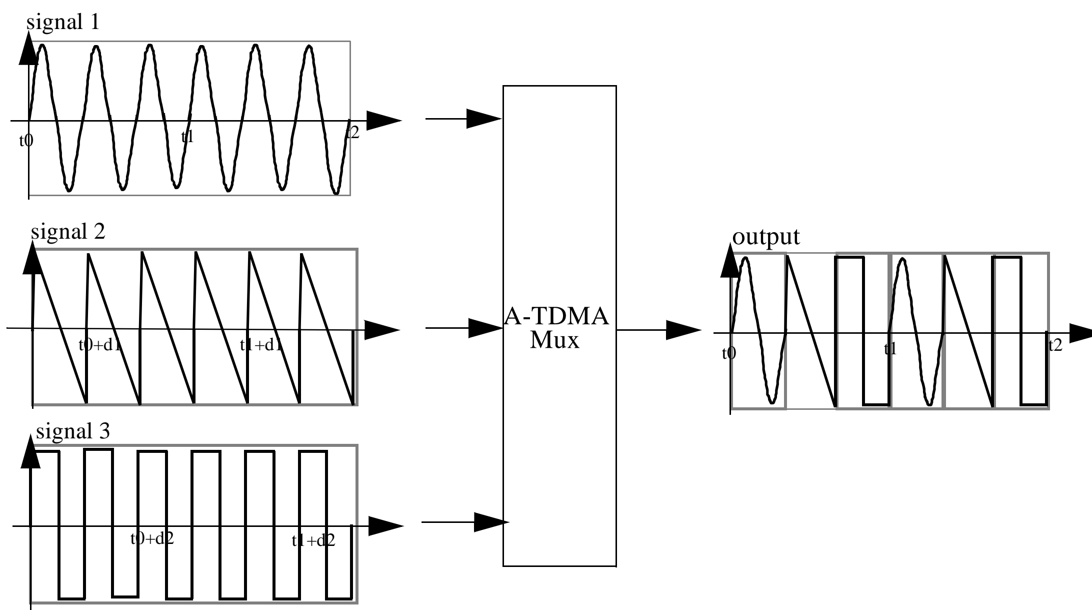
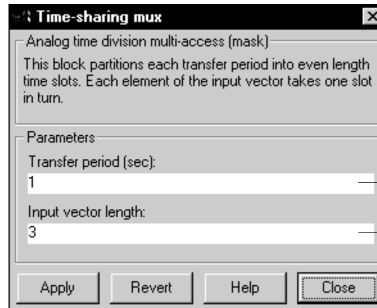


Figure 6-27: Analog TDMA Multiplexing

In this example, the A-TDMA multiplexer samples signals 1, 2, and 3 in turn and outputs one composite signal. The block samples the first element of the input vector at time t_0, the second element at time ($t_0 + t_s/3$), and the third element at time ($t_0 + 2*t_s/3$).

Dialog Box



Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Mixed
States	N/A
Direct Feedthrough	Yes

Pair Block

A-TDMA Demux

Time-Share Demux

Time-share demux

Catagory Time Division Multiple Access

Location Multiple Access Sublibrary

Description The Time-Share Demux block recovers N (N>1) digital signals from one input digital signal using demultiplexing. This block assumes that the input signal is the output of the D-TDMA Mux block.

This blocks accepts a scalar input and outputs N digital signals. Each input signal shares a sample time t_s. The D-TDMA Mux block samples each input signal in turn during the t_s time interval. The output has sampling time ts/N.

The figure below depicts the D-TDMA Mux process:

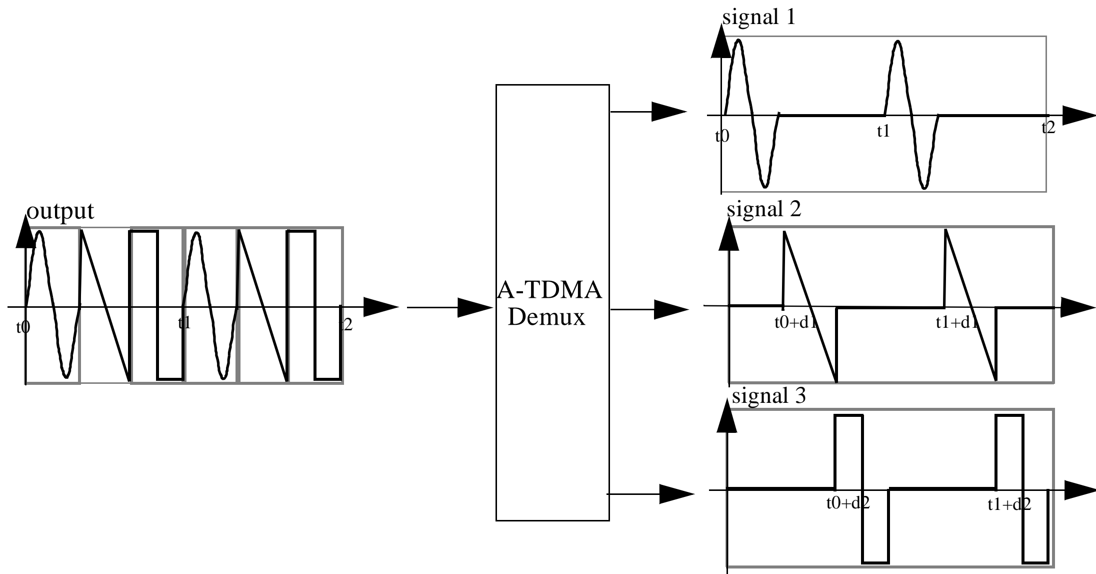
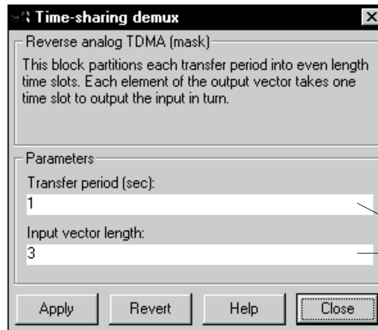


Figure 6-28: Analog TDMA Demultiplexing

Each recovered signal is a sampled output from the input signal. d1 and d2 are offset values; here d1 = $t_s/3$ and d2 = $2*t_s/3$. In general, di = $i*t_s/N$, where i ranges from 1 to N–1.

Dialog Box



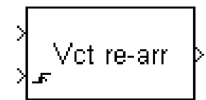
Match these parameters to those used in the corresponding Time-Share Mux block.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Mixed
States	N/A
Direct Feedthrough	Yes

Pair Block

A-TDMA Mux



Vector Redistributor

Catagory Utility

Location Multiple Access Sublibrary

Description The Vector Redistibutor block rearranges N input signals into M output signals. Both N and M are positive integers. This block is a generalization of the Time-Share Mux and Time-Share Demux blocks. The input and output signals can be any combination of analog and digital signals.

The figure below is an example of a vector redistribution of three input signals to two output signals:

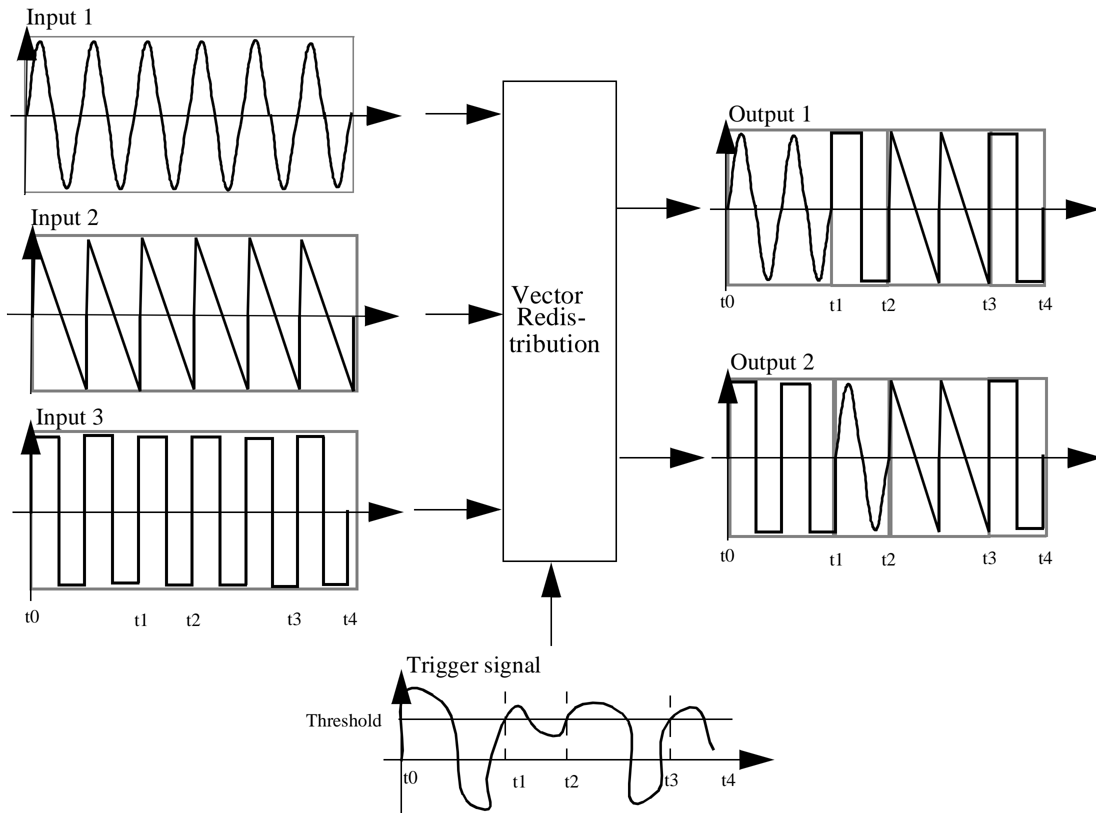


Figure 6-29: Vector Redistribution of Three Input Signals to Two Output signals

Input port 1 accepts a length N vector, where N is the number of input signals that the block redistributes. Input port 2 accepts a scalar trigger signal; the raising edge of the trigger signals triggers the block to rearrange the order of the input signal. In the example above, the triggers occur at t1, t2, and t3. The output of this block is a length M signal, where M>1. N and M do not have to be equal.

The rearrangement pattern is defined in the **Switch box** parameter, which is a matrix with M columns. The number of rows is arbitrary. When the first trigger occurs, the Vector Redistributor block uses the first column of the Switch box to rearrange the input vector signal. When the second trigger occurs, the block uses the second row of the **Switch box** to rearrange the input vector. This process continues until the block exhausts all the rows available. At that point the process starts again with the first row.

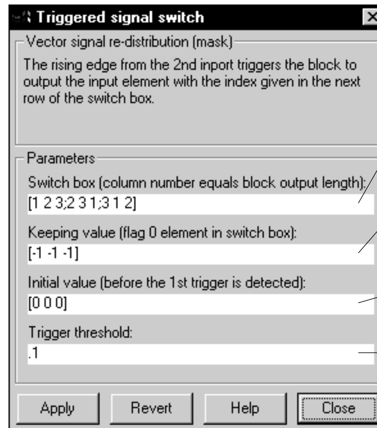
For the example above, the switch box matrix is:

```
switch_box = [1 3; 3 1; 2 2; 3 3];
```

Assume that the first trigger has occurred. Then the values in the first row define the rearrangement. The rule is that the `i` th column input value goes to the output specified in the `i` th column of the first row. In this case, it means the input vector element 1 goes to output 1, and input vector element 3 goes to output 2. In the second row, the rearrangement rule specifies that input vector element 3 goes to output 1, and input vector element 1 goes to output 2.

If a 0 is entered in the ith element of the rearrangement row, the block inspects the **Keeping value** parameter at that index to determine whether the ith output is held at its last value or reset to its initial value. For example, if instead of [1 3] in the first row we had placed [0 3], the Vector Redistributor block would check the first element of the **Keeping value** vector. If that element is greater than or equal to 0, output 1 is held at the last value it had during the previous trigger. If that element is a negative number, the Vector Redistributor block sets output 1 to the initial value as specified in the **Initial Value** vector.

Dialog Box



Specify the rearrangement pattern. All elements of the **Switch box** matrix must be in the range [0, N-1].

A length N vector that specifies whether the output defaults to the last held value or resets to its initial value. The test only occurs if the **Switch box** has a 0 entry in the rearrangement pattern.

A length N vector that specifies the output value prior to the first trigger occurence.

Specify the value that, once crossed from below, forces a trigger.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Input	Yes
Vectorized Input 2	No
Vectorized Output	Yes
Time Base	Auto
States	N/A
Direct Feedthrough	Yes

Transmitting and Receiving Filters

Before a signal is sent out to a transmission channel, or after a signal is received from a transmission channel, filters are usually involved. For example, bandpass filters may be needed for a frequency sharing channel; lowpass filters may be needed for a white noise channel; a matched filter may be needed for a fading channel.

The Transmitting/Receiving Filter Sublibrary provides lowpass filters, highpass filters, bandpass filters and bandstop filters. Both analog and digital filters are included. The filter types can be Butterworth filters, Chebychev type I or type II filters, or elliptic filters. More filters can be found in the Filters sublibrary in the DSP Blockset. The DSP Blockset is recommended but not required to use this toolbox.

This figure shows the Transmitting/Receiving Filter Sublibrary:

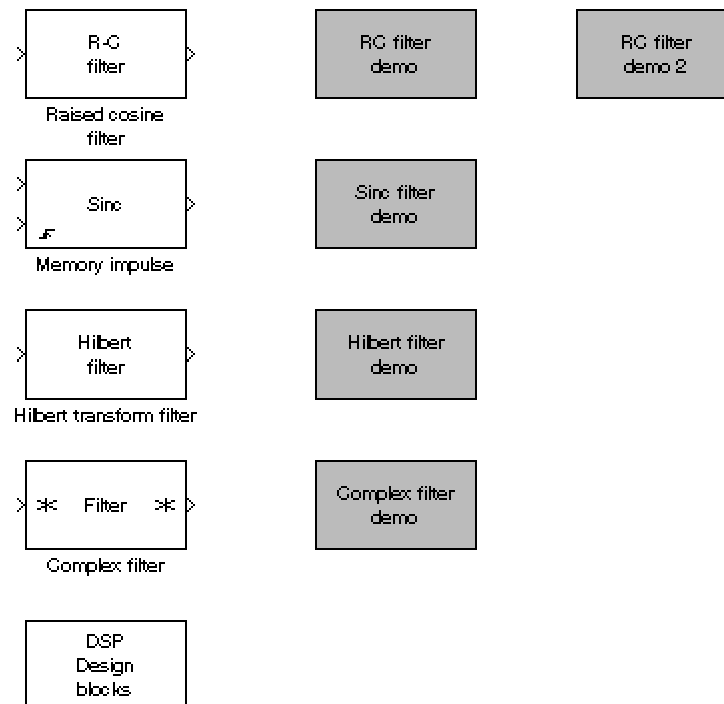
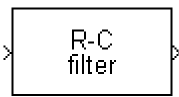


Figure 6-30: Transmitting/Receiving Filter Sublibrary

Transmitting/Receiving Filters Reference Table

This table lists the Simulink blocks in the Transmitting/Receiving Filters Sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
Complex Filter	Implements IIR and FIR filter with complex parameters
Hilbert Filter	Implements a Hilbert filter
Raised Cosine Filter	Implements a raised cosine filter
Sinc	Outputs the zero-crossing value of an input signal when triggered



Catagory Filtering

Location Transmitting and Receiving Filters Sublibrary

Description The Raised Cosine Filter block implements a raised cosine filter. In digital transmission, digital signals are represented by step waves. A step wave has infinite bandwidth. In narrow band transmission, the bandwidth is a critical issue. The spectrum of a step wave spreads over a wide range of frequency domains. To restrict the spectrum of the source signal, an R-C (raised cosine) filter is commonly used. The transfer function *H(f)* of an R-C filter is

$$H(f) = \begin{cases} T & \text{if } 0 \le |f| \le \frac{1-r}{2T} \\ \frac{T}{2}\left[1 + \cos\left(\frac{\pi T}{r}\left(|f| - \frac{1-r}{2T}\right)\right)\right] & \text{if } \frac{1-r}{2T} \le |f| \le \frac{1+r}{2T} \\ 0 & \text{if } \frac{1+r}{2T} \le |f| \end{cases}$$

where r is the rolloff factor and T is the time interval of the digital message. After a digital signal passes an R-C filter, the frequency of the signal is limited in the band [-W, W], where $W > \frac{1}{2T}$. The time response *h(t)* of the filter equals:

$$h(t) = \frac{\operatorname{sinc}\left(\frac{t}{T}\right)\cos\left(\frac{\pi r t}{T}\right)}{1 - 4\frac{r^2 t^2}{T^2}}$$

The time impulse response and the frequency response of the R-C filter for r=0, 0.5, 1 are shown in the figure below. Note that the time domain impulse response of the filter shown in the upper plot of the figure displays only three taps. In theory, the taps extends to +/– infinity.

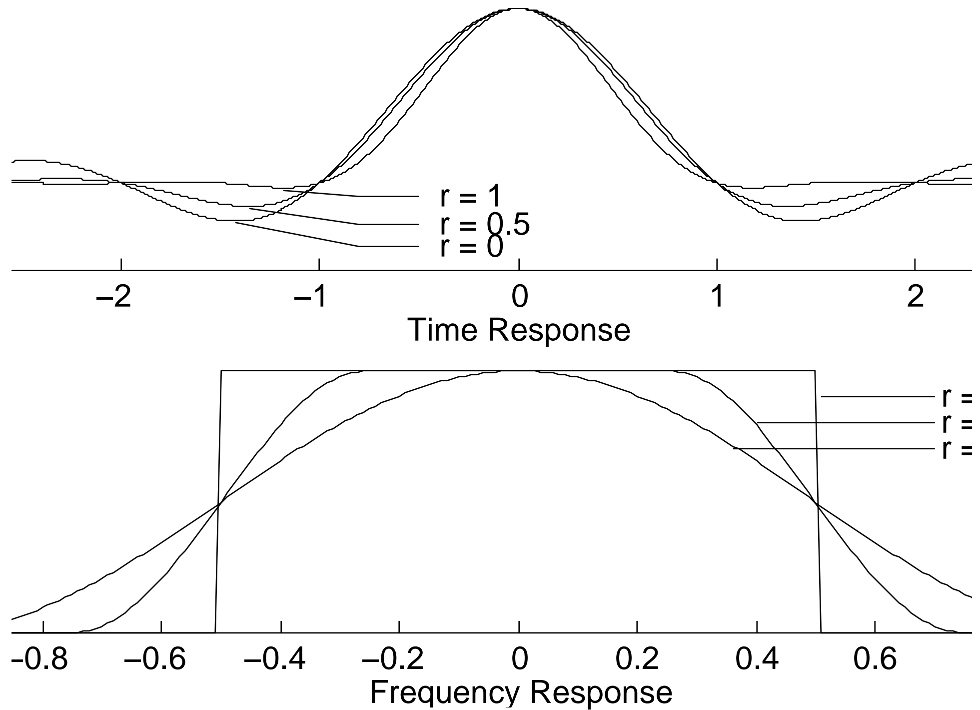


Figure 6-31: Time and Frequency Responses of the R-C Filter

In some applications, the R-C filter is divided into two parts, one for the transmitting side and one for the receiving side. In this case, each side is the square root of the R-C filter.The impulse response of a square-root R-C filter is:

$$h(t) = \frac{4r}{\pi\sqrt{T}} \frac{\cos\left((1+r)\pi\frac{t}{T}\right) + \frac{\sin\left((1-r)\pi\frac{t}{T}\right)}{4r\frac{t}{T}}}{\left(1-\left(4r\frac{t}{T}\right)^2\right)}$$

A Raised Cosine filter is a *noncausal* filter. In practical design, the implementation of a noncausal filter is impossible because a noncausal filter depends on future information. To solve this problem, this toolbox has artificially added a time delay to the filter in the implementation. The time delay is usually T multiplied by a positive integer.

By default, an R-C filter takes a digital pulse signal with the nonzero pulse with its one sample step pulse width. When the input signal has its nonzero pulse longer than one R-C filter sample step, a 1/sinc filter should be added to truncate the pulse width. In this block, the parameter **With or without sinc filter** specifies whether a 1/sinc filter should be applied to the input signal. In general, the application of a 1/sinc filter is suggested.

There is one case, however, when 1/sinc should not be used. When the R-C filter is broken into two square-root R-C filters, do not apply a 1/sinc filter to the second R-C.

Example

The figure below is a example of using the R-C filters to process an input signal. The three signals outputted to the scope are the original digital signal, one R-C filter filtered signal, and two cascaded square-root R-C filters filtered signal.

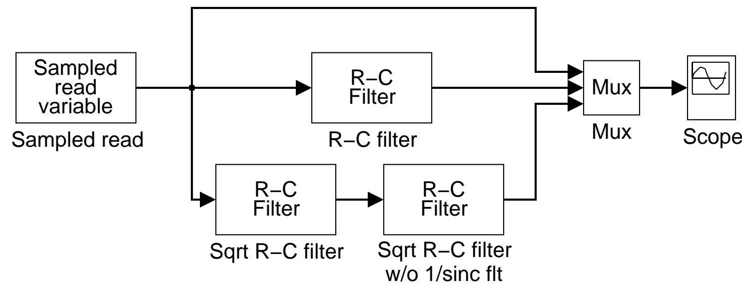


Figure 6-32: Time Response and Frequency Response of the R-C Filter

The digital signal source repeatedly outputs the sequence [-1 0 1 2 3 1] at digital duration time 0.1 second. All R-C filters use computation sample time 0.01, rolloff factor 0.5, and an FIR filter. The R-C filter in the middle uses normal R-C filter with delay step as 6. The two bottom R-C filters use square-root R-C filter with delay step as 3. The left side square-root R-C filter uses 1/sinc filter; the right side R-C filter does not. Because the two square-root R-C filters are cascaded, the delay for the two filters together is six delay steps.

The figure below is a snapshot of the scope. The dashed line shows the digital input signal, which is the stair step curve. The dotted line is the output of a single normal R-C filter. The solid curve is the output of the two square-root R-C filters. Because the two filtered signals are very close in value, the dotted and dashed lines overlay each other and can barely be distinguished. Both signals have six steps of delay from the original signal.

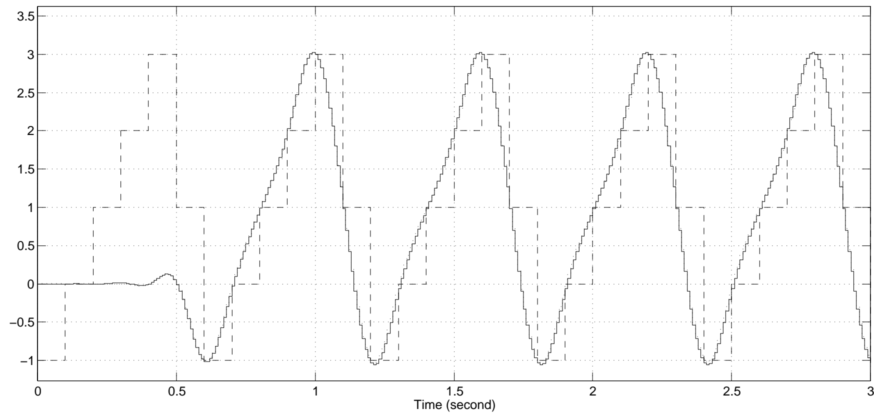
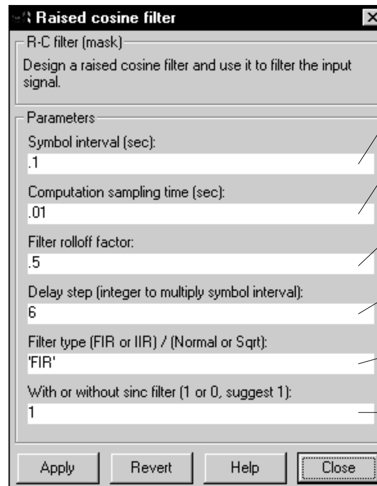


Figure 6-33: Original vs. RC-Filtered Signals

Dialog Box



The duration period for each transferred digit.

The calculation sampling time. This value must be less that the duration of digit. Typically, the ratio of these two parameters is an integer.

The rolloff factor r. Set this parameter to a value between 0 and 1 inclusive.

Set this to a positive integer. The actual delay time is delay_step*symbol_interval.

A string variable input. Specify either `'FIR'` or `'IIR'` and `'Normal'` or `'Sqrt'`.

Set to 1 if you require a 1/sinc filter and 0 otherwise.

Characteristics

No. of Inputs	1
No. of Output	1
Vectorized Input	No
Vectorized Outputs	No
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

Catagory Filter

Location Transmitting and Receiving Filters Sublibrary

Description The Sinc block outputs the zero-crossing value during one sample interval when triggered by the raising edge of the trigger signal. The transfer function of this block is a sinc function

$$\mathrm{sinc}(f) = \begin{cases} \dfrac{\sin(\pi f)}{\pi f} & t \neq 0 \\ 1 & t = 0 \end{cases}$$

The time domain impulse response of this block is an unit width square pulse. The pulse width unit is specified in the **Holding time** (t_s) parameter.

This block has two input ports and one input port. Input port 1 accepts the signal. Input port 2 takes the trigger signal. The figure below shows an example of this block:

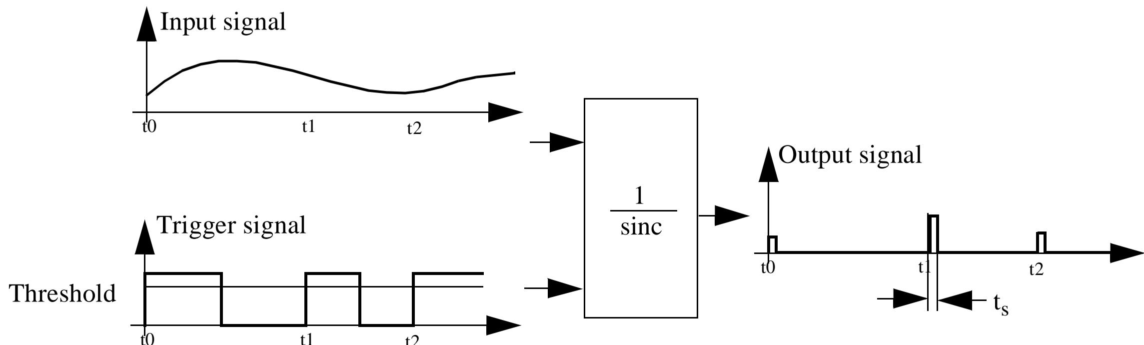
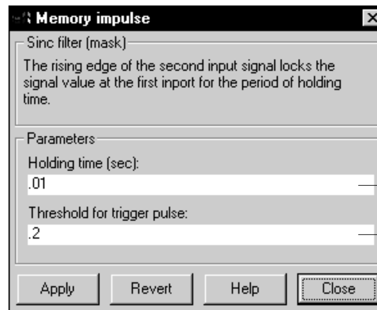


Figure 6-34: The Functionality of the Sinc Block

This block is used in constructing the Raised Cosine Filter block.

Dialog Box



The amount of time for the output pulse period.

The threshold for detecting the raising edge of the trigger signal. When this entry is zero, the block outputs the input signal of the first input signal and the trigger signal is ignored.

Characteristics

No. of Inputs	1
No. of Output	1
Vectorized Input	No
Vectorized Outputs	No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Category Filter

Location Transmitting and Receiving Filters Sublibrary

Description The Hilbert Filter block implements a Hilbert transfer filter. The Hilbert transform filter has the transfer function

$$H(f) = -j\operatorname{sgn}(f)$$

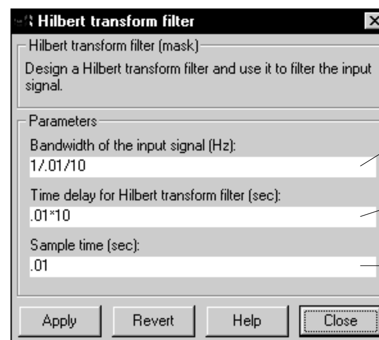
which has the impulse response

$$h(t) = \frac{1}{\pi t}$$

As discussed in the Tutorial Chapter, a Hilbert transfer filter is a noncausal filter. To implement a noncausal filter, a time delay is added artificially. To use this block, you must specify the bandwidth of the input signal, the time delay (DLY), and computation sample time (TS). We encourage you to use the MATLAB function *hilbiir* to test the parameters to be used in the block. For an accurate calculation, the parameters should be chosen such that DLY is at least a few times larger than ST, and rem(DLY, ST) = ST/2.

Please be aware that when you input a signal bandwidth exceeded the bandwidth parameter, this block may output unexpected result.

Dialog Box



The upper limit of the bandwidth of the input signal.

The time delay of the Hilbert transform filter (in second).

The sample time of the Hilbert filter.

Equivalent M-function hilbert in Signal Processing Toolbox

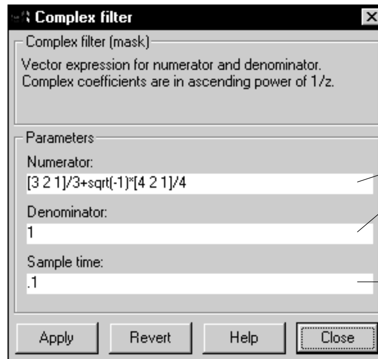
Complex Filter

Catagory Filter

Location Utility Sublibrary

Description The Complex Filter block implements both infinite impulse response (IIR) and finite impulse response (FIR) filters. This block accepts a real or a complex input signal and outputs a complex signal. The numerator and denominator are complex vectors that represent the coefficients of the filter's transfer function. The coefficients are in ascending power of 1/z.

Dialog Box



Specify the numerator and denominator coefficients of the complex filter. The coefficients must be in ascending powers of 1/z. When the denominator is set to 1, the filter is an FIR filter.

Specify the calculation sample time in seconds

Characteristics

No. of Inputs/Output	1/1
Vectorized Inputs	Real/Complex
Vectorized Outputs	Complex
Time Base	Discrete
States	2*(length(numerator)+ length(denominator)-2)
Direct feedthrough	Yes, if the 1st element in numerator is non-zero.

Channels

A channel is a communication media that transfers a signal from the transmitting side to the receiving side. A channel may reduce the signal quality in the receiving side because of transmitting noise, transmitting interference, and transmitting loss. There are a large number of different transmitting channels in industrial applications. This sublibrary provides models for both passband channels and baseband channels. The figure below shows the Channel Sublibrary:

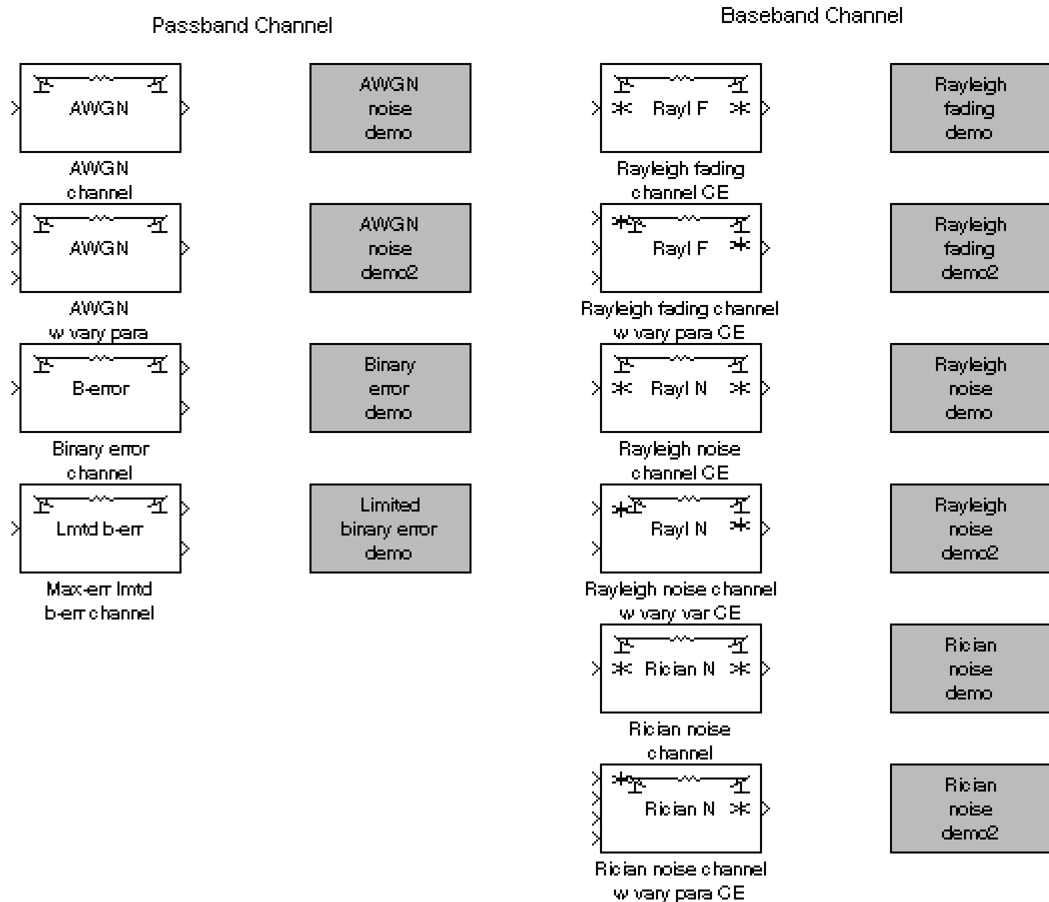


Figure 6-35: Passband Channel Sublibrary

The passband channel provides four blocks:

- AWGN Channel
- Varying AWGN Channel
- Binary-error channel
- Limited binary-error channel

Channel Reference Table

This table lists the Simulink blocks in the Channel Sublibrary.
(This table lists the blocks in alphabetical order for your convenience.):l

Block Name	Description
AWGN Channel	Additive White Gaussian noise, the most common passband channel model
Binary Error Channel	Binary noise with binary signal input and binary signal outputs
Limited Binary Error Channel	Binary noise with limited error occurance probability
Rayleigh Fading CE Channel	Rayleigh noise, modeled in baseband, with signal attenuation
Rayleigh Noise CE Channel	Rayleigh noise modeled in baseband
Rician Noise CE Channel	Rician noise modeled in baseband
Varying AWGN Channel	Additive White Gaussian noise, modeled in passband, with varying statistical parameters
Varying Rayleigh Fading CE Channel	Rayleigh noise, modeled in baseband, with varying statistical parameters and signal attenuation
Varying Rayleigh Noise CE Channel	Rayleigh noise, modeled in baseband, with varying statistical parameters
Varying Rician Noise CE Channel	Rician noise, modeled in baseband, with varying statistical parameters

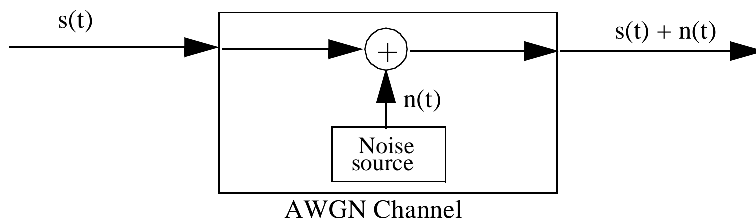
AWGN Channel

Catagory Passband Channel

Location Channel Sublibrary

Description

The AWGN (additive white Gaussian noise) Channel block adds white Gaussian noise to the signal transmitting through this channel. The AWGN channel model is commonly used in communications. The model for an AWGN channel is:



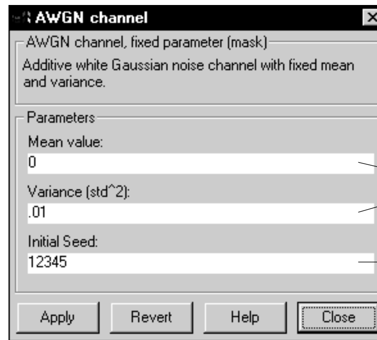
AWGN Channel

This block uses the Gaussian White Noise Generator block as the noise source. The vector length for the entry in the **Initial Seed** parameter must match the input vector size of this block. The output vector size equals the input vector size.

You must specify the mean(s) and covariances of the of the noise. The mean can be either a vector of length equal to the seed or a scalar, in which case all the elements of the noise vector share the same mean value. The covariance matrix can be one of three things:

- An N-by-N positive semi-definite matrix, where N is the length of the seed. The off-diagonal elements of the matrix are the correlations.
- A length N vector, in which case the individual elements of the noise vector are uncorrelated but have unequal variances.
- A scalar, in which case all the elements of the noise vector are uncorrelated but share the same variance.

Dialog Box

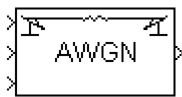


Specify the mean and variance of the noise output.

Initialize the seed. The length of the seed sets the length of the random output vector.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Varying AWGN Channel

Catagory Passband Channel

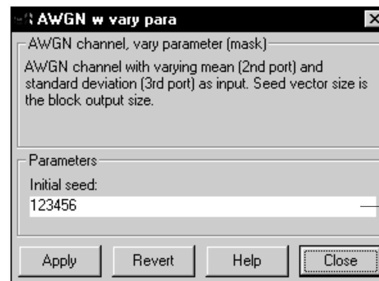
Location Channel Sublibrary

Description

The AWGN (additive white Gaussian noise) Channel block adds white Gaussian noise to the signal transmitting through this channel. The AWGN channel model is commonly used in communication.

This block has three input ports and one output port. The first input port inputs the transmitting signal. The second input port inputs the mean value of the Gaussian noise. The third input port inputs the variance. The second and the third input port can accept a scalar or a vector signal. In the vector case, the vector length must equal the length of the first input port signal. The vector length for the entry in the **Initial Seed** parameter must match the input vector size of this block. The output vector size equals the input vector size.

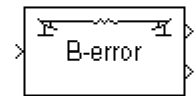
Dialog Box



Initialize the seed.

Characteristics

No. of Inputs	3/1
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Binary Error Channel

Category Passband Channel

Location Channel Sublibrary

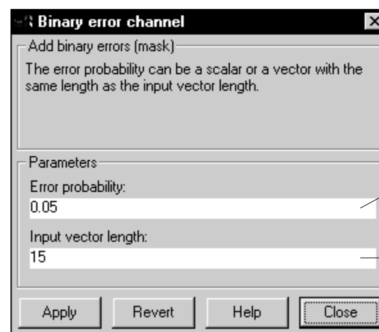
Description The Binary Error Channel block generates adds binary errors to the signal transmitted through this channel. This block is useful for the binary signal transmission.

This block has one input port and two output ports. The input port takes the transmitting binary signal. The first output port outputs the binary signal passed the channel. The second output port outputs the number of errors added to the transmitting signal.

To use this block, you must specify the error occurring probability and the input signal vector size.

The parameter specified in the first parameter entree is the binary error occurring probability. If you need to specify the error pattern, you should use the Limited Binary Error Channel.

Dialog Box

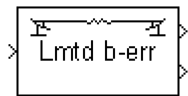


The binary error occurring probability. The value of this parameter must be no smaller than zero and no larger than one.

Input signal vector length, which is the same vector length of the signal to the first output port.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Limited Binary Error Channel

Category Passband Channel

Location Channel Sublibrary

Description

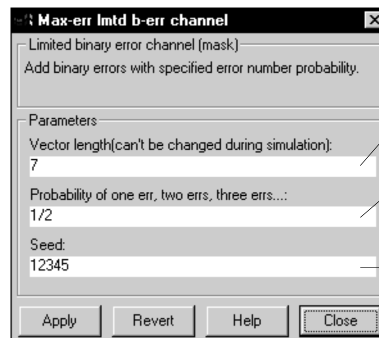
The Limited Binary Error Channel block generates adds binary errors to the vector signal transmitted through this channel. This block is useful for binary signal transmission.

This block has one input port and two output ports. The input port takes the transmitting binary signal. The first output port outputs the binary signal passed the channel. The second output port outputs the number of errors added to the transmitting signal at the current computation.

To use this block, you must specify the input signal vector length, the error occurring pattern, and the random number generator seed.

You can specify the error occuring pattern for this block as a vector, [*p1*, *p2*, *p3*,...*pM*], where *p1* is the probability of a single error in the channel for the current computing output vector, *p2* is the probability of two errors, *p3* is the probability of three errors, and so on. The vector size M must be less than or equal to the block output length. Please note that $\sum_{i=1}^{m} p_i \leq 1$. The probability of all zero vectors is $1 - \sum_{i=1}^{m} p_i$. The minimum vector size for this probability vector is 1 and the maximum vector size for this vector is the input signal vector length.

Dialog Box



Input signal vector length, which is also the vector length of the first output port.

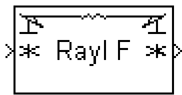
Specify the error occur pattern. The sum of this vector must be no larger than 1. The vector length of this entry should be no larger than the input vector length.

Initialize the seed.

Limited Binary Error Channel

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Rayleigh Fading CE Channel

Catagory Baseband Channel

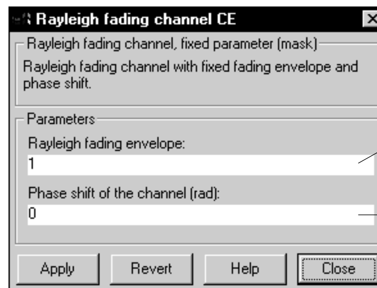
Location Channel Sublibrary

Description The Rayleigh Fading CE Channel block alters the amplitude and phase of the baseband signal transmitting through this channel. The input and the output of the block are scalar complex signals. The Rayleigh fading channel assumes all frequency components of the transmitted signal undergo the same attenuation and phase shift. The relation between the input signal *u(t)* and the output signal *y(t)* of this block has the relationship

$$y(t) = \alpha e^{j\phi} u(t)$$

where α is the fading envelope and ϕ is the phase shift of the channel. This block requires you to assign both fading envelope and phase shift in the parameters entries.

Dialog Box



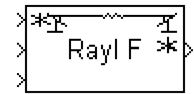
Specify the fading envelope, a scalar. This parameter is α in the Rayleigh fading channel equation.

Specify the phase shift, a scalar. This parameter is ϕ in the Rayleigh fading channel equation.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/Complex
Time Base	Auto
States	N/A

Varying Rayleigh Fading CE Channel



Catagory Baseband Channel

Location Channel Sublibrary

Description

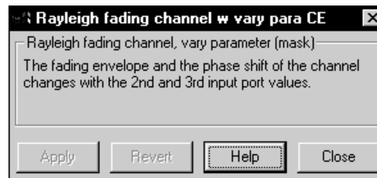
The Rayleigh Fading CE Channel block changes the amplitude and phase of the baseband signal transmitting through this channel. The Rayleigh fading channel assumes all frequency components of the transmitted signal undergo the same attenuation and phase shift. The relation between the input signal $u(t)$ and the output signal $y(t)$ of this block has the relationship

$$y(t) = \alpha(t) e^{j\phi(t)} u(t)$$

where $\alpha(t)$ is the fading envelope and $\phi(t)$ is the channel phase shift. These two parameters are scalar factors inputted from second and third input ports respectively.

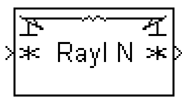
The first port's input and the output are scalar complex signals. The second and third inputs are real numbers. There are no parameters to set in this block.

Dialog Box



Characteristics

No. of Inputs/Outputs	3/1
Vectorized Input 1	Complex
Vectorized Inputs 2 and 3	No, No
Vectorized Outputs	Complex
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Rayleigh Noise CE Channel

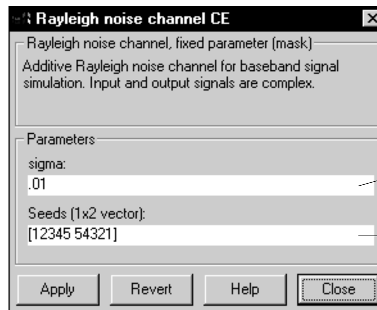
Catagory Baseband Channel

Location Channel Sublibrary

Description The Rayleigh Noise CE Channel block adds two uncorrelated Gaussian noises to the real and imaginary parts of the baseband signal transmitting through this channel. The input and output are scalar complex signals. The Rayleigh noise channel is widely used in baseband channel simulations.

This block uses the Gaussian Random Noise Generator block. The mean value is zero. The covariance matrix is $diag(\sigma^2, \sigma^2)$, where σ^2 is the entry to the parameter **sigma**.

Dialog Box



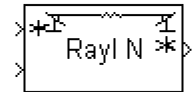
Specify the variance σ^2 for the noise. This parameter is a scalar.

Initialize the seed, a length two vector.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/Complex
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Varying Rayleigh Noise CE Channel



Catagory

Baseband Channel

Location

Channel Sublibrary

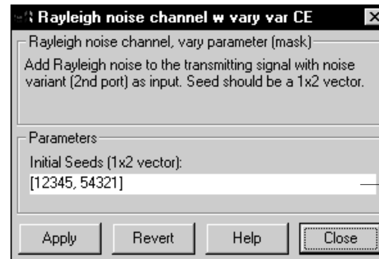
Description

The Varying Rayleigh Noise CE Channel block adds two uncorrelated Gaussian noises to the real and imaginary parts of the baseband signal transmitting through this channel. The Rayleigh noise channel is a widely used additive noise in baseband channel simulations.

This block uses the Gaussian Random Noise Generator block as the noise source. The mean value is zero. The covariance matrix is $diag(\sigma^2(t), \sigma^2(t))$, where $\sigma^2(t)$ is the value inputted to the second input port of this block.

The first input port inputs the complex transmitting signal. The second input port inputs the variance of the Gaussian noise. The input value must be a non-negative scalar value.

Dialog Box



Initialize the seed, a length two vector.

Characteristics

No. of Inputs/Outputs	2/1
Vectorized Input 1	Complex
Vectorized Input 2	No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Rician Noise CE Channel

Catagory Baseband Channel

Location Channel Sublibrary

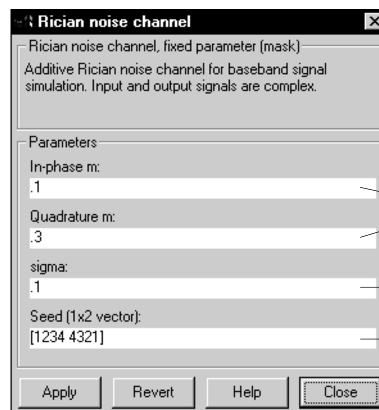
Description

The Rician Noise CE Channel block adds two uncorrelated nonzero mean Gaussian noises to the real and imaginary parts of the baseband signal transmitting through this channel. The input and the output are scalar complex signals. The Rician noise channel model is a generalized additive noise in baseband channel simulations.

This block uses the Gaussian Random Noise Generator block as the noise source. The mean values of the Gaussians used to generate the Rician noise are [*inphase_m, quadrature_m*]. The covariance matrix is *diag*(σ^2, σ^2), where the variance σ^2 is the entry to the parameter **sigma**. The block adds the two uncorrelated Gaussian noise signals to the real and imaginary parts of the complex signal transmitting through this channel.

Rician noise is a noncentered χ^2 distribution noise. A Rayleigh noise channel is a zero mean Rician noise channel.

Dialog Box



Specify the in_phase and quadrature means of the Gaussian noise.

Specify the variance of the Gaussian noise.

Initialize the seed, a length two vector.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Complex/Complex
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Varying Rician Noise CE Channel

Catagory Baseband Channel

Location Channel Sublibrary

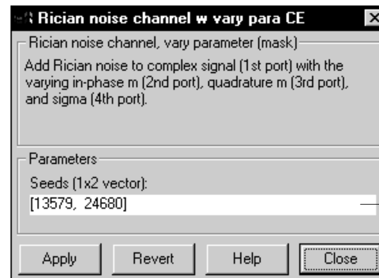
Description

The Varying Rician Noise CE Channel block adds two uncorrelated nonzero mean Gaussian noises to the real and imaginary parts of the baseband signal transmitting through this channel. The Rician noise channel model is a generalized additive noise in baseband channel simulations. This block uses a length two Gaussian noise vector as the noise source

This block has three input ports. The first port inputs the complex transmitting signal. The second input port inputs the mean value *m*, where *m* is a scalar. The mean value vector is [*m, m*]. The third input port inputs the covariance *c*, where *c* is also a scalar. The covariance matrix is *diag(c, c)*. The two uncorrelated Gaussian noise signals are added to the real and imaginary components of the complex signal transmitting through this channel.

Rician noise is a noncentered χ^2 distribution noise. A Rayleigh noise channel is a zero mean Rician noise channel.

Dialog Box



Initialize the seed, a two-element vector.

Characteristics

No. of Inputs/Outputs	3/1
Vectorized Input 1	Complex
Vectorized Inputs 2 and 3	No, No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Synchronization

A receiver must be synchronized with the transmitter in order to receive a transmitted signal correctly. Synchronization techniques play very important roles in communication systems.

The synchronization techniques in communication systems includes phase, frequency, symbol, and frame synchronization. Phase synchronization techniques are commonly called phase lock techniques. This toolbox implements analog phase locked-loop (PLL) and the classical digital PLL as examples of using Simulink blocks to build synchronization functionalities.

There are a number of PLL implementation methods. The most commonly used PLL methods are:

- Simple PLL
- Baseband model PLL
- Linear baseband model PLL
- Charge pump PLL

This figure shows the Synchronization Sublibrary:

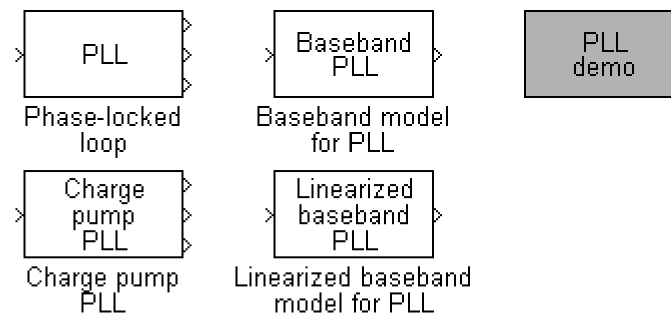


Figure 6-36: Synchronization Sublibrary

Synchronization Reference Table

This table lists the Simulink blocks in the Synchronization sublibrary. (This table lists the blocks in alphabetical order for your convenience.):

Block Name	Description
Baseband Model PLL	Implements a baseband model of the phase-locked loop
Charge Pump PLL	Implements a charge pump phase locked loop, which uses a sequential logic phase detector
Linearized Baseband Model PLL	Implements a linearized model of the phase locked loop
PLL	Implements a phase locked loop, which recovers the phase of the input signal

PLL

Catagory Phase Recovery

Location Synchronization Sublibrary

Description

The PLL (phase-locked loop) block automatically adjusts the phase of a locally generated signal to match the phase of the input signal. This block is a phase synchronization block that uses a simple phase locked loop method. A PLL is a feedback control system to automatically adjust the phase of a locally generated signal

$$\hat{s}(t) = \sin(2\pi f_0 t + \theta(t))$$

to the phase of an incoming signal

$$s(t) = \sin(2\pi f_0 t + \theta(t))$$

provided that the frequency of the incoming signal f_0 is known. The limitation of the PLL is that the $\theta(t)$ is a slowly changing variable with respect to the value of f_0.

A typical PLL consists of three components: a phase detector, a filter H(s), and a VCO (voltage controlled oscillator). The phase detector in a simple PLL is a multiplier. The structure of a simple PLL is shown in the figure below:

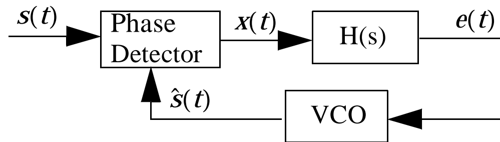


Figure 6-37: The Structure of a Typical Phase-Locked Loop

Note that $\frac{\pi}{2}$ has been added to the VCO as the initial value of the phase shift.

The output of the multiplier, $x(t)$, equals:

$$x(t) = \frac{1}{2}\sin[\theta(t) - \theta(t)] + \frac{1}{2}\sin[2\pi f_0 t + \theta(t) - \theta(t)]$$

After passing through a low-pass filter H(s), the signal *e(t)* becomes: Note that

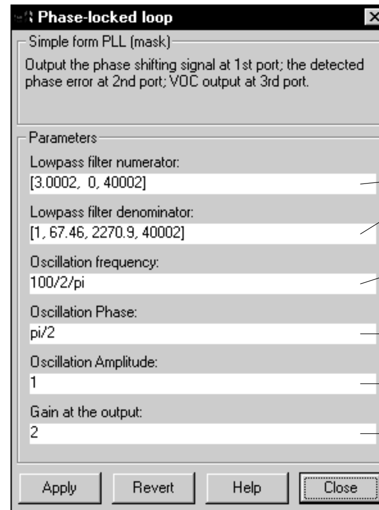
$$e(t) = \frac{1}{2}\sin[\theta(t) - \hat{\theta}(t)]$$

the PLL fails when the deviation of $\theta(t)$ is close to f_0.

H(s) is a lowpass filter. You can use the function `pllfilt` to design the filter.

The input port takes the received transmitting signal. This block has three output ports. Output port 1 outputs the detected phase difference *e(t)*. Output port 2 outputs the detected phase error *x(t)*. Output port 3 outputs the phase matched signal $\hat{s}(t)$ with a $\pi/2$ phase delay.

Dialog Box



Specify the numerator and denominator of the lowpass filter's transfer function.

The VCO oscillator frequency. Match this value to the frequency of the input signal.

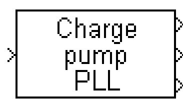
The initial phase of the VCO oscillator signal.

The amplitude of the VCO oscillator signal.

Select a gain for the outputted signal (output port 1).

Characteristics

No. of Inputs/Outputs	1/3
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes



Charge Pump PLL

Catagory Phase Recovery

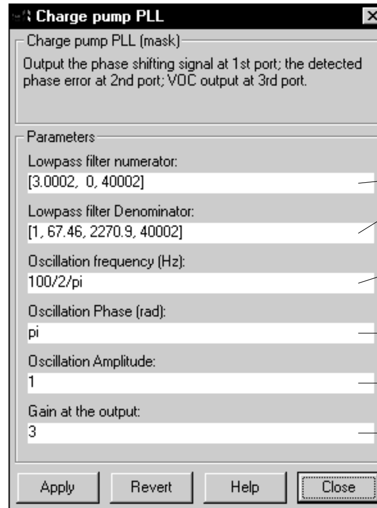
Location Synchronization Sublibrary

Description The Charge Pump PLL (phase-locked loop) block is a phase synchronization block, which means that it automatically adjusts the phase of a locally generated signal to match the phase of an input signal. A typical PLL consists of three components: a phase detector, a filter H(s), and a VCO (voltage controlled oscillator). In the PLL block, a multiplier is used as phase detector. In practice, a number of other phase detectors can be used. One of them is the sequential logic phase detector. A PLL with a sequential logic phase detector (also called a digital phase detector) is known as a charge pump PLL.

A sequential logic phase detector operates on the zero crossings of the signal waveform. The equilibrium point of the phase difference between the input signal and the VCO signal equals π. The sequential logic detector can compensate for any frequency difference that might exist between a VCO and an incoming signal frequency. Hence, the sequential logic phase detector acts as frequency detector. For this reason, a sequential logic phase detector is also called a phase/frequency detector (PFD). A discussion of the charge pump PLL can be found in .

The input port inputs the received transmitting signal. The first outport outputs the detected phase difference. The second outport outputs the phase detector output, which is the detected error. The third outport outputs the phase matched signal with the input except a π phase delay.

Dialog Box



Specify the numerator and denominator of the lowpass filter's transfer function.

The VCO oscillator frequency. Match this value to the frequency of the input signal.

The initial phase of the VCO oscillator signal.

The amplitude of the VCO oscillator signal.

Select a gain for the outputted signal (output port 1).

Characteristics

No. of Inputs/Outputs	1 /3
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Baseband PLL

Baseband Model PLL

Catagory Phase Recovery

Location Synchronization Sublibrary

Description The Baseband Model PLL (phase-locked loop) block is a phase synchronization block that uses a baseband model method. A typical PLL consists of three components: a phase detector, a filter H(s), and a VCO (voltage controlled oscillator)Figure 6-37:. Using the multiplier phase detector and rearranging the variables, the PLL in the figure below is equivalent to the baseband model of the PLL:

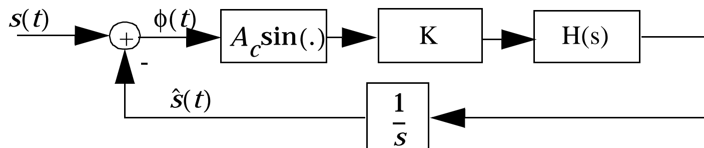
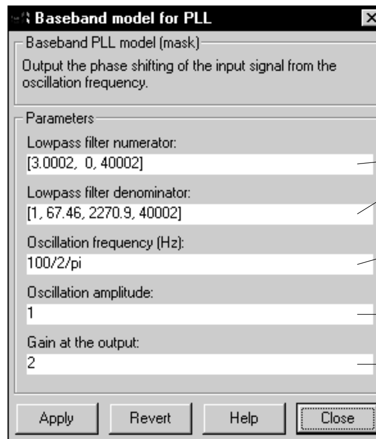


Figure 6-38: PLL Baseband Model

The baseband model for the PLL is a nonlinear system. The model depends on the amplitude A_c of the incoming signal. The model is independent of the carrier frequency f_0.

The input of this block is the received transmission signal. The output of this block is the detected phase difference.

Dialog Box



Specify the numerator and denominator of the lowpass filter's transfer function.

The VCO frequency. Match this value to the frequency of the input signal.

The amplitude of the VCO signal.

Select a gain for the outputted signal (output port 1).

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Linearized baseband PLL

Linearized Baseband Model PLL

Catagory Phase Recovery

Location Synchronization Sublibrary

Description The Linearized Baseband Model PLL block is an implementation of the simplest PLL model. A typical PLL consists of three components: a phase detector, a filter H(s), and a VCO (voltage controlled oscillator)Figure 6-37:.

The baseband model for the PLL is a nonlinear system. It is difficulty to analyze a nonlinear system. Assuming that the phase error $\phi(t)$ is small, the equation $\sin\phi(t) = \phi(t)$ can be used as an approximation. This approximation leads to a linearized baseband model PLL as shown in the figure below:

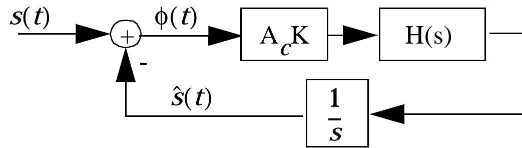


Figure 6-39: Linearized PLL Baseband Model

The transfer function from the input phase $s(t)$ to phase error $\phi(t)$ is

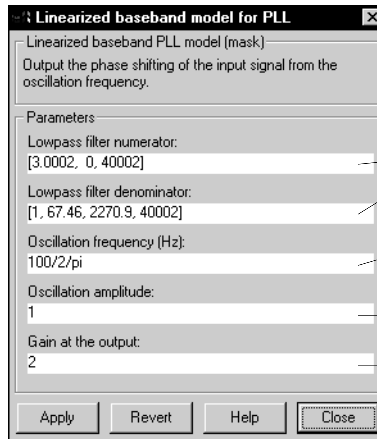
$$G(s) = \frac{s}{s + A_c K H(s)}$$

A PLL without a filter is a first order PLL. A PLL with $(n-1)$th order H(s) is a nth order PLL. It is known that the steady state phase error $\phi(t)$ is zero for a rational H(s). You can use the Control System Toolbox to design and to analyze the PLLs.

The model depends on the amplitude A_c of the incoming signal. The model is independent of the carrier frequency f_0.

The input of this block is the received transmitting signal. The output of this block is the detected phase difference.

Dialog Box



Specify the numerator and denominator of the lowpass filter's transfer function.

The VCO frequency. Match this value to the frequency of the input signal.

The amplitude of the VCO signal.

Select a gain for the outputted signal (output port 1).

Characteristics

No. of Inputs/Outputs	1 /1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Utilities

This toolbox includes a number of useful blocks in communication system design and analysis that do not fit any of the categories mentioned in the previous sections. Many of these blocks are used as the basic blocks to build other function blocks.

This figure shows the Utility Sublibrary:

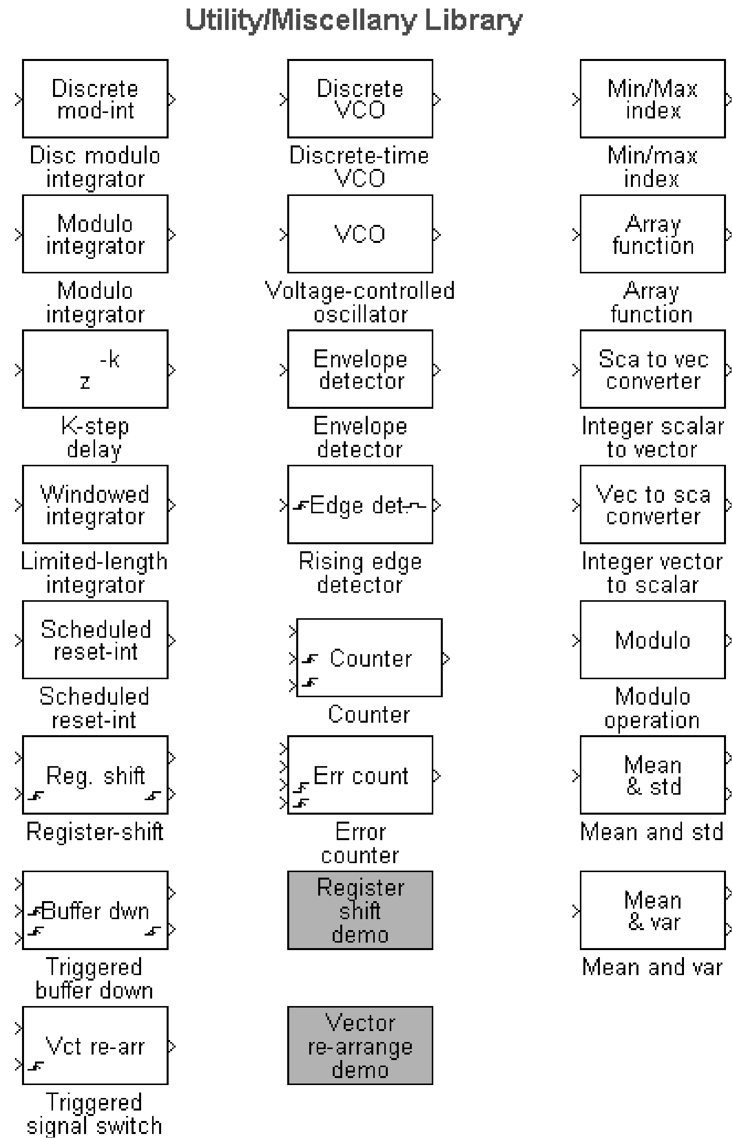


Figure 6-40: Utility Sublibrary

Utility Reference Table

This table lists the Simulink blocks in the Utility Sublibrary. (Blocks are listed in alphabetical order for your convenience.)

Block Name	Description
Array Function	Performs a specified function on each index of an input array.
Complex Filter	Implements IIR and FIR complex filters.
Discrete-Time Modulo Integrator	Discrete-time integrator using a modulo reset.
Discrete Time VCO	Implements a discrete-time voltage controlled oscillator.
Edge Detector	Detects the rising edge of a signal.
Envelope Detector	Detects the upper or lower bound envelope of the input signal.
Error Counter	Counts the number of errors (differences) between two input signals when triggered.
K-Step Delay	Introduces a specified number of time step delays to the input signal.
Mean and Std	Calculates the mean and standard deviation of input data.
Mean and Variance	Calculates the mean and variance of input data.
Min/Max Index	Detects the minimum or maximum element of a vector and outputs the element's index.
Modulo	Performs modulo arithmetic.
Modulo Integrator	Integrates using a modulo reset.

Block Name	Description
Number Counter	Counts the number of input pulses.
Register Shift	Accumulates input scalars and outputs the data in a specified vector format.
Scalar to Vector Converter	Coverts scalars to a vector format.
Scheduled Reset Integrator	Integrates with state reset at specified time steps.
Triggered Buffer Down	Converts an input vector into sequential format when triggered.
VCO	Implements a voltage controlled oscillator.
Vector Redistributor	Redistributes elements from input vectors to output vectors in a specified rearrangement scheme. **Note**: This block is documented in the “Multiple Access Sublibrary” section of this chapter.
Vector to Scalar Converter	Converts vectors to a scalar format.
Windowed Integrator	Integrates with a fixed integration time window.

Discrete
mod-int

Discrete Time Modulo Integrator

Catagory Discrete Reset Integrator

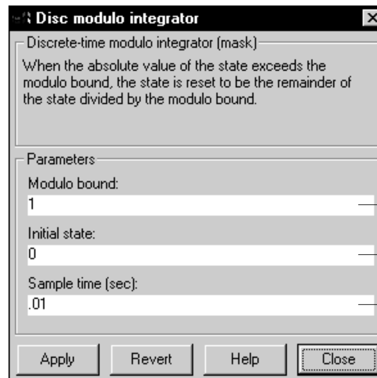
Location Utility Sublibrary

Description

The Discrete Time Modulo Integrator block resets its integration state to a modulo value when the state value exceeds the modulo bound. The block output is `rem(int_out, module_bound)`, where `int_out` is the output of the discrete-time integrator prior to the modulo operation and `modulo_bound` is the output limit.

The discrete-time modulo integrator keeps the output and the integration state in a fixed range. The modulo integrator is useful for monotonically increasing or decreasing functions, but works with any integrable function.

Dialog Box



Set the bound for the state modulo calculation.

Initialize the state.

Specify the integration sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	1
Direct feedthrough	No

Modulo Integrator

Modulo
integrator

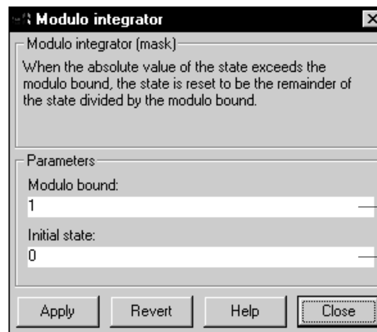
Catagory Reset Integrator

Location Utility Sublibrary

Description The Modulo Integrator block resets its integration state to be modulo value when the state value exceeds the modulo bound. The block output is `rem(int_out, modulo_bound)`, where `int_out` is the output of the integrator prior to the modulo operation and `modulo_bound` is the output limit.

The modulo integrator keeps the output and the integration state in a fixed range. The modulo integrator is useful for monotonically increasing or decreasing functions, but works with any integrable function.

Dialog Box

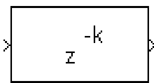


Set the bound for the state modulo calculation.

Initialize the state.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Input/Outputs	No/No
Time Base	Continuous
States	1
Direct feedthrough	No



K-Step Delay

Catagory

Signal Delay

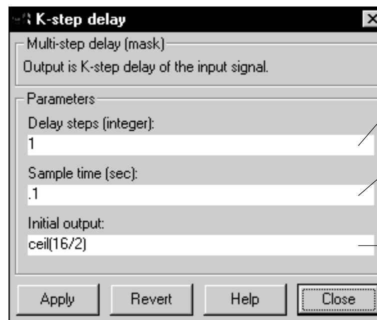
Location

Utility Sublibrary

Description

The K-Step Delay block delays its input signal by a K-sampling time interval. This block inputs and outputs scalar signals, and it is equivalent to a z^{-k} discrete-time operator.

Dialog Box



Specify the number of timesteps in the delay. K must be a positive integer.

The time interval between sample, a two-element vector. The first element is the sample time; the second element, if present, is the offset time from the beginning of the sample period.

A scalar, which specifies the initial condition before the offset time. The initial conditions of the K discrete-time states are always zero.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	K
Direct feedthrough	Yes

Windowed Integrator

Windowed integrator

Catagory Integration over a finite time window

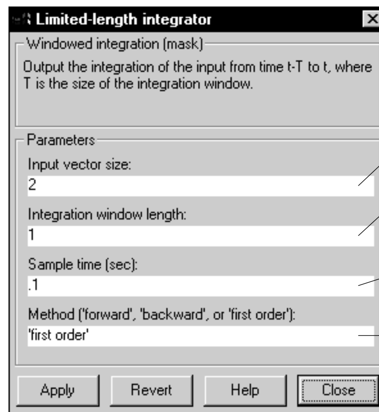
Location Utility Sublibrary

Description The Windowed Integrator block integrates the input signal using a fixed integration time window. The output *y(t)* of the block is:

$$y(t) = \int_{t-T}^{t} x(\tau)\,d\tau$$

where T is a fixed time window for the integration. This block assumes $x(t) = 0$ when $t < 0$. The input signal can be a vector. You must specify the input signal vector length in the parameter assignment. The output vector length equals the input vector length.

Dialog Box



Specify the input vector length.

Specify the length of time for which the integration occurs. This parameter is T in the windowed integrator equation above.

Specify the calculation sample time. This parameter must be less than T.

A string variable. Select the integration method.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Discrete time
States	1
Direct feedthrough	No

Scheduled
reset-int

Scheduled Reset Integrator

Catagory

Integration with reset

Location

Utility Sublibrary

Description

The Scheduled Reset Integrator block resets its states following a fixed resetting schedule. The output of this block *y(t)* is

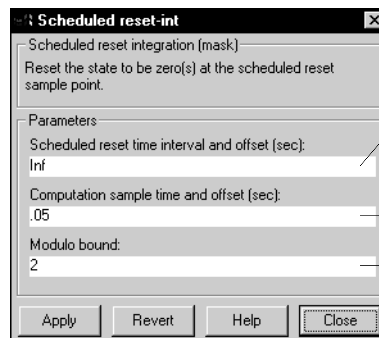
$$y(t) = \int_{ts}^{t} x(\tau)\,d\tau$$

where ts is the last state reset time. The state reset $ts = k *$ time_interval + offset, where k is an integer. The above integration formula applies for the entire time interval.

The block combines the functionality of the Reset Integrator and Modulo Integrator blocks. The block calculates its integration state with a module bound. The output value of this block is equivalent to `rem(y(t), modul e_bound)`, where `y(t)` is the output of the integrator shown in the above equation.

This block can accept a vector input. The output vector size equals the input vector size.

Dialog Box

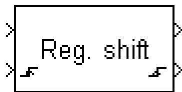


Specify the reset time. Setting this parameter to Inf forces the block to act like a normal discrete time integrator. When this parameter is a two-element vector, the second element is the offset value.

Specify the calculation sample time.

Set the bound for the state module calculation.

Characteristics		
	No. of Inputs/Outputs	1/1
	Vectorized Inputs/Outputs	Yes/Yes
	Time Base	Discrete time
	States	Number of inputs
	Direct feedthrough	No



Register Shift

Catagory I/O Utility

Location Utility Sublibrary

Description

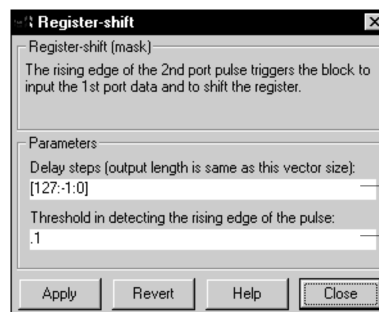
The Register Shift block is an I/O utility that places input data in an internal register and outputs the data following when a trigger signal occurs. This block accumulates the scalar input signal in an internal register at every raising edge of the triggered signal. The output of this block can be a vector or scalar.

The initial conditions of all registers are assigned zeros. When the block detects a trigger, the content in i th index of the register shifts to $(i+1)$th index. The 0th index keeps the current input value. The output follows a FIFO (first in, first out) pattern. The number of indices output when a trigger occurs is equal to the number of delay steps specified in parameter **Delay steps**. Setting **Delay steps** to 1 means that a scalar value will be output when a trigger occurs. Setting **Delay steps** to a value n>1 means that a length n vector will be output when a trigger occurs.

The block detects a trigger when the trigger signal crosses from below the threshold value to above or equal to the threshold value.

Input port 1 takes the input signal value, and input port 2 takes the trigger signal.

Dialog Box



Specify how many delay steps the block uses.

Set the threshold value for the trigger signal.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/Yes
Time Base	Auto
States	The maximum number in the output vector pattern plus one.
Direct feedthrough	Yes, when the minimum number in the output vector pattern is zero; otherwise, no.

Triggered Buffer Down

Catagory I/O Utility

Location Utility Sublibrary

Description

The Triggered Buffer Down block converts a vector signal into sequential signal(s). The block has three input ports. Input port 1 takes a vector signal, input port takes the input-trigger signal, and input port 3 takes the output-trigger signal.

Each time the block detects an input-trigger signal, the block replaces its internal register by the signal input from the first input port. Inside the block, there is an index counter, which keeps the register index number from where the block outputs its value. The block outputs the i th register content when the index counter value is i . Each time the block detects an output-trigger signal, the index counter increases its value specified in the **Increment for each index** parameter. The initial values in the registers are zeros. The initial value for the index counter can be set by the value in the **Initial index** parameter. Each time an input-trigger signal is detected, the index counter is reset to the initial index value.

This block outputs a vector when the value in the increment for each index is a vector. The output vector size is the same as the vector size in the entry. Each output keeps its own index counter and the increment values are added independently. The increment value must be an integer, but it can be a negative integer.

The block detects the raising edge of a trigger when the trigger signal crosses the threshold value from below. The signal does not have to actually cross the threshold; a trigger occurs even if the signal remains constant at the threshold value as long as it approached the threshold value from below.

The first output port outputs the sequential signal(s). The second output port outputs a spike when the internal registers are refreshed.

Example

Let the block input be a length 3 vector. The Initial index is a two-element zero vector. The increment for each index is [1, 2]. The output trigger signal frequency is three times the input-trigger signal frequency. The cyclic signal flag is 1.

As shown in the figure below, at t0, the contents in the registers refresh to [0, 2, 1]. The first output signals at t0, t1, t2 are 0, 2, 1 respectively. The second output signals at t0, t1, t2 are 0, 1, 2 respectively.

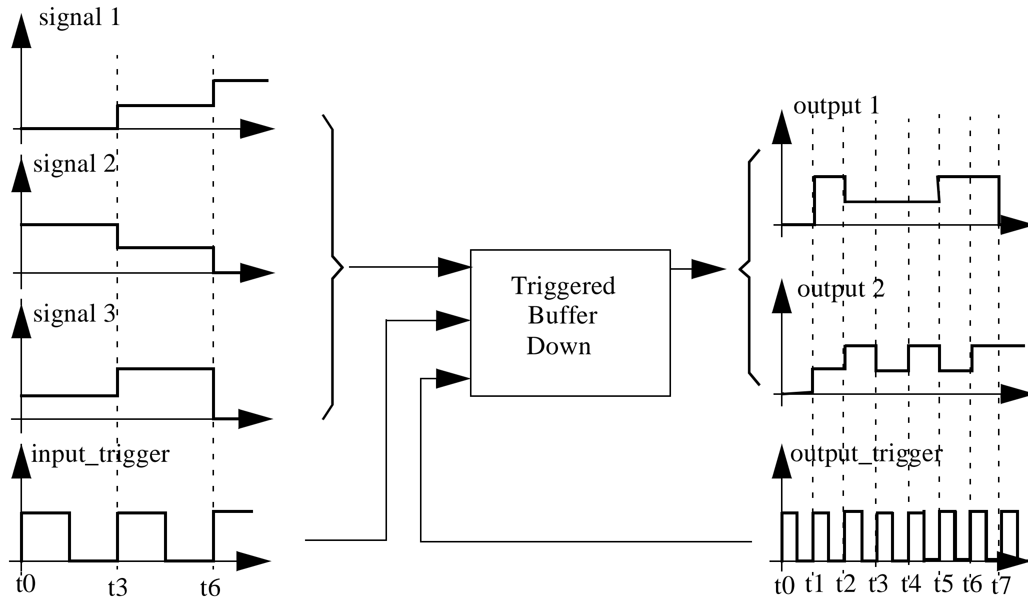
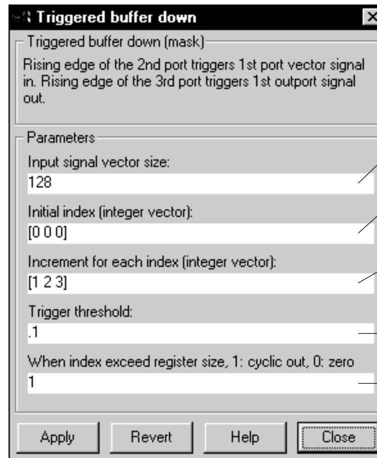


Figure 6-41: Example for Triggered Buffer Down Block

Dialog Box



Specify the input vector length.

Set the initial value for the index counter. This parameter also sets the output vector length.

The increment value for each index. The length of this vector must equal the length of the initial index vector.

Set the threshold for the trigger signal.

When the index exceeds the register size, there are two options. The first option is cyclic, which forces the block to recalculate the block using modulo N arithmetic, where N is the input signal vector size. The second option is just to force the output to 0.

Characteristics

No. of Inputs/Outputs	3/2
Vectorized Input 1	Yes
Vectorized Input 2 and 3	No, No
Vectorized Output 1	Yes
Vectorized Output 2	No
Time Base	Auto
States	The input vector size
Direct feedthrough	Yes

Discrete
VCO

Discrete Time VCO

Catagory Utility

Location Utility Sublibrary

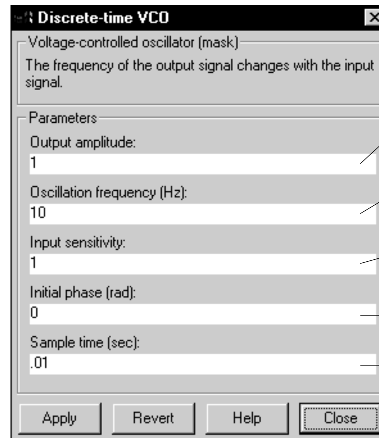
Description The Discrete Time VCO (voltage controlled oscillator) block generates a signal with frequency proportional to the voltage of the input signal. This is a discrete-time block.

For a given input signal *u(t)*, the VCO outputs *y(t)*:

$$y(t) = A_c \cos\left(2\pi f_c t + 2\pi k_c \int_0^t u(t)\,dt + \varphi\right)$$

where k_c is the sensitivity constant of the VCO and φ is the initial phase shift.

Dialog Box



Specify the output amplitude, which is variable A_c in the VCO equation above.

The modulation frequency, which is variable f_c in the VCO equation.

The phase sensitivity value, which is the gain applied to the input signal.

The initial phase of the oscillator.

The calculation sample time.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Discrete time
States	N/A
Direct feedthrough	Yes

Catagory

Voltage Controlled Oscillation

Location

Utility Sublibrary

Description

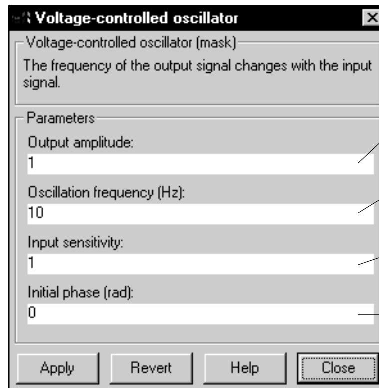
The VCO (voltage controlled oscillator) block generates a signal that is proportional to the voltage of the input signal.

For a given input signal *u(t)*, the VCO outputs *y(t)*:

$$y(t) = A_c \cos\left(2\pi f_c t + 2\pi k_c \int_0^t u(t)\,dt + \varphi\right)$$

where k_c is the sensitivity constant of the VCO and φ is the initial phase shift.

Dialog Box



Specify the output amplitude, which is variable A_c in the VCO equation above.

The modulation frequency, which is variable f_c in the VCO equation.

The phase sensitivity value, which is the gain applied to the input signal.

The initial phase of the oscillator.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Continuous
States	N/A
Direct feedthrough	Yes

Envelope Detector

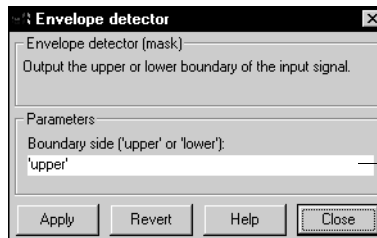
Envelope detector

Catagory Utility

Location Utility Sublibrary

Description The Envelope Detector block detect the upper or lower boundary envelope. You can use this block to outline the envelope of a given signal or in signal demodulation. The block detects the local maximum (when the upper envelope is found) or the local minimum (when the lower envelope is found). Because of the detection method used, the result of the Envelop Detector is noise sensitive. For the best solution, use a lowpass filter before the envelope detector.

Dialog Box



A string variable. Specify whether the block detects the upper or lower boundary.

Characteristics

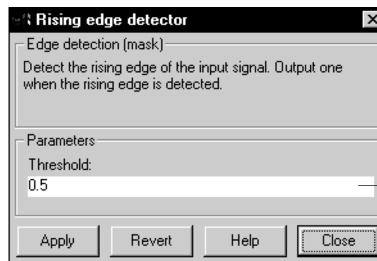
No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Catagory Raising Edge Detection

Location Utility Sublibrary

Description The Edge Detector block detects the raising edge of a signal. This block scans through the input signal and finds the time when the signal crosses from below the specified threshold value. This block outputs one when the raising edge is detected; otherwise it outputs zeros.

Dialog Box



Set the threshold value for signal detection.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Number Counter

Catagory Counter

Location Utility Sublibrary

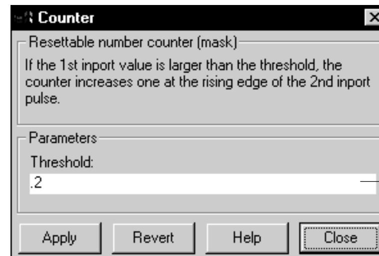
Description The Number Counter block counts the number of input pulses.

The first input port inputs the counting enable/disable signal, the second input port inputs the clock pulse signal, and the third input port inputs the counter reset signal.

At the time when the clock pulse signal (second input port) crosses from below the threshold value *t*, the counter adds one if the enable/disable signal (first port) is larger than or equal to the threshold. The clock pulse signal is ignored if the enable/disable signal is less than the threshold.

The counter has zero initial value. When the reset signal (third port) value crosses the threshold value from below, the counter is reset to zero. The output of this block is the value in the counter.

Dialog Box



Set the threshold for pulse signal counting.

Characteristics

No. of Inputs/Outputs	3/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Error Counter

Catagory

Utility

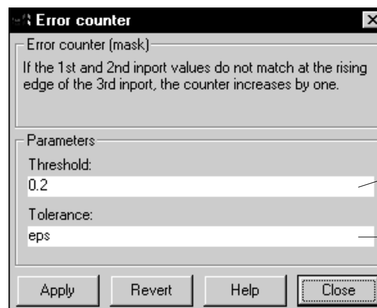
Location

Utility Sublibrary

Description

The Error Counter block counts the number differences between the first input signal and the second signal at the time of the raising edge of the third signal. When the difference between the signal from the first and second ports is larger than a tolerance value, the error counter adds one. The raising edge of the fourth signal resets the error counter. This block accepts scalar signals only.

Dialog Box



Set the threshold value for the rising edge of the third input signal.

Specify the tolerance allowed between the first and second input signals.

Characteristics

No. of Inputs/Outputs	4/1
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Min/Max Index

Catagory

Utility

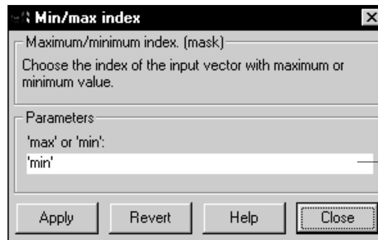
Location

Utility Sublibrary

Description

The Min/Max Index block detects the maximum (or minimum) element of the vector input signal. It outputs (`index`-1) of the maximum (or minimum) element, where `index` is the index of the detected element in the input vector. In other words, if the *i*th element in the input vector has maximum value (or minimum value) in the input vector, the output of this block is `i - 1`. When more than one element share the same maximum value (or minimum value), this block outputs the smallest detected `index`-1.

Dialog Box



A string variable. Select whether the block detects the maximum or minimum vector element.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Array Function

Catagory Multiple input functions

Location Utility Sublibrary

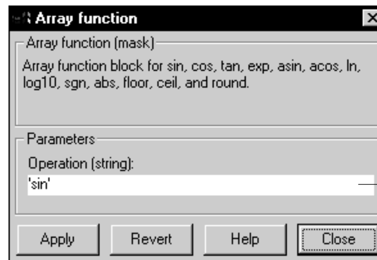
Description

The Array Function block outputs a function calculation of each element of the input vector signal. The length of the output vector is the same as the length of the input vector. Each output element is a function of the input element.

$$y_i = f(u_i)$$

where the function f can be a choice from the strings `sin`, `cos`, `tan`, `asin`, `acos`, `ln`, `log10`, and `sgn`.

Dialog Box



A string variable. Specify which function the block performs on the input vector.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes

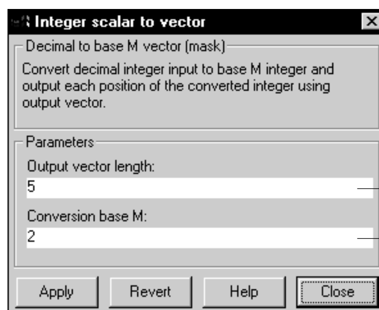
Scalar to Vector Converter

Catagory Data Conversion

Location Utility Sublibrary

Description The Scalar to Vector Converter block converts a scalar base ten positive integer into a number with specified base, b. The output of this block is a vector with given output vector length. The first element in the output vector outputs the coefficient for b^0, the second element in the output vector outputs the coefficient for b^1, and so on.

Dialog Box



Specify the vector length of the output data.

Specify the base for the data conversion. This must be an integer greater than 1.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	No/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block Vector to Scalar Converter

Vec to sca
converter

Vector to Scalar Converter

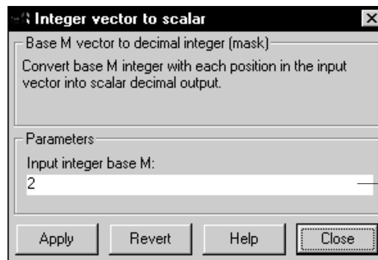
Catagory Data Conversion

Location Utility Sublibrary

Description

The Vector to Scalar Converter block converts the input vector into a scalar positive base ten integer. The input base is `M`, which is specified in the **Input integer base M** parameter. The first element in the input vector is the coefficient for M^0, the second element in the input vector is the coefficient for M^1, and so on.

Dialog Box



The base for the conversion calculation. The entry must be an integer greater than 1.

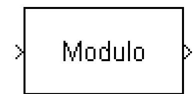
Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/No
Time Base	Auto
States	N/A
Direct feedthrough	Yes

Pair Block

Scalar to Vector Converter

Modulo

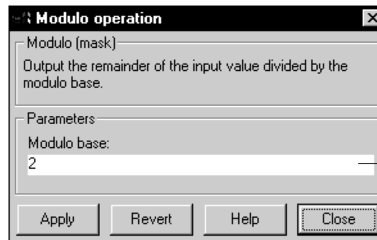


Catagory Modulo arithmetic

Location Utility Sublibrary

Description The Modulo block computes the modulus in a specified base of the input signal. The length of the output vector is the same length as the input vector. The modulo base must be an integer greater than 1.

Dialog Box



Specify the base for the modulo arithmetic calculation.

Characteristics

No. of Inputs/Outputs	1/1
Vectorized Inputs/Outputs	Yes/Yes
Time Base	Auto
States	N/A
Direct feedthrough	Yes

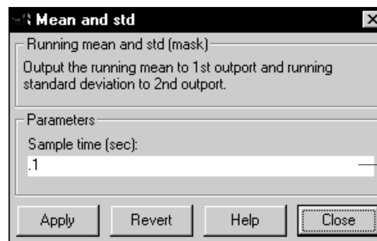
Mean and Std

Catagory Statistics

Location Utility Sublibrary

Description The Mean and Std block calculates the running mean and standard deviation of the scalar input. The block updates the mean and standard deviation at a sample time point specified in the parameter entry. The first output port outputs the mean value, and the second port outputs the standard deviation of the input signal.

Dialog Box



Specify the update calculation time of the block.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	2
Direct feedthrough	Yes

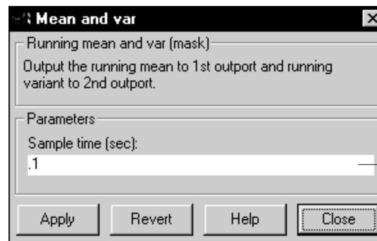
Mean and Variance

Catagory Statistics

Location Utility Sublibrary

Description The Mean and Variance block calculates the running mean and variance of the input scalar signal. The block updates its mean and variance at a sample time point given in the parameter entry. The first output port outputs the mean value, and the second output port outputs the variant of the input signal.

Dialog Box



Specify the update calculation time of the block.

Characteristics

No. of Inputs/Outputs	1/2
Vectorized Inputs/Outputs	No/No
Time Base	Auto
States	2
Direct feedthrough	Yes

A

RTW and Communications

Template Makefiles A-3

Example: TCM Simulation A-5

RTW Guidelines A-9

Limitations And Notes A-11

This section provides information on using the Real-Time Workshop® (RTW) with the Simulink library of the Communications Toolbox. It contains information about building stand-alone C programs from models that contain Communications Toolbox blocks.

For information about...	See...
Editing the template makefile	“Template Makefiles” on A-2
An example code generation task	“Example: TCM Simulation” on A-4
Generating code from models that use MEX-file S-functions	“Writing MEX-File S-Functions for RTW” on A-8
Valid block names for code generation	“Block Names and Code Generation” on A-9
Limitation	“Limitations and Notes” on A-10

See the *Real-Time Workshop User’s Guide* for complete details on code generation.

Template Makefiles

To generate code for your target platform, Real-Time Workshop uses a template makefile. A *template makefile* is the model file that Real-Time Workshop uses to create the actual makefile that builds the executable code. There is a template makefile for every supported target and compiler combination, all with the extension ".tmf". For example:

- On UNIX systems, there is a standard nonreal-time unix.tmf file that works with Sun make or gmake and with cc or gcc.
- On the PC, the nonreal-time template makefiles are usually called dos_*.tmf, where * is an abbreviation for the compiler. For instance, dos_watg.tmf is the template makefile for the WATCOM compiler and Gnu make facility.

Updating the Template Makefile for the Toolbox

Many of the blocks in the Communication Toolbox Simulink library are implemented as C language MEX-files. If you use the Real-Time Workshop to generate C code for a model that uses the blocks, you must ensure that your template makefile references their MEX-file source. Modify your template makefile as follows:

- Add the Communication Toolbox commsfun directory to the include path used by your C compiler. For example, to do this with the unix.tmf template makefile, add the text shown in **bold** to the MATLAB_INCLUDES macro definition.

```
MATLAB_INCLUDES = \
        -I$(MATLAB_ROOT)/simulink/include \
        -I$(CODEGEN_ROOT)/common/include \
        -I$(CODEGEN_ROOT)/external/rpc \
        -I$(MATLAB_ROOT)/toolbox/comm/commsfun
```

To do this with the `dos_watg.tmf` file, add the text shown in **bold** to the `MATLAB_INCLUDES` macro definition.

```
MATLAB_INCLUDES = \
        -I$(CODEGEN_ROOT)\nrt \
        -I$(MATLAB_ROOT)\simulink\include \
        -I$(MATLAB_ROOT)\simulink\src \
        -I$(CODEGEN_ROOT)\common\include \
        -I$(MATLAB_ROOT)\toolbox\comm\commsfun
```

The various template makefiles use different make facilities which in general have different syntaxes, so edit your file carefully.

- Add a rule for compiling sources in the Communications Toolbox `commsfun` directory. Make sure that the indented lines start with a single tab, not spaces.

  For example, to do this with the `unix.tmf` template makefile, add the text shown in **bold** to the list of rules:

```
%.o : $(MATLAB_ROOT)/simulink/src/%.c
    $(CC) $(CFLAGS) -c $<

%.o : $(MATLAB_ROOT)/toolbox/comm/commsfun/%.c
    $(CC) $(CFLAGS) -c $<
```

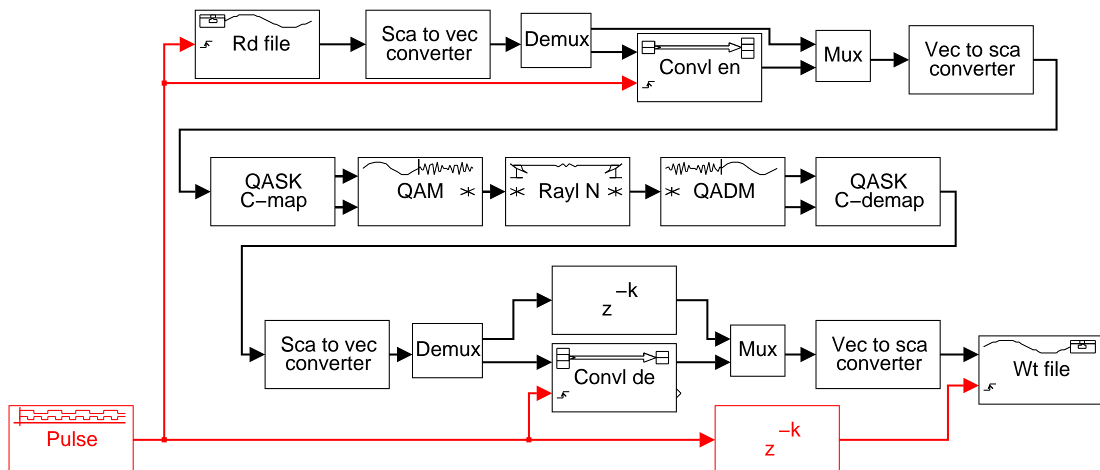
  To do this with the `dos_watg.tmf` file, add the text shown in **bold** to the list of rules.

```
%.obj : $(MATLAB_ROOT)\simulink\src\%.c
    $(PRINTENV) CFLAGS
    $(CC) @CFLAGS $<

%.obj : $(MATLAB_ROOT)\toolbox\comm\commsfun\%.c
    $(PRINTENV) CFLAGS
    $(CC) @CFLAGS $<
```

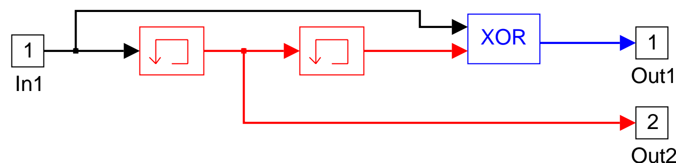
For more information on the template makefiles provided with the Real-Time Workshop, see the section “Template Makefiles” in Chapter 3 of the *Real-Time Workshop User’s Guide*.

Example: TCM Simulation

This section describes how to generate C code for an example model. The goal of this example is to create a stand-alone DOS or UNIX executable that reads a signal from a `test1.dat` file. The contents of the file is transmitted from the sender to the receiver by using the trellis code modulation (TCM) method, the receiver creates a new `test2.dat` file containing the received data. To try this example, load the block diagram `tsttcm` from the MATLAB workspace.



This block uses the convolution code with its transfer functions showing in the next figure.



Create a data file test1.dat in your working space:

```
N = 300;
x = randint(N, 1, 4);
fid = fopen('test1.dat');
for i = 1:length(x)
   fprintf(fid,'%d\n',x(i));
end
fclose(fid);
```

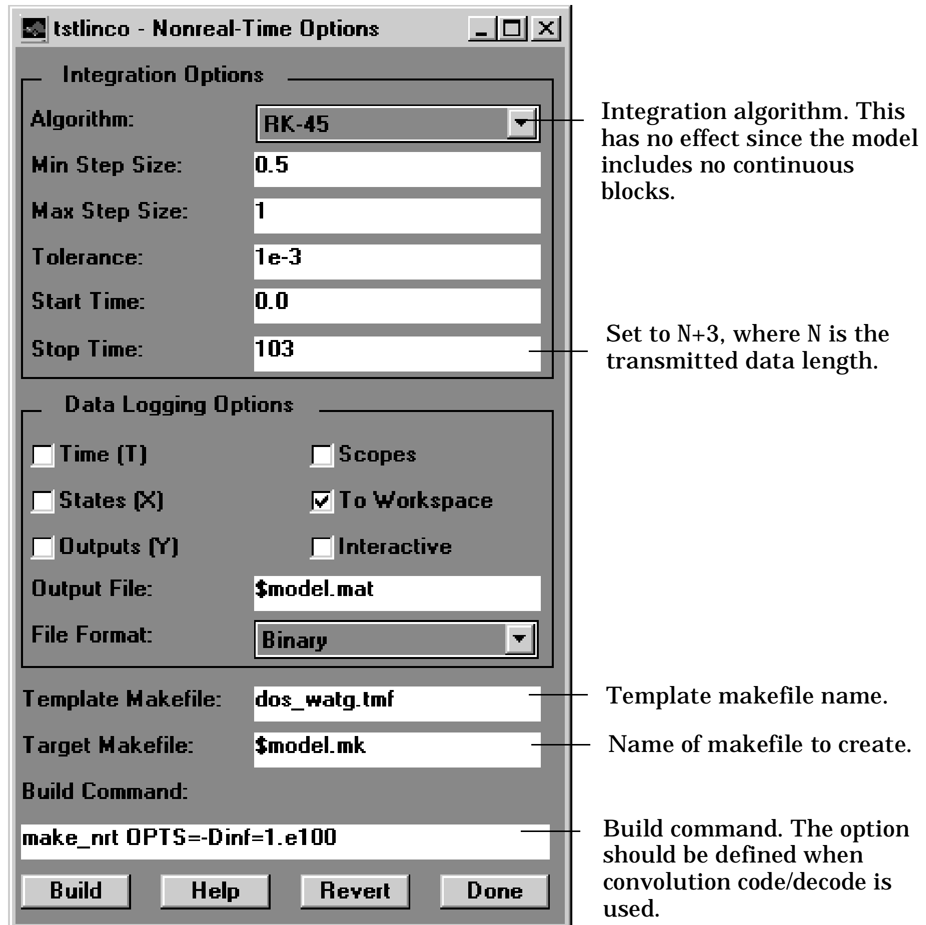
To simulate the system, set the stop time to be N+3 from the parameter of the simulation menu, where N is given in the above code. Then, start the simulation from the simulation menu.

The simulation results in a data file test2.dat, which has the same contents (except the transfer error) with the file test1.dat. You can use the command load test1.dat to compare the transmitted data x and the received data test2.

To generate code, follow the procedures:

- Select **Nonreal-Time Options** from the **Code** menu.

  Set the code generation parameters for your model in the dialog box that appears. Note that the figure below shows appropriate settings for one possible DOS configuration:



- Once you have set all the parameters, click on **Build**.

  Real-Time Workshop automatically creates, compiles, and links the C code according to the specifications in your template makefile.
- Run the executable.
- From the MATLAB prompt, enter:

```
!tsttcm
```

- Or, from the operating system prompt, enter:

```
tsttcm
```

  Your results are in the file `test2.dat`. Use the `load` command to load the file into MATLAB and examine the model output. The produced file `test2.dat` should be the same as the contents in `test1.dat` file except three zeros has been added to the beginning of the `test2.dat` file.

Once you have generated C code for a model, you can edit the code to optimize it for your application.

RTW Guidelines

This section provides a brief summary of some guidelines that apply to Real-Time Workshop code generation tasks. It includes:

- Writing MEX-File S-Functions for RTW
- Block Names and Code Generation

Writing MEX-File S-Functions for RTW

Remember these guidelines any time you generate code from a model:

- The MEX-file cannot call any MATLAB routines.
- If the MEX-file uses any routines from the MEX External Interface Library, include `cg_matrx.h` instead of `mex.h`. To handle this case, make sure the lines below are present in your MEX-file:

```
#ifdef MATLAB_MEX_FILE
#include "mex.h"
#else
#include "cg_matrx.h"
#endif
```

- `cg_matrx` defines all of these functions as macros:
  - `mxGetM`
  - `mxGetN`
  - `mxGetPr`
  - `mxGetString`
  - `mxSetM`
  - `mxSetN`
  - `mxSetPr`
  - `mxCreateFull`
  - `mxFreeMatrix`

All other MEX library calls are unsupported outside of the MEX interface. Bracket them with the statements

```
#ifdef MATLAB_MEX_FILE
#endif
```

Only real, full matrices are supported.

- Make sure that calls to `mxGetString` do not specify a return value.

  If you specify a return value for `mxGetString`, the MEX-file will not compile correctly due to a limitation of the `mxGetString` macro provided in `cg_matrx`. Obtain the desired string through the function's second argument, which returns a pointer to the string.
- Make sure that the `#define S_FUNCTION_NAME` statement is correct. The specified S-function name must match the filename.

See the *Real-Time Workshop User's Guide* for more information on MEX-file S-functions.

Block Names and Code Generation

Block names cannot end with an asterisk (`*`) if you want to compile code from the model. Real-Time Workshop creates a comment in the C code that includes the block name. If you have a masked system that includes other blocks and whose names ends with `*`, the C code comment will include the string `*/`. This will probably produce a syntax error, but could compile and produce unexpected results.

Limitations And Notes

RTW can generate code with C coded S-functions only. You cannot generate code with a M coded S-function. There are a few display blocks that cannot be used with RTW. The M-function blocks in the toolbox are:

Blocks	Library
Eye Pattern Diagram	Source/Sink
Error Rate Meter	Source/Sink

In addition, the Convolution Decode block can be used with RTW only when the Trellis Figure Plot Length is set to be zero, which means that no trellis will be plotted.

When Convolution Encode block and/or Convolution Decode block is presented in the block diagram, you have to set the build command as follows:

```
make_nrt OPTS=-Dinf=1.e100
```

The reason for doing so is that the transfer function inputted to the Convolution Encode block and Convolution Decode block uses `inf` in the input.

Bibliography

Benedetto, S., E. Biglieri, and V. Castellani, 1987, *Digital Transmission Theory*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-214313-5 TK5103.7.B46.

Best, Roland E., 1993, *Phase-Locked Loops: Theory, Design and Applications*, McGraw-Hill Book Company, New York, NY.
ISBN 0-07-911386-9 TK7872P38B48.

Black, H.S., 1953, *Modulation Theory*, D.Van Nostrand Company, Inc, Princeton, NJ. TK 5101 B55.

Cadzow, J.A., and H.F. Van Landingham, 1985, *Signals, Systems, and Transforms*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-004151-3 TK7871.6.C627.

Campopiano, C.N., B.G. Glazer, 1962, "A Coherent Digital Amplitude and Phase Modulation Scheme," *IRE trans. CS*-10:90-95.

Cassandras, Christos G., 1993, *Discrete Event Systems, Modeling and Performance Analysis*, Irwin and Aksen Associate Inc. Boston, MA. ISBN 0-256-11212-6 T57.6.C38.

Cahn, C.R., 1960, Combined Digital Phase and Amplitude Modulation Communication Systems, *IRE trans. CS*-8:150-155.

Chen, D. and C. Moler, *Symbolic Math Toolbox*, The MathWorks, Inc., Natick, MA.

Cheng, David K., 1989, *Field and Wave Electromagnetics*, Addison Wesley, Reading, MA. ISBN 0-201-12819-5 QC760.C48.

Compton, R.T. Jr., 1988, *Adaptive Antennas, Concepts and Performance*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-004151-3 TK7871.6.C627.

Couch, Leon W. II, 1993, *Digital and Analog Communication Systems*, Macmillan Publishing Company, New York, NY.
ISBN 0-02-325281-2 TK5101.C69.

Cover, T.M. and J.A. Thomas, 1991, *Elements of Information Theory*, John Wiley and Sons, New York, NY.
ISBN 0-471-06259-6 Q360.C68.

Forney, G.D., R.G. Gallager, G.R. Lang, F.M. Longstaff, and S.U. Qureshi, 1984, “Efficient Modulation for Band-Limited Channels,” *IEEE Journal SAC*-2:632-646.

Foschini, G.J., R.D. Gitlin, and S.B. Weinstein, 1974, “Optimization of Two Dimensional Signal Constellations in the Presence of Gaussian Noise,” *IEEE trans. COM*-22:28-38.

Freeman, Roger L., 1991, *Telecommunication transmission handbook*, John Wiley and Sons, New York, NY.
ISBN 0-471-51816-6 TK 5101.F66.

Gagliardi, Robert M., 1988, *Introduction to Communications Engineering*, John Wiley and Sons, New York, NY.
ISBN0-471-85644-4 TK 5101.G24.

Gardner, F.M. “Charge-pump phase-lock loops,” *IEEE Trans. on Communications*, Vol. 28, No. 11, pp 1894-1858, Nov, 1980.

Gardner, F.M. “Phase accuracy of charge pump PLLs,” *IEEE Trans. on Communications*, Vol. 30, No 10, pp 2362-2363, Oct. 1982.

Gersho, A. and R.M. Gray, 1991, *Vector quantization and signal compression*, Kluwer Academic Publishers, Boston, MA.
ISBN 07923-9181-0, TK5102.5.G45.

Grace, A., A.J. Laub, J.N. Little, C.M. Thompson, *Control System Toolbox*, The MathWorks, Inc., Natick, MA.

Gray, R.M, 1990, *Source Coding Theory*, Kluwer Academic Publishers, Boston, MA. ISBN 0-7923-9048-2, TK5102.5.G69.

Gruber, J.J. and G. Williams, 1992, *Transmission Performance of Evolving Telecommunications Networks*, Artech House, Boston, MA.
ISBN 0-89006-591-8, TK5102.5.G753.

Halsall, F., 1992, *Data Communications, Computer Networks and Open Systems*, Third Edition, Addison Wesley, Reading, MA.
ISBN 0-201-56506-4 TK5105.H35.

Hambley, Allan R., 1989, *An Introduction to Communication Systems*, Computer Science Press, New York, NY.
ISBN 0-7167-8184-0 TK 5103.7.H36.

Handel, Rainer, Manfred N. Huber, and Stefan Schroder, 1994, *ATM Networks, Concepts, Protocols, Applications*, Second Edition, Addison Wesley, Reading, MA. ISBN 0-201-42274-3.

Helstrom, Carl W., 1995, *Elements of Signal Detection & Estimation*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-808940-X TK6588.H45.

Haykin, S, 1994, *Communication Systems*, John Wiley and Sons, New York, NY. ISBN0-471-57178-8 TK 5101.G24H37.

Immink, Kees A.S., 1991, *Coding Techniques for Digital Recorders*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-140047-9 TK 5102.S34.

Jeruchim, C.J, P. Balaban, and K.S. Shanmugan, 1992, *Simulation of Communication Systems*, Plunum Press, New York, NY.
ISBN 0-306-43989-1 TK5102.5.J47.

Krauss, T.P., L. Shure, and J.N. Little, 1993, *Signal Processing Toolbox*, The MathWorks, Inc., Natick, MA.

Lee, B.G., M. Kang, and J. Lee, 1993, *Broadband Telecommunications Technology*, Artech House, Boston, MA.
ISBN 0-89006-653-X, TK5103.7.L43.

Lee, Edward A., David G. Messerschmitt, 1994, *Digital Communication*, second edition, Kluwer Academic Publishers, Boston, MA.
ISBN 0-7923-9391-0, TK5103.7.L44.

Lee, W.C.Y., 1989, *Mobile Cellular Telecommunication Systems*, McGraw-Hill Book Company, New York, NY.
ISBN 0-07-037030-3 TK6570.M6L35.

Lin, Shu and Costello, Daniel J. Jr., 1983, *Error Control Coding*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-283796-X QA268.L55.

Lucky, R.W, and J.C. Hancock, 1962, On the Optimum Performance of N-ary Systems Having Two Degrees of Freedom, *IRE trans. CS*-10:185-192.

Lynch, Thomas J., 1985, *Data Compression Techniques and Applications*, Van Nostand Reinhold, New York, NY.
ISBN 0-534-03418-7 QA 76.9.D33L96.

Maclean, T.S.M., 1986, *Principles of antennas: wire and aperture*, Cambridge University Press, Cambridge, U.K.
ISBN 0-521-30668 X TK7871.6.M33.

Maral, G. and Bousquet, M., 1993, *Satellite Communications Systems*, John Wiley and Sons, New York, NY.
ISBN 0-471-93032 TK 5104.M3613.

Martin, James, 1988, *Data Communication Technology*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-196643-x 025 TK5105.M3558.

Meyr, Heinrich and Ascheid, Gerd, 1990, *Synchronization in Digital Communications*, Vol. 1, Phase- Frequency-Locked Loops, and Amplitude Control, John Wiley and Sons, New York, NY.
ISBN 0-0471-50193-X (v.1) TK 5103.7.M49.

Miller, Michael J. and Ahamed, Syed, 1988, *Digital Transmission Systems and Networks*, Vol. 2, Applications, Computer Science Press, Rockville, MD. ISBN 0-88175-094-8 (v1)
ISBN 0-88175-176-6 (v2) TK5103.7.M5.5.

Nicholson, David L., 1988, *Spread Spectrum Signal Design*, LPE and AJ Systems, Computer Science Press, Rockville, MD.
ISBN 0-88175-102-2 TK5102.5.N49.

Oppenheim, Alan V. and Schafer, Ronald W., 1975, *Digital Signal Processing*.

Proakis, J.G. and M. Salehi, *1994, Communication Systems Engineering*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-158932-6 TK 5101.P75.

Ramo, S., J.R. Whinnery, and T.V. Duzer, 1994, *Fields and Waves in Communication Electronics*, John Wiley and Sons, New York, NY. ISBN 0-471-58551-3 QC665.E4R36.

Richharia, M., 1995, *Satellite Communication Systems, Design Principles*, McGraw-Hill Book Company, New York, NY.
ISBN 0-07-052374-6.

Rosenfeld, Azriel and Kak, Avinash, C., 1982, *Digital Picture Processing*, Vol. 1, ISBN 0-12-597301-2 TA1630.R6.7.

Schwartz, Mischa, 1980, *Information Transmission, Modulation, and Noise – a Unified Approach to Communication Systems*, McGraw-Hill Book Company, New York, NY. ISBN 0-07-055782-9 TK5101.S3.

Senior, J.M, 1992, *Optical Fiber Communications*, *Principles and Practice*, Second Edition, Prentice-Hall, Inc. Englewood Cliffs, NJ. ISBN0-13-635426-2 TK5103.59.S46.

Simon, Marvin K., Sami M. Hinedi, and William C. Lindsey, 1995, *Digital Communication Techniques, Signal Design and Detection*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-200610-3 TK7872.F73S75.

Sklar, Bernard, 1988, *Digital Communications: Fundamentals and Applications*, Prentice-Hall, Inc. Englewood Cliffs, NJ. ISBN 0-13-211939-0 TK 5103.7.S55.

Stearns, S.D. and D.R. Hush, 1990, *Digital Signal Analysis*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-582016-2 TK5103.7.S52.

Stirling, R.C., 1987, *Microwave Frequency Synthesizer*, Prentice- Hall, Inc. Englewood Cliffs, NJ. ISBN0-13-213117-X TK5.102.5S698.

Taub, Herbert and Schilling, Donald L., 1986, *Principles of Communication Systems*, McGraw-Hill Book Company, New York, NY. ISBN 0-07-055782-9 TK5101.S3.

Thomas, C.M, M.Y. Weidner, and S.H. Durrani, 1974, “Digital Amplitude-Phase Keying with M-ary Alphabets,” *IEEE trans. COM*-22:168-180.

Tzannes, Nicolaos S., 1985, *Communication and Radar Systems*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN0-13-153545-5 TK6575.T95.

Ungerboeck, G, 1982, “Channel Coding with Multilevel/Phase Signals,” *IEEE trans. IT*-28:55-67.

Ungerboeck, G., 1987, “Trellis-Coded Modulation with Redundant Signal Sets, Part I: Introduction,” *IEEE Communication Magazine*, Vol. 25, No. 2, pp5-11.

Ungerboeck, G., 1987, “Trellis-Coded Modulation with Redundant Signal Sets, Part II: State of the Art,” *IEEE Communication Magazine*, Vol. 25, No. 2, pp12-21.

Van Trees, Harry L., 1968, *Detection, Estimation, and Modulation Theory*, John Wiley and Sons, Inc., New York, NY.

Veldhuis, Raymond, and Marcel Breeuwer, *1993, An Introduction to Source Coding*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-489089-2 TK 5102.5.V45.

Viterbi, Andrew J., 1995, *CDMA Principles of Spread Spectrum Communication*, Addison Wesley, Reading, MA.
ISBN 0-201-63374-4 TK5103.45.V57.

Webb, William T., and Lajos Hanzo, 1994, *Modern Quadrature Amplitude Modulation, Principles and Applications for Fixed and Wireless Communications*, IEEE Press, New York, NY.
ISBN 0-7273-1701-6.

Wilson, Stephen G., *1996, Digital Modulation and Coding*, Prentice-Hall, Inc. Englewood Cliffs, NJ.
ISBN 0-13-210071-1 TK 5100.9.W55.

Zayed, A.I., 1993, *Advances in Shannon's Sampling Theory,* CRC Press, Ann Arbor, MI. ISBN 0-8493-4293-7 QA276.6.Z316.

Symbols

μ-Law Compressor block 6-51
μ-Law Expander block 6-52

A

addition in GF(q^M) 3-136
additive white Gaussian noise (AWGN) 3-5
ADM CE with Carrier block 6-151
ADM with Carrier block 6-133
A-law compander 3-19
A-Law Compressor block 6-49
A-Law Expander block 6-50
alphabet size. *See* M-ary number 3-9
AM with Carrier block 6-131
AM With Carrier CE block 6-150
analog modulation/demodulation passband reference table 6-114
arbitrary constellation 3-76
Array Function block 6-307
array functions 3-128
AWGN (additive white Gaussian noise) 3-5
AWGN Channel block 6-263

B

baseband analog demodulation 3-92
baseband analog modulation 3-90
baseband channels 3-121
baseband digital demodulation 3-97
baseband digital modulation 3-93
Baseband Model PLL block 6-17, 6-259, 6-266, 6-267, 6-281
baseband modulation demodulation 3-89
BCH code 3-33
BCH Code View Table block 6-63
BCH Decode Sequence In/Out block 6-72
BCH Decode Vector In/Out block 6-68
BCH decoding 3-36
BCH Encode Sequence In/Out block 6-70
BCH Encode Vector In/Out block 6-66
BCH encoding 3-34
Binary Vector Noise Generator block 6-13
bit error rate 3-12
block names
    for code generation A-10
burst error correction code 3-36

C

C language
    generating code from a model A-2
Calculations in Galois fields 3-130
CDMA (code division multiple access) 3-110
channel reference table 6-262
channels 3-120
    baseband 3-121
    fading 3-121
    passband 3-120
    Rayleigh noise 3-121
    Rician noise 3-121
Charge Pump PLL block 6-34, 6-279
circle constellation 3-75
code division multiple access (CDMA) 3-110
code generation A-2
    example A-5
    guidelines A-9
    valid block names A-10
code rate 3-23
codebook 3-14
    training of 3-16
Coherence MFSK Corr Demod block 6-227
Coherence MFSK Demod block 6-175
Coherence MFSK Demod CE block 6-203

coherent demodulation 3-81, 3-92
coherent vs. noncoherent demodulation 3-92
Communications Toolbox 1-6
    installation 1-5
    organization 1-6
Complex Filter block 6-260
constellation 3-68
    arbitrary 3-76
    circle 3-75
    square 3-74
constellation size. *See* M-ary number
constraint length 3-23
continuous phase shift-keying methods (MFSK and MSK) 3-96
convolutional code 3-37
    transfer functions 3-38
convolutional code transfer functions
    binary form 3-40
    nonlinear form 3-41
    octal form 3-40
    state space form 3-41
Convolutional Decode Sequence In/Out block 6-108
Convolutional Decode Vector In/Out block 6-104
convolutional decoding 3-44
    hard decision decoding 3-46
    soft decision decoding 3-44
Convolutional Encode Sequence In/Out block 6-106
Convolutional Encode Vector In/Out block 6-102
convolutional encoding 3-44
Costas phase-locked loop 3-56
Cyclic code 3-33
Cyclic Decode Sequence In/Out block 6-95
Cyclic Decode Vector In/Out block 6-92
Cyclic Encode Sequence In/Out block 6-93
Cyclic Encode Vector In/Out block 6-91
cyclotomic coset 3-137
cyclotomic cosets of GF(q^M) 3-137

D

data buffering 2-16
data compression 2-3
data flow simulation 2-10
decision point 3-8
demapping 3-8
demo blocks 2-6
demodulation
    coherent 3-81, 3-82, 3-92
    coherent vs. noncoherent 3-92
    Costas phase-locked loop 3-56
    effect of lowpass filters on 3-54
    noncoherent 3-93
differences between Simulink and MATLAB simulation 2-10
differential pulse code modulation (DPCM) 3-20
digital mod/demod map/demap baseband reference table 6-185
digital modulation 2-3
digital TDMA 3-107
Discrete Time Module Integrator block 6-289
Discrete Time VCO block 6-300
display devices 3-3, 3-5, 3-7
distortion
    of a quantized signal 3-15
double-sideband suppressed carrier amplitude modulation and demodulation 3-56
double-sideband with transmission carrier amplitude modulation and demodulation 3-57
DPCM (differential pulse code modulation) 3-20
DPCM Decode block 6-48
DPCM Encode block 6-46
DSB ADM block 6-116

DSB ADM CE block 6-138
DSB-SC AM block 6-115
DSB-SC AM CE block 6-137
D-TDMA Demux block 6-240
D-TDMA Mux block 6-238

E

Edge Detector block 6-303
Envelope Detector block 6-302
envelope, fading 6-14
equivalent systematic linear code 3-31
Error Counter block 6-305
Error Rate Meter block 6-35
error-control coding 2-3
  code rate 3-23
  constraint length 3-23
  linear code 3-23
error-control coding reference table 6-54
error-rate analysis 3-12
extension fields 3-130
eye-pattern 6-28
Eye-Pattern & Scatter Plot block 6-28
eye-pattern diagrams 3-8
eye-pattern plot 2-14

F

fading channels 3-121
fading envelope 6-14
FDM block 6-124
FDM CE block 6-144
FDMA (frequency division multiple access) 3-108
filters
  Hilbert transform 3-118
  raised cosine 3-113
  sinc 3-118
  transmitting and receiving 3-113
FM (frequency modulation) 3-65
FM block 6-122
FM CE block 6-142
frequency division multiple access (FDMA) 3-108
frequency modulation (FM) and demodulation 3-65

G

Galois fields
  calculations in 3-130
  extension fields 3-130
  prime fields 3-130
Gaussian noise 3-5
Gaussian Random Noise Generator block 6-9
generator matrix 3-30
generator polynomial 3-33
GF(q) 3-130
  other related functions 3-138
  polynomials over 3-131
GF(q^M) 3-132
  addition in 3-136
  cyclotomic cosets of 3-137
  minimal poylonials over 3-137
  multiplication in 3-136
  polynomial roots in 3-138

H

Hamming code 3-32
Hamming Decode Sequence In/Out block 6-61
Hamming Decode Vector In/Out block 6-58
Hamming distance 3-46
  survivor 3-47
Hamming Encode Sequence In/Out block 6-59
Hamming Encode Vector In/Out block 6-57

hard decision decoding 3-44, 3-46
Hilbert transform filter 3-118

I

independent, identically distributed (i.i.d) 6-8
input/output
  sequential format 3-27
  vector format 3-26
  vector vs. sequential format 3-25
installation 1-5
integration
  modulo 3-127
  scheduled reset 3-128
  windowed 3-128

K

K-Step Delay block 6-291

L

linear block code 3-30
  syndrome 3-31
Linear Block Decode Sequence In/Out block 6-100
Linear Block Decode Vector In/Out block 6-97
Linear Block Encode Sequence In/Out block 6-98
Linear Block Encode Vector In/Out block 6-96
linear code 3-23
  equivalent systematic 3-31
Linearized Baseband Model PLL block 6-283
log likelihood
  in soft decision decoding 3-44

M

makefile *See* template makefile
M-ary amplitude shift-keying modulation (MASK) 3-71
M-ary number 3-9, 6-211
M-ary phase shift-keying (MPSK) modulation 3-87
M-ary quadrature shift-keying (MQSK) modulation 3-73
MASK (M-ary shift-keying modulation) 3-71
MASK Demap block 6-215
MASK Demod block 6-159
MASK Demod CE block 6-188
MASK Map block 6-213
MASK Mod block 6-157
MASK Mod CE block 6-187
MATLAB 1-3
mean and variance 3-129
Mean and Variance block 6-312
memoryless codes 3-37
metric
  in soft decision decoding 3-46
MEX-files A-9
MFSK Map block 6-225
MFSK Mod block 6-173
MFSK Mod CE block 6-202
Min/Max Demap block 6-226
Min/Max Index block 6-306
minimal polynomial over GF(q^M) 3-137
minimum/maximum index 3-128
m-law compander 3-18
modulation
  baseband analog 3-90
  baseband digital 3-93, 3-97
  digital 2-3
  passband and baseband 3-52
modulation and demodulation 3-52

baseband 3-89
constellation 3-68
double-sideband suppressed carrier amplitude 3-56
double-sideband with transmission carrier amplitude 3-57
frequency 3-65
passband analog 3-53
passband digital 3-67
quadrature amplitude modulation (QAM) and demodulation 3-62
signal space 3-68
single-sideband suppressed carrier amplitude 3-60
types of 3-69
Modulo block 6-310
modulo integration 3-127
Modulo Integrator block 6-290
modulo operation 3-129
MPSK (M-ary phase shift-keying) modulation 3-87
MPSK Correlation Demodulation block 6-233
MPSK Demod block 6-181
MPSK Demod CE block 6-206
MPSK Map block 6-232
MPSK Mod block 6-179
MPSK Mod CE block 6-205
MQSK (M-ary quadrature shift-keying) 3-73
multiple access 2-3
signal multiple access 3-104
multiple access reference table 6-237

N

noise
Gaussian 3-5
Poisson random integer 3-7
Rayleigh 3-6
Rician 3-6
uniform 3-7
noncentrality parameter of Rician noise 3-6
Noncoherence MFSK Corr Demod block 6-229
Noncoherence MFSK Demod block 6-177
Noncoherence MFSK Demod CE block 6-204
noncoherent demodulation 3-82, 3-93
Number Counter block 6-304
number of error bits 3-12
number of symbol errors 3-12

O

organization of this manual 1-6

P

parity-check matrix 3-31
partition
training of 3-16
vector 3-14
passband and baseband modulation 3-52
passband channels 3-120
passband digital modulation and demodulation 3-67
PDM block 6-127
PDM CE block 6-146
PLL block 6-32, 6-277
PM block 6-125
PM CE block 6-145
Poisson Random Integer Generator block 6-12
Poisson random integer noise 3-7
polynomial roots in GF(q^M) 3-138
polynomials over GF(q) 3-131
predictive quantization 3-19
predictor 3-20
prime fields 3-130

Q

QADM block 6-120
QADM CE block 6-140
QAM (quadrature amplitude modulation) 3-62
QAM block 6-118
QAM CE block 6-139
QASK Demap Arbitrary Constellation block 6-224
QASK Demap Circle Constellation block 6-221
QASK Demap Square Constellation block 6-218
QASK Demod Arbitrary Constellation block 6-171
QASK Demod CE Arbitrary Constellation block 6-200
QASK Demod CE Circle Constellation block 6-196
QASK Demod CE Square Constellation block 6-192
QASK Demod Circle Constellation block 6-167
QASK Demod Square Constellation block 6-163
QASK Map Arbitrary Constellation block 6-222
QASK Map Circle Constellation block 6-219
QASK Map Square Constellation block 6-216
QASK Mod Arbitrary Constellation block 6-169
QASK Mod CE Arbitrary Constellation block 6-198
QASK Mod CE Circle Constellation block 6-194
QASK Mod CE Square Constellation block 6-190
QASK Mod Circle Constellation block 6-165
QASK Mod Square Constellation block 6-161
quadrature amplitude modulation (QAM) and demodulation 3-62
Quantization Decode block 6-45
quantized signal distortion 3-15

R

raised cosine filter 3-113
Raised Cosine Filter block 6-251
random error-correction code 3-36
random signal generators 3-5
Rayleigh Fading CE Channel block 6-269
Rayleigh noise 3-6
Rayleigh Noise CE Channel block 6-271
Rayleigh noise channel 3-121
Rayleigh Noise Generator block 6-14
Real-Time Workshop 1-4, A-1
  guidelines A-9
Reed-Solomon code 3-36
Reed-Solomon Decode Binary Sequence In/Out block 6-89
Reed-Solomon Decode Binary Vector In/Out block 6-80
Reed-Solomon Decode Integer Sequence In/Out block 6-84
Reed-Solomon Decode Integer Vector In/Out block 6-76
Reed-Solomon Encode Binary Vector In/Out block 6-78
Reed-Solomon Encode Integer Sequence In/Out block 6-82
Reed-Solomon Encode Integer Vector In/Out block 6-74
reference tables for MATLAB functions 5-4
Register Shift block 6-295
Rician noise 3-6
Rician Noise CE Channel block 6-273
Rician noise channel 3-121
Rician Random Noise Generator block 6-15

S

Sampled Read From Workspace block 6-22
scalar quantization 3-13
scalar to vector conversion 3-128
Scalar to Vector Converter block 6-308
scatter plot 3-11, 6-28
scheduled reset integration 3-128
Scheduled Reset Integrator block 6-293
sequential input/output format 3-27
S-functions A-9
signal generators and display devices 3-3
signal generators, random 3-5
Signal Processing Toolbox 1-3
Signal Quantizer block 6-40
signal space 3-68
simulation
  data flow 2-10
  differences between Simulink and MATLAB 2-10
  time flow 2-10
SIMULINK 1-4
Sinc block 6-257
sinc filter 3-118
single-sideband suppressed carrier amplitude modulation and demodulation 3-60
soft decision decoding 3-44
  log likelihood 3-44
  metric 3-46
source coding 2-3, 3-13
source coding reference table 6-39
sources and sinks reference table 6-6
square constellation 3-74
SSB ADM block 6-130
SSB ADM CE block 6-149
SSB-AM block 6-128
SSB-AM CE block 6-147
survivor 3-47
symbol decision. *See* demapping 3-8
symbol error rate 3-12
symbol interval 6-210
synchronization 3-123, 6-276
  baseband PLL simulation 3-125
syndrome 3-31
systematic code 3-23
systematic linear code 3-30

T

TDMA (time division multiple access) 3-107
TDMA, digital 3-107
template makefile
  modifying A-3
time division multiple access (TDMA) 3-107
time flow simulation 2-10
Time-share Demux block 6-244
Time-Share Mux block 6-242
transfer functions
  binary form 3-40
  nonlinear form 3-41
  octal form 3-40
  state space form 3-41
transfer functions convolutional code 3-38
transfer probability 3-45
transmission delay 2-15
transmitting and receiving filters 3-113
transmitting/receiving filters reference table 6-250
trellis plot 3-48
Triggered Buffer Down block 6-297
Triggered Read From Workspace block 6-18
Triggered Signal Quantizer block 6-43
Triggered Write To Workspace block 6-25

U

uniform noise 3-7
Uniform Random Integer Generator block 6-11
Uniform Random Noise Generator block 6-8
utilities 3-126
utility reference table 6-287

V

Varying AWGN Channel block 6-265
Varying Rayleigh Fading CE Channel block 6-270
Varying Rayleigh Noise Channel CE block 6-272
Varying Rician Noise Channel CE block 6-274
VCO (voltage controlled oscillator) 3-127
VCO block 6-301
vector input/output format 3-26
vector partition 3-14
Vector Pulse block 6-23
Vector Redistributor block 6-246
Vector to Scalar Converter block 6-309
vector vs. sequential input/output 3-25
Viterbi algorithm 3-47
Viterbi method 3-44
    hard decision decoding 3-44
    soft decision decoding 3-44
voltage controlled oscillator (VCO) 3-127

W

windowed integration 3-128
Windowed Integrator block 6-292