Communications Toolbox

For Use with MATLAB®

Computation

Visualization

Programming

User's Guide

Version 2

How to Contact The MathWorks:

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

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

508-647-7000 Phone

508-647-7001 Fax

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

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

Communications Toolbox User's Guide

Printing History:

April 1996	First printing	New
May 1997	Second printing	Revised for MATLAB 5
September 2000	Third printing	Revised for Version 2 (Release 12)
May 2001	Online only	Revised for Version 2.0.1 (Release 12.1)
July 2002	Fourth printing	Revised for Version 2.1 (Release 13)

Contents

Preface

1 A Detailed Example

2 Using the Communications Toolbox

3 Function Reference

A Appendix: Galois Fields of Odd Characteristic

Preface

This chapter provides a brief overview of the Communications Toolbox, as well as information about this documentation set. The sections are as follows.

What Is the Communications Toolbox?

The Communications Toolbox is a set of MATLAB® functions that can help you design and analyze advanced communication systems. Functions in the toolbox can accomplish these tasks:

- Random signal production
- Error analysis, including eye diagrams and scatter plots
- Source coding, including scalar quantization, differential pulse code modulation, and companders
- Error-control coding, including convolutional and linear block coding
- Analog and digital modulation/demodulation
- Filtering of data using special filters
- Computations in Galois fields

Related Products

The MathWorks provides several products that are especially relevant to the kinds of tasks you can perform with the Communications Toolbox. They are listed in the table below. In particular, the Communications Toolbox *requires* these products:

- MATLAB
- Signal Processing Toolbox

For more information about any of these products, see either

- The online documentation for that product if it is installed or if you are reading the documentation from the CD
- The MathWorks Web site, at `http://www.mathworks.com`; see the "products" section

The toolboxes listed below all include functions that extend the capabilities of MATLAB. The blocksets all include blocks that extend the capabilities of Simulink.®

Product	Description
CDMA Reference Blockset	Design and simulate IS-95A mobile phone equipment
Communications Blockset	Design and simulate communication systems
DSP Blockset	Design and simulate DSP systems
Signal Processing Toolbox	Perform signal processing, analysis, and algorithm development
Simulink	Design and simulate continuous- and discrete-time systems

Using This Guide

This guide describes and illustrates the capabilities of the Communications Toolbox. The table below matches sections of this guide with your possible learning goals.

Goal	Section
Examine an example in detail, to begin learning about the toolbox	"A Detailed Example" on page 1-1
Learn how this toolbox implements a particular category of functionality, such as source coding	"Using the Communications Toolbox" on page 2-1
Learn about particular functions in this toolbox	Online function reference

Expected Background

This guide assumes that you already have background knowledge in the subject of communications. If you do not yet have this background, then you can acquire it using a standard communications text or the books listed in one of this guide's sections titled "Selected Bibliography for... ."

For New Users

Start with "A Detailed Example", which describes an example in detail. Then read those parts of "Using the Communications Toolbox" that address the functionality that concerns you. When you find out from that chapter which functions you want to use, refer to the online references pages that describe those functions.

For Experienced Users

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

You might also want to browse through "A Detailed Example" and "Using the Communications Toolbox" based on your interests or needs.

Supplementing This Guide with Command-Line Help

Command-line help is text that MATLAB displays in its Command Window. The table below lists two kinds of command-line help that are available for the Communications Toolbox, along with the command that you type at the MATLAB prompt in order to display the help text.

Type of Command-Line Help	MATLAB Command
List of functions in the Communications Toolbox	`help comm`
Information about a particular function	`help function` (for example, `help ademod`)

Method-Specific Help

Some multipurpose functions also provide command-line help on specific methods. For example, `help encode` displays text that describes the use of the `encode` command for error-control encoding. One specific method of error-control encoding is BCH encoding. The command

```
encode bch
```

displays text that describes the use of the `encode` command for BCH encoding. The functions that provide method-specific help are: `amod`, `ademod`, `amodce`, `ademodce`, `ddemod`, `ddemodce`, `decode`, `demodmap`, `dmod`, `dmodce`, `encode`, and `modmap`. The general help text, displayed by the `help function` command, lists the available methods.

Configuration Information

To determine if the Communications Toolbox is installed on your system, type

```
ver
```

at the MATLAB prompt. MATLAB displays information about the version of MATLAB you are running, including a list of all toolboxes installed on your system and their version numbers. Check the list to see if the Communications Toolbox appears.

For information about installing the toolbox, see the *MATLAB Installation Guide* for your platform.

Technical Conventions

This section mentions some technical conventions that this guide uses.

Polynomials as Vectors

MATLAB represents a polynomial in one variable x using a vector that lists the polynomial's coefficients, arranged according to the powers of x. *Descending order* means that the coefficient of the highest power of x appears first and that the polynomial's constant term appears last. *Ascending order* is the opposite. The table below illustrates the conventions for functions in this toolbox and for built-in MATLAB functions.

Category of Functions	Vector That Represents the Polynomial $1+2x+3x^2$
Error-control coding using Hamming, BCH, cyclic, or generic linear block codes	[1, 2, 3] (ascending order)
Computations in Galois fields of odd characteristic	
Other functions in the Communications Toolbox	[3, 2, 1] (descending order)
Functions in MATLAB, such as `roots` and `polyval`	

Matrices

Matrix dimensions are described by listing the number of rows and the number of columns of the matrix in that order, as below.

```
u = [1 2 3;4 5 6] % A 2-by-3 matrix
```

Typographical Conventions

This manual uses some or all of these conventions.

Item	Convention	Example
Example code	`Monospace` font	To assign the value 5 to `A`, enter `A = 5`
Function names, syntax, filenames, directory/folder names, and user input	`Monospace` font	The `cos` function finds the cosine of each array element. Syntax line example is `MLGetVar ML_var_name`
Buttons and keys	**Boldface** with book title caps	Press the **Enter** key.
Literal strings (in syntax descriptions in reference chapters)	**`Monospace bold`** for literals	`f = freqspace(n,'whole')`
Mathematical expressions	*Italics* for variables Standard text font for functions, operators, and constants	This vector represents the polynomial $p = x^2 + 2x + 3$.

MATLAB output	`Monospace` font	MATLAB responds with
		`A =`
		`5`

Menu and dialog box titles	**Boldface** with book title caps	Choose the **File Options** menu.
New terms and for emphasis	*Italics*	An *array* is an ordered collection of information.
Omitted input arguments	(...) ellipsis denotes all of the input/output arguments from preceding syntaxes.	`[c,ia,ib] = union(...)`
String variables (from a finite list)	*`Monospace italics`*	`sysc = d2c(sysd,'method')`

1

A Detailed Example

This chapter describes a particular example in detail, to help you get started using the Communications Toolbox. This chapter assumes very little about your prior knowledge of MATLAB, although it still assumes that you have a basic knowledge about communications subject matter.

The topics here are as follows.

What the Example Does

The example creates a random digital signal consisting of integers between 0 and 8, and modulates it using two varieties of the 8-ary quadrature amplitude shift keying (QASK) technique. This technique associates each integer in the signal with some point in an eight-point signal constellation, and then uses the associations to create a modulated signal.

There are 8!, that is, `factorial(8)`, ways to associate eight symbols with eight constellation points. One category of configurations implements what is called Gray coding. In a Gray coded constellation, the symbol associated with a given point and the symbol of any of the point's nearest neighbors differ in exactly one bit. Thus, the constellation point associated with the symbol 3 (= 011) can have as a nearest neighbor the point associated with the symbol 1 (= 001), 2 (= 010), or 7 (= 111), but not any other number.

In order to compare the behavior of different constellation configurations, the example modulates the message signal separately using two varieties of 8-QASK modulation. Both varieties use constellations with the same points, but one variety labels the constellation points so as to implement Gray code while the other variety does not implement Gray code. After modulating, the example adds noise to both modulated signals, demodulates both noisy signals, and compares the bit error rates in the two cases.

The example outputs the two bit error rates. The expectation is that although noise might cause demodulation errors in both cases, the errors in the Gray coding case should involve fewer bits. When you execute the example, check to see whether the bit error rate from the Gray coding case is smaller than the bit error rate from the non-Gray coding case.

Functions in the Example

The example uses several functions from the toolbox, as the table below indicates.

Function	Purpose in Example
`randint`	Generate a random signal
`dmodce`	Modulate signals
`ddemodce`	Demodulate signals
`biterr`	Compute bit error rate
`modmap`	Plot a signal constellation

Where to Find the Example

If you have already installed MATLAB and the Communications Toolbox, then the toolbox will be there whenever you start up MATLAB. The example is contained in a file called `commgettingstarted.m`, which is located in the `toolbox/comm/commdemos` directory within your MATLAB installation. You can view the contents of the example file by typing

```
type commgettingstarted
```

at the MATLAB prompt.

You can execute the example by typing

```
commgettingstarted
```

at the MATLAB prompt.

How the Example Works

These sections display and explain portions of the example code:

Setting Up Parameters

The first part of the example defines variables that the rest of the example uses. The symbol alphabet has `M` different symbols, namely, the integers between 0 and `M-1`. The message is a column vector having `len` entries, each of which is chosen from the symbol alphabet.

The variables `Fd` and `Fs` refer to the relative sampling rates for the modulation scheme. They would be more meaningful if the example were sampling a real signal that had a natural notion of time. However, because this example uses a random signal that does not have a built-in notion of time, the main purpose of `Fd` and `Fs` is to indicate that the modulated signal has three entries for every one entry of the original signal.

```
% Set up parameters.
M = 8; % Number of symbols in alphabet
len = 10000; % Number of symbols in the original message
Fd = 1; % Assume the original message is sampled
% at a rate of 1 sample per second.
Fs = 3; % The modulated signal will be sampled
% at a rate of 3 samples per second.
```

Creating the Signal

The variable `signal` is a `len`-by-1 matrix, that is, a column vector of length `len`, whose entries are randomly chosen integers between 0 and `M-1`. This is the signal that the example will modulate. The `randint` function is part of this toolbox.

```
% Create a signal.
signal = randint(len,1,M); % Random digital message
% consisting of integers between 0 and M-1
```

Modulating the Signal

This part of the example modulates the data in the column vector `signal` in two different ways. The `dmodce` function performs both modulations and puts the results into the two-column matrix `modsignal`.

The first call to `dmodce`, which creates the first column of `modsignal`, tells `dmodce` to use QASK modulation on `M` symbols. The string `'qask'` indicates the QASK method as well as the default square constellation configuration. In this case, the configuration implements Gray code.

The second call to `dmodce`, which creates the second column of `modsignal`, tells `dmodce` to use QASK modulation with a signal constellation whose configuration is represented in the vectors `inphase` and `quadr`. The variables `inphase` and `quadr` are length-`M` vectors that list the in-phase and quadrature components, respectively, of the points in the signal constellation. The points are listed in sequence, to associate a message symbol of k with the (k+1)st elements in `inphase` and `quadr`. Whereas Gray code labels the constellation points in a special way, this configuration lists points in a sequence that is merely convenient for creating `inphase` and `quadr`.

These lines also illustrate some common ways to manipulate matrices in MATLAB. If you are not familiar with the colon notation in MATLAB or with functions like `ones` and `zeros`, then you should consult the MATLAB documentation set.

```
% Use M-ary QASK modulation with two different labeled
% square constellations.
modsignal(:,1) = dmodce(signal,Fd,Fs,'qask',M);
inphase = [-3:2:3 -3:2:3];
quadr = [ones(1,4), -1*ones(1,4)];
modsignal(:,2) = dmodce(signal,Fd,Fs,'qask/arb',inphase,quadr);
```

Adding Noise

According to the definition of baseband QASK modulation, `modsignal` is a *complex* matrix having `len*Fs/Fd` rows and two columns. The command below adds normally distributed random numbers to the real and imaginary parts of

modsignal, to produce a noisy signal noisy. The randn function is a built-in MATLAB function.

Notice that the command adds to modsignal an entire real matrix of the appropriate size and an entire imaginary matrix of the appropriate size. Using a loop to add noise to individual scalar entries of modsignal would be less efficient, because MATLAB is optimized for matrix operations.

```
% Add noise to real and imaginary parts of the modulated signal.
noisy = modsignal+.5*randn(len*Fs/Fd,2)...
+j*.5*randn(len*Fs/Fd,2);
```

Demodulating the Signal

This part of the example demodulates the noisy modulated signal, noisy, in two different ways. The ddemodce function performs both demodulations by operating on each column of noisy separately. In each case, ddemodce puts the results into the two-column matrix newsignal.

```
% Demodulate to recover the message.
newsignal(:,1) = ddemodce(noisy(:,1),Fd,Fs,'qask',M);
newsignal(:,2) = ddemodce(noisy(:,2),Fd,Fs,...
'qask/arb',inphase,quadr);
```

Computing and Displaying Bit Error Rates

The biterr function compares each demodulated signal (that is, each column of newsignal) to the original signal. Then biterr computes the number of bit errors, as well as the rate or fraction of bit errors. The built-in MATLAB function disp displays the two bit error rates in the MATLAB Command Window.

```
% Check whether Gray code resulted in fewer bit errors.
% Compare signal with each column of newsignal.
[num,rate] = biterr(newsignal,signal);
disp('Bit error rates for the two constellations used here')
disp('-------------------------------------------------')
disp(['Gray code constellation:     ', num2str(rate(1))])
disp(['Non-Gray code constellation: ', num2str(rate(2))])
```

Plotting a Signal Constellation

The `modmap` function plots and labels the default square signal constellation having `M` points. The constellation that `inphase` and `quadr` determine looks the same, except that the points are labeled from left to right across each row in the diagram, starting with the upper row.

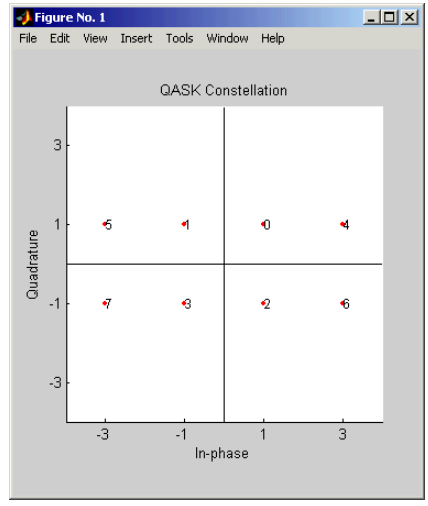
```
% Plot signal constellations with Gray code labeling.
modmap('qask',M);
```

Output from the Example

The example produces output in the MATLAB Command Window like that shown below. Because the message signal and the noise are random, you will probably not get the exact numbers below. (For information about states and repeatable sequences of random numbers, see the reference page for the built-in MATLAB function `rand`.)

```
Bit error rates for the two constellations used here
----------------------------------------------------
Gray code constellation:     0.0003
Non-Gray code constellation: 0.00036667
```

The example also produces a figure window containing the signal constellation plot in the figure below. The horizontal axis represents the in-phase components and the vertical axis represents the quadrature components. The dots are the constellation points. The number next to each dot is the message symbol associated with that dot. By considering the binary form of each number from 0 to `M-1`, you can check that this constellation implements Gray code.



2

Using the Communications Toolbox

A typical communication system includes a signal source, sink, and channel, as well as processes for transmitting and receiving. This chapter describes and illustrates how to implement communication components using the functions provided in the Communications Toolbox. Each section in this chapter corresponds to a category of functionality within the Communications Toolbox. The sections are as follows.

Random Signals and Error Analysis

Simulating a communication system often involves analyzing its response to the noise inherent in real-world components. Such analysis aims to illustrate the system's response and possibly to help design a system appropriate for the most likely kinds of noise.

Error Analysis Features of the Toolbox

Error analysis tasks supported in the Communications Toolbox include

- Simulating noise or signal sources using random signals
- Computing the error rate or number of errors
- Plotting an eye diagram
- Generating a scatter plot

This section describes these toolbox functions that accomplish error-analysis tasks: `biterr`, `eyediagram`, `randerr`, `randint`, `randsrc`, `scatterplot`, `symerr`, and `wgn`. Because error analysis is often a component of communication system simulation, other portions of this guide provide additional examples.

Random Signals

Random signals are useful for simulating noise, errors, or signal sources. Besides built-in MATLAB functions like `rand` and `randn`, you can also use these functions from this toolbox:

- `wgn`, for generating white Gaussian noise
- `randsrc`, for generating random symbols
- `randint`, for generating uniformly distributed random integers
- `randerr`, for generating random bit error patterns

While `randsrc` and `randint` are suitable for representing sources, `randerr` is more appropriate for modeling channel errors.

White Gaussian Noise

The `wgn` function generates random matrices using a white Gaussian noise distribution. You specify the power of the noise in either dBW (decibels relative to a watt), dBm, or linear units. You can generate either real or complex noise.

For example, the command below generates a column vector of length 50 containing real white Gaussian noise whose power is 2 dBW. The function assumes that the load impedance is 1 ohm.

```
y1 = wgn(50,1,2);
```

To generate complex white Gaussian noise whose power is 2 Watts, across a load of 60 ohms, use either of the commands below. Notice that the ordering of the string inputs does not matter.

```
y2 = wgn(50,1,2,60,'complex','linear');
y3 = wgn(50,1,2,60,'linear','complex');
```

To send a signal through an additive white Gaussian noise channel, use the `awgn` function.

Random Symbol Matrices

The `randsrc` function generates random matrices whose entries are chosen independently from an alphabet that you specify, with a distribution that you specify. A special case generates bipolar matrices.

For example, the command below generates a 5-by-4 matrix whose entries are independently chosen and uniformly distributed in the set {1,3,5}. (Your results might vary because these are random numbers.)

```
a = randsrc(5,4,[1,3,5])

a =

     3     5     1     5
     1     5     3     3
     1     3     3     1
     1     1     3     5
     3     1     1     3
```

If you want 1 to be twice as likely to occur as either 3 or 5, then use the command below to prescribe the skewed distribution. Notice that the third input argument has two rows, one of which indicates the possible values of `b` and the other indicates the probability of each value.

```
b = randsrc(5,4,[1,3,5; .5,.25,.25])
```

```
b =

     3     3     5     1
     1     1     1     1
     1     5     1     1
     1     3     1     3
     3     1     3     1
```

Random Integer Matrices

The `randint` function generates random integer matrices whose entries are in a range that you specify. A special case generates random binary matrices.

For example, the command below generates a 5-by-4 matrix containing random integers between 2 and 10.

```
c = randint(5,4,[2,10])

c =

     2     4     4     6
     4     5    10     5
     9     7    10     8
     5     5     2     3
    10     3     4    10
```

If your desired range is [0,10] instead of [2,10] then you can use either of the commands below. They produce different numerical results, but use the same distribution.

```
d = randint(5,4,[0,10]);
e = randint(5,4,11);
```

Random Bit Error Patterns

The `randerr` function generates matrices whose entries are either 0 or 1. However, its options are rather different from those of `randint`, because `randerr` is meant for testing error-control coding. For example, the command below generates a 5-by-4 binary matrix having the property that each row contains exactly one 1.

```
f = randerr(5,4)
```

```
f =

     0     0     1     0
     0     0     1     0
     0     1     0     0
     1     0     0     0
     0     0     1     0
```

You might use such a command to perturb a binary code that consists of five four-bit codewords. Adding the random matrix `f` to your code matrix (modulo 2) would introduce exactly one error into each codeword.

On the other hand, if you want to perturb each codeword by introducing one error with probability 0.4 and two errors with probability 0.6, then the command below should replace the one above.

```
% Each row has one '1' with probability 0.4, otherwise two '1's
g = randerr(5,4,[1,2; 0.4,0.6])

g =

     0     1     1     0
     0     1     0     0
     0     0     1     1
     1     0     1     0
     0     1     1     0
```

Note The probability matrix that is the third argument of `randerr` affects only the *number* of 1s in each row, not their placement.

As another application, you can generate an equiprobable binary 100-element column vector using any of the commands below. The three commands produce different numerical outputs, but use the same *distribution*. Notice that the third input arguments vary according to each function's particular way of specifying its behavior.

```
binarymatrix1 = randsrc(100,1,[0 1]); % Possible values are 0,1.
binarymatrix2 = randint(100,1,2); % Two possible values
binarymatrix3 = randerr(100,1,[0 1;.5 .5]); % No 1s, or one 1
```

Error Rates

Comparing messages before and after transmission can help you evaluate the quality of a communication system design or the performance of a special technique or algorithm. If your communication system uses several bits to represent a single symbol, then counting bit errors is different from counting symbol errors. In either the bit- or symbol-counting case, the error rate is the number of errors divided by the total number (of bits or symbols) transmitted.

The `biterr` function compares two messages and computes the number of bit errors and the bit error rate. The `symerr` function compares two messages and computes the number of symbol errors and the symbol error rate.

Example: Computing Error Rates

The script below uses the `symerr` function to compute the symbol error rates for a noisy linear block code. After artificially adding noise to the encoded message, it compares the resulting noisy code to the original code. Then it decodes and compares the decoded message to the original one.

```
m = 3; n = 2^m-1; k = n-m; % Prepare to use Hamming code.
msg = randint(k*200,1,2); % 200 messages of k bits each
code = encode(msg,n,k,'hamming');
codenoisy = rem(code+(rand(n*200,1)>.95),2); % Add noise.
% Decode and correct some errors.
newmsg = decode(codenoisy,n,k,'hamming');
% Compute and display symbol error rates.
[codenum,coderate] = symerr(code,codenoisy);
[msgnum,msgrate] = symerr(msg,newmsg);
disp(['Error rate in the received code: ',num2str(coderate)])
disp(['Error rate after decoding: ',num2str(msgrate)])
```

The output is below. The error rate decreases after decoding because the Hamming decoder corrects some of the errors. Your results might vary because the example uses random numbers.

```
Error rate in the received code: 0.054286
Error rate after decoding: 0.03
```

Comparison of Symbol Error Rate and Bit Error Rate

In the example above, the symbol errors and bit errors are the same because each symbol is a bit. The commands below illustrate the difference between symbol errors and bit errors in other situations.

```
a = [1 2 3]'; b = [1 4 4]';
format rat % Display fractions instead of decimals.
[snum,srate] = symerr(a,b)

snum =

      2

srate =

     2/3

[bnum,brate] = biterr(a,b)

bnum =

     5

brate =

     5/9
```

`bnum` is 5 because the second entries differ in two bits and the third entries differ in three bits. `brate` is 5/9 because the total number of bits is nine. The total number of bits is, by definition, the number of entries in `a` or `b` times the maximum number of bits among all entries of `a` and `b`.

Eye Diagrams

An eye diagram is a simple and convenient tool for studying the effects of intersymbol interference and other channel impairments in digital transmission. To construct an eye diagram, plot the received signal against time on a fixed-interval axis. At the end of the fixed time interval, wrap around to the beginning of the time axis. Thus the diagram consists of many overlapping curves. One way to use an eye diagram is to look for the place

where the “eye” is most widely opened, and use that point as the decision point when demapping a demodulated signal to recover a digital message.

To produce an eye diagram from a signal, use the `eyediagram` function. The signal can have different formats, as the table below indicates.

Representing In-Phase and Quadrature Components of Signal

Signal Format	Source of In-Phase Components	Source of Quadrature Components
Real matrix with two columns	First column	Second column
Complex vector	Real part	Imaginary part
Real vector	Vector contents	Quadrature component is always zero

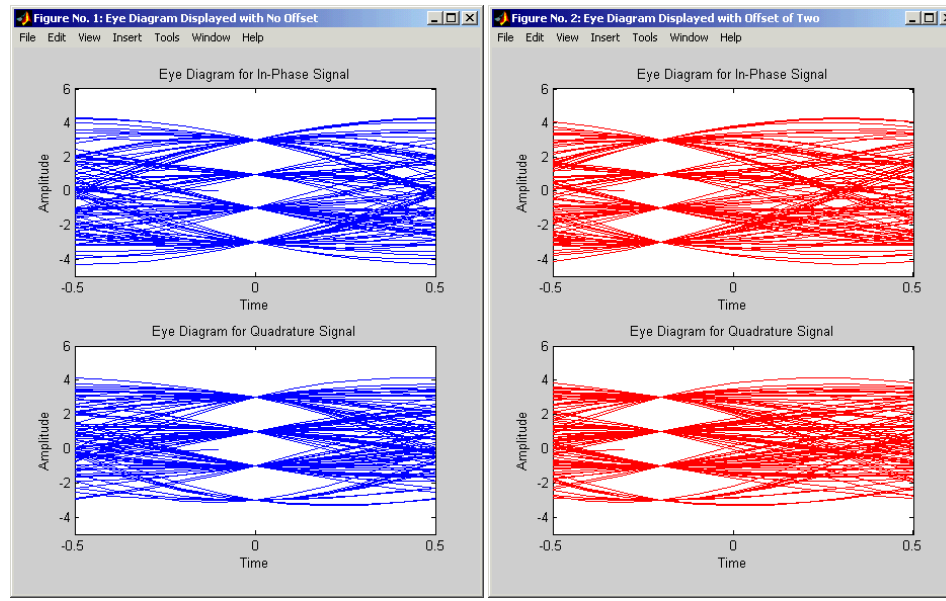
Example: Eye Diagrams

The code below illustrates the use of the eye diagram for finding the best decision point. It maps a random digital signal to a 16-QASK waveform, then uses a raised cosine filter to simulate a noisy transmission channel. Several commands manipulate the filtered data to isolate its steady-state behavior. Then the `eyediagram` command produces an eye diagram from the resulting signal.

```
% Define the M-ary number and sampling rates.
M = 16; Fd = 1; Fs = 10;
Pd = 100; % Number of points in the calculation
msg_d = randint(Pd,1,M); % Random integers in the range [0,M-1]
% Modulate using square constellation QASK method.
msg_a = modmap(msg_d,Fd,Fd,'qask',M);
% Assume the channel is equivalent to a raised cosine filter.
delay = 3; % Delay of the raised cosine filter
rcv = rcosflt(msg_a,Fd,Fs,'fir/normal',.5,delay);

% Truncate the output of rcosflt to remove response tails.
propdelay = delay .* Fs/Fd + 1; % Propagation delay of filter
```

```
rcv1 = rcv(propdelay:end-(propdelay-1),:); % Truncated version
N = Fs/Fd;

% Plot the eye diagram of the resulting signal sampled and
% displayed with no offset.
offset1 = 0;
h1 =  eyediagram(rcv1,N,1/Fd,offset1);
set(h1,'Name','Eye Diagram Displayed with No Offset');
```

Notice that a vertical line down the center of the diagram would cross the “eye” at its most widely opened point, as in the left-hand side below.



In the right-hand diagram above, a similar vertical line would *not* cross the eye at the most widely opened point. This diagram results from the commands

```
offset2 = 2;
h2 = eyediagram(rcv1,N,1/Fd,offset2,'r-');
set(h2,'Name','Eye Diagram Displayed with Offset of Two');
```

This example continues by using the information gathered from the eye diagrams to choose the decision-timing offset in the `demodmap` command.

(Notice that the actual offset value in `demodmap` is `offset1+1` because `eyediagram` and `demodmap` express offsets in a different way.)

```
% Continue, using the offset information for digital demapping.
newmsg1 = demodmap(rcv1,[Fd offset1+1],Fs,'qask',16);
s1 = symerr(msg_d,newmsg1) % Number of symbol errors
```

The output is

```
s1 =

     0
```

By contrast, an offset value based on `offset2` leads to errors in the recovered digital signal. Your exact number of errors might vary because the message `msg_d` consists of random numbers.

```
newmsg2 = demodmap(rcv1,[Fd offset2+1],Fs,'qask',16);
s2 = symerr(msg_d,newmsg2)
```

The output is

```
s2 =

    8
```

As an additional example of using the `eyediagram` function, the commands below display the eye diagram with no offset, but based on data that is sampled with an offset of two samples. This sampling offset simulates errors in timing that result from being two samples away from perfect synchronization.

```
h3 = eyediagram(rcv1(1+offset2:end,:),N,1/Fd,0);
set(h3,'Name','Eye Diagram Sampled with Offset of Two');
```

Scatter Plots

A scatter plot of a signal shows the signal's value at a given decision point. In the best case, the decision point should be at the time when the eye of the signal's eye diagram is the most widely open.

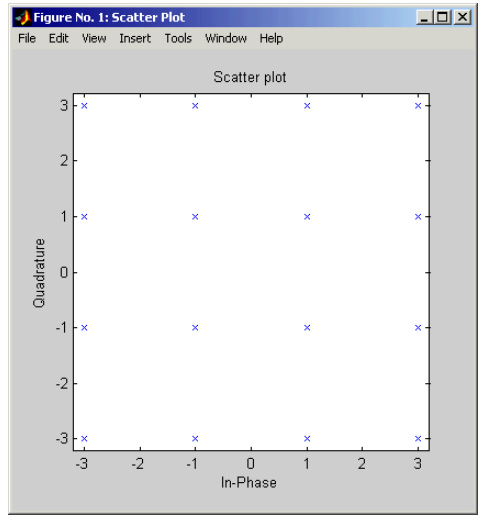
To produce a scatter plot from a signal, use the `scatterplot` function. The signal can have different formats, as in the case of the `eyediagram` function. See the table "Representing In-Phase and Quadrature Components of Signal" on page 2-8 for details.

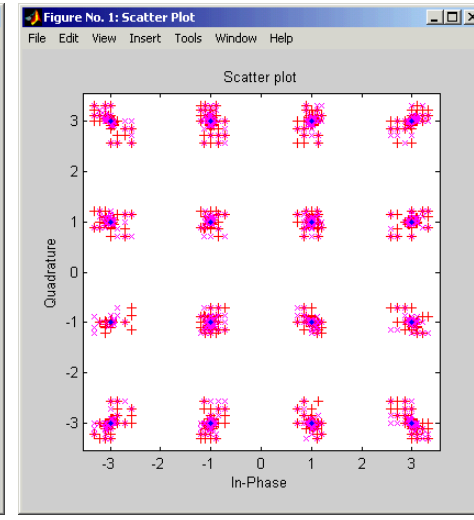
Example: Scatter Plots

The code below is similar to the example from the section "Example: Eye Diagrams" on page 2-8. It produces a scatter plot from the received analog signal, instead of an eye diagram.

```
% Define the M-ary number and sampling rates.
M = 16; Fd = 1; Fs = 10;
Pd = 200; % Number of points in the calculation
msg_d = randint(Pd,1,M); % Random integers in the range [0,M-1]
% Modulate using square constellation QASK method.
msg_a = modmap(msg_d,Fd,Fs,'qask',M);
% Assume the channel is equivalent to a raised cosine filter.
rcv = rcosflt(msg_a,Fd,Fs);
% Create the scatter plot of the received signal,
% ignoring the first three and the last four symbols.
N = Fs/Fd;
rcv_a = rcv(3*N+1:end-4*N,:);
h = scatterplot(rcv_a,N,0,'bx');
```

Varying the third parameter in the `scatterplot` command changes the offset. An offset of zero yields optimal results, shown on the left below.





The diagram on the right results from the commands below. The x's and +'s reflect two offsets that are not optimal because they are too late and too early, respectively. Notice that in the diagram, the dots are the actual constellation points, while the other symbols are perturbations of those points.

```
hold on;
scatterplot(rcv_a,N,N+1,'r+',h); % Plot +'s
scatterplot(rcv_a,N,N-1,'mx',h); % Plot x's
scatterplot(rcv_a,N,0,'b.',h); % Plot dots
```

Source Coding

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

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

Source Coding Features of the Toolbox

This toolbox supports scalar quantization, predictive quantization, and arithmetic coding. It does not support vector quantization. Functions in the toolbox can accomplish these tasks:

- Quantize a signal according to a partition and codebook that you specify
- Optimize partition and codebook parameters for a set of training data
- Encode or decode a signal using the differential pulse code modulation (DPCM) technique
- Optimize DPCM parameters for a set of training data
- Perform μ-law or A-law compressor or expander calculations
- Perform arithmetic coding and decoding

Representing Quantization Parameters

Scalar quantization is a process that maps all inputs within a specified range to a common value. It maps inputs in a different range of values to a different common value. In effect, scalar quantization digitizes an analog signal. Two parameters determine a quantization: a partition and a codebook. This section describes how toolbox functions represent these parameters.

Partitions

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

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

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

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

```
partition = [0,1,3];
```

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

Codebooks

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

```
codebook = [-1, 0.5, 2, 3];
```

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

Quantizing a Signal

The previous section described how you can represent the partition and codebook that determine your scalar quantization process. This section shows how to use these parameters in the `quantiz` function.

Scalar Quantization Example 1

The code below shows how the `quantiz` function uses `partition` and `codebook` to map a real vector, `samp`, to a new vector, `quantized`, whose entries are either -1, 0.5, 2, or 3.

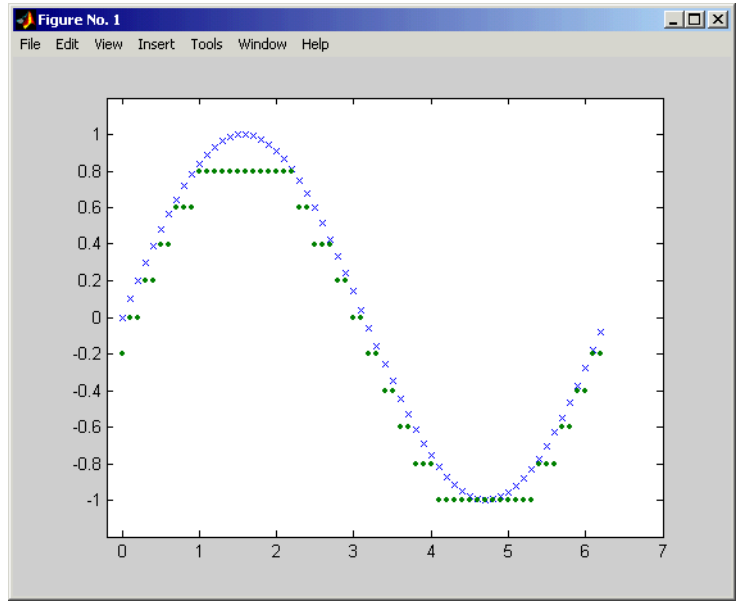
```
partition = [0,1,3];
codebook = [-1, 0.5, 2, 3];
samp = [-2.4, -1, -.2, 0, .2, 1, 1.2, 1.9, 2, 2.9, 3, 3.5, 5];
[index,quantized] = quantiz(samp,partition,codebook);
quantized
```

```
quantized =

  Columns 1 through 6

   -1.0000   -1.0000   -1.0000   -1.0000    0.5000    0.5000

  Columns 7 through 12

    2.0000    2.0000    2.0000    2.0000    2.0000    3.0000

  Column 13

    3.0000
```

Scalar Quantization Example 2

This example illustrates the nature of scalar quantization more clearly. After quantizing a sampled sine wave, it plots the original and quantized signals. The plot contrasts the *x*'s that make up the sine curve with the dots that make up the quantized signal. The vertical coordinate of each dot is a value in the vector `codebook`.

```
t = [0:.1:2*pi]; % Times at which to sample the sine function
sig = sin(t); % Original signal, a sine wave
partition = [-1:.2:1]; % Length 11, to represent 12 intervals
codebook = [-1.2:.2:1]; % Length 12, one entry for each interval
[index,quants] = quantiz(sig,partition,codebook); % Quantize.
plot(t,sig,'x',t,quants,'.')
axis([-.2 7 -1.2 1.2])
```



Determining Which Interval Each Input Is In

The `quantiz` function also returns a vector that tells which interval each input is in. For example, the output below says that the input entries lie within the intervals labeled 0, 6, and 5, respectively. Here, the 0th interval consists of real numbers less than or equal to 3; the 6th interval consists of real numbers greater than 8 but less than or equal to 9; and the 5th interval consists of real numbers greater than 7 but less than or equal to 8.

```
partition = [3,4,5,6,7,8,9];
index = quantiz([2 9 8],partition)
index =

     0
     6
     5
```

If you continue this example by defining a codebook vector such as

```
codebook = [3,3,4,5,6,7,8,9];
```

then the equation below relates the vector `index` to the quantized signal `quants`.

```
quants = codebook(index+1);
```

This formula for `quants` is exactly what the `quantiz` function uses if you instead phrase the example more concisely as below.

```
partition = [3,4,5,6,7,8,9];
codebook = [3,3,4,5,6,7,8,9];
[index,quants] = quantiz([2 9 8],partition,codebook);
```

Optimizing Quantization Parameters

Quantization distorts a signal. You can lessen the distortion by choosing appropriate partition and codebook parameters. However, testing and selecting parameters for large signal sets with a fine quantization scheme can be tedious. One way to produce partition and codebook parameters easily is to optimize them according to a set of so-called *training data*.

Note The training data that you use should be typical of the kinds of signals that you will actually be quantizing.

Example: Optimizing Scalar Quantization Parameters

The `lloyds` function optimizes the partition and codebook according to the Lloyd algorithm. The code below optimizes the partition and codebook for one period of a sinusoidal signal, starting from a rough initial guess. Then it uses these parameters to quantize the original signal using the initial guess parameters as well as the optimized parameters. The output shows that the mean square distortion after quantizing is much less for the optimized parameters. Notice that the `quantiz` function automatically computes the mean square distortion and returns it as the third output parameter.

```
% Start with the setup from 2nd example in "Quantizing a Signal."
t = [0:.1:2*pi];
sig = sin(t);
partition = [-1:.2:1];
codebook = [-1.2:.2:1];
% Now optimize, using codebook as an initial guess.
```

```
[partition2,codebook2] = lloyds(sig,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
[index2,quant2,distor2] = quantiz(sig,partition2,codebook2);
% Compare mean square distortions from initial and optimized
[distor, distor2] % parameters.

ans =

    0.0148    0.0024
```

Implementing Differential Pulse Code Modulation

The quantization in the section "Quantizing a Signal" on page 2-14 requires no *a priori* knowledge about the transmitted signal. In practice, you can often make educated guesses about the present signal based on past signal transmissions. Using such educated guesses to help quantize a signal is known as *predictive quantization*. The most common predictive quantization method is differential pulse code modulation (DPCM).

The functions `dpcmenco`, `dpcmdeco`, and `dpcmopt` can help you implement a DPCM predictive quantizer with a linear predictor.

DPCM Terminology

To determine an encoder for such a quantizer, you must supply not only a partition and codebook as described in "Representing Quantization Parameters" on page 2-13, but also a *predictor*. The predictor is a function that the DPCM encoder uses to produce the educated guess at each step. A linear predictor has the form

$$y(k) = p(1)x(k\text{-}1) + p(2)x(k\text{-}2) + \ldots + p(m\text{-}1)x(k\text{-}m\text{+}1) + p(m)x(k\text{-}m)$$

where x is the original signal, $y(k)$ attempts to predict the value of $x(k)$, and p is an m-tuple of real numbers. Instead of quantizing x itself, the DPCM encoder quantizes the *predictive error*, `x-y`. The integer m above is called the *predictive order*. The special case when $m = 1$ is called *delta modulation*.

Representing Predictors

If the guess for the `k`th value of the signal `x`, based on earlier values of `x`, is

```
y(k) = p(1)x(k-1) + p(2)x(k-2) +...+ p(m-1)x(k-m+1) + p(m)x(k-m)
```

then the corresponding predictor vector for toolbox functions is

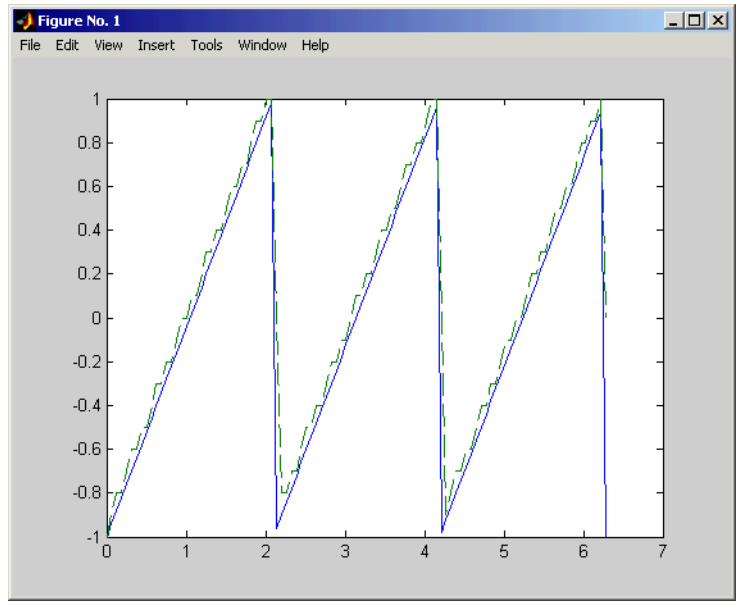
```
predictor = [0, p(1), p(2), p(3),..., p(m-1), p(m)]
```

Note The initial zero in the predictor vector makes sense if you view the vector as the polynomial transfer function of a finite impulse response (FIR) filter.

Example: DPCM Encoding and Decoding

A simple special case of DPCM quantizes the difference between the signal's current value and its value at the previous step. Thus the predictor is just $y(k) = x(k - 1)$. The code below implements this scheme. It encodes a sawtooth signal, decodes it, and plots both the original and decoded signals. The solid line is the original signal, while the dashed line is the recovered signals. The example also computes the mean square error between the original and decoded signals.

```
predictor = [0 1]; % y(k)=x(k-1)
partition = [-1:.1:.9];
codebook = [-1:.1:1];
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Original signal
% Quantize x using DPCM.
encodedx = dpcmenco(x,codebook,partition,predictor);
% Try to recover x from the modulated signal.
decodedx = dpcmdeco(encodedx,codebook,predictor);
plot(t,x,t,decodedx,'--')
distor = sum((x-decodedx).^2)/length(x) % Mean square error

distor =

    0.0327
```



Optimizing DPCM Parameters

The section "Optimizing Quantization Parameters" on page 2-17 describes how you can use training data with the `lloyds` function to help find quantization parameters that will minimize signal distortion. This section describes similar procedures for using the `dpcmopt` function in conjunction with the two functions `dpcmenco` and `dpcmdeco`, which first appear in the previous section.

Note The training data that you use with `dpcmopt` should be typical of the kinds of signals that you will actually be quantizing with `dpcmenco`.

Example: Comparing Optimized and Nonoptimized DPCM Parameters

This example is similar to the one in the last section. However, whereas the last example created `predictor`, `partition`, and `codebook` in a straightforward but haphazard way, this example uses the same codebook (now called `initcodebook`) as an initial guess for a new *optimized* codebook parameter. This example also uses the predictive order, 1, as the desired order of the new

optimized predictor. The `dpcmopt` function creates these optimized parameters, using the sawtooth signal `x` as training data. The example goes on to quantize the training data itself; in theory, the optimized parameters are suitable for quantizing other data that is similar to `x`. Notice that the mean square distortion here is much less than the distortion in the previous example.

```
t = [0:pi/50:2*pi];
x = sawtooth(3*t); % Original signal
initcodebook = [-1:.1:1]; % Initial guess at codebook
% Optimize parameters, using initial codebook and order 1.
[predictor,codebook,partition] = dpcmopt(x,1,initcodebook);
% Quantize x using DPCM.
encodedx = dpcmenco(x,codebook,partition,predictor);
% Try to recover x from the modulated signal.
decodedx = dpcmdeco(encodedx,codebook,predictor);
distor = sum((x-decodedx).^2)/length(x) % Mean square error

distor =

    0.0063
```

Companding a Signal

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

The `compand` function supports two kinds of companders: μ-law and A-law companders. Its reference page lists both compressor laws.

Example: A μ-Law Compander

The code below quantizes an exponential signal in two ways and compares the resulting mean square distortions. First, it uses the `quantiz` function with a partition consisting of length-one intervals. In the second trial, `compand` implements a μ-law compressor, `quantiz` quantizes the compressed data, and finally `compand` expands the quantized data. The output shows that the distortion is smaller for the second scheme. This is because equal-length intervals are well suited to the logarithm of `sig`, but not well suited to `sig` itself.

```
Mu = 255; % Parameter for mu-law compander
sig = -4:.1:4;
sig = exp(sig); % Exponential signal to quantize
V = max(sig);
% 1. Quantize using equal-length intervals and no compander.
[index,quants,distor] = quantiz(sig,0:floor(V),0:ceil(V));

% 2. Use same partition and codebook, but compress
% before quantizing and expand afterwards.
compsig = compand(sig,Mu,V,'mu/compressor');
[index,quants] = quantiz(compsig,0:floor(V),0:ceil(V));
newsig = compand(quants,Mu,max(quants),'mu/expander');
distor2 = sum((newsig-sig).^2)/length(sig);
[distor, distor2] % Display both mean square distortions.

ans =

    0.5348    0.0397
```

Arithmetic Coding

Arithmetic coding offers a way to compress data and can be useful for data sources having a small alphabet. The length of an arithmetic code, instead of being fixed relative to the number of symbols being encoded, depends on the statistical frequency with which the source produces each symbol from its alphabet.

The `arithenco` and `arithdeco` functions support arithmetic coding and decoding.

Note Arithmetic coding is different from Huffman coding. For long sequences from sources having skewed distributions and small alphabets, arithmetic coding compresses better than Huffman coding.

Representing Arithmetic Coding Parameters

Arithmetic coding requires statistical information about the source of the data being encoded. In particular, the `counts` input argument in the `arithenco` and `arithdeco` functions lists the frequency with which the source produces each

symbol in its alphabet. You can determine the frequencies by studying a set of test data from the source. The set of test data can have any size you choose, as long as each symbol in the alphabet has a nonzero frequency.

For example, before encoding data from a source that produces 10 x's, 10 y's, and 80 z's in a typical 100-symbol set of test data, define

```
counts = [10 10 80];
```

Alternatively, if a larger set of test data from the source contains 22 x's, 23 y's, and 185 z's, then define

```
counts = [22 23 185];
```

Example: Creating and Decoding an Arithmetic Code

The example below performs arithmetic encoding and decoding, using a source whose alphabet has three symbols.

```
seq = repmat([3 3 1 3 3 3 3 3 2 3],1,50);
counts = [10 10 80];
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq));
```

Selected Bibliography for Source Coding

[1] Cover, Thomas M., and Joy A. Thomas, *Elements of Information Theory*, New York, John Wiley & Sons, 1991.

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

[3] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

[4] Sklar, Bernard, *Digital Communications, Fundamentals and Applications*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

Block Coding

Error-control coding techniques detect and possibly correct errors that occur when messages are transmitted in a digital communication system. To accomplish this, the encoder transmits not only the information symbols, but also one or more redundant symbols. The decoder uses the redundant symbols to detect and possibly correct whatever errors occurred during transmission.

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

If you want to process binary codes, then read about these topics:

- "Block Coding Features of the Toolbox" on page 2-25
- "Block Coding Terminology" on page 2-26
- "Representing Words for Binary Block Codes" on page 2-26
- "Parameters for Binary Block Codes" on page 2-29
- "Creating and Decoding Binary Block Codes" on page 2-34
- "Performing Other Binary Block Code Tasks" on page 2-37

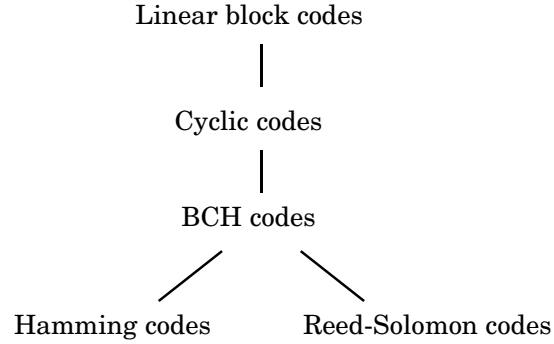
If you want to process Reed-Solomon codes, which are nonbinary, then read about these topics:

- "Block Coding Features of the Toolbox" on page 2-25
- "Block Coding Terminology" on page 2-26
- "Representing Words for Reed-Solomon Codes" on page 2-39
- "Parameters for Reed-Solomon Codes" on page 2-40
- "Creating and Decoding Reed-Solomon Codes" on page 2-41

For background information about block coding, see the works listed in "Selected Bibliography for Block Coding" on page 2-45.

Block Coding Features of the Toolbox

The class of linear block coding techniques includes categories shown below.



The Communications Toolbox supports general linear block codes. It also includes functions to process cyclic, BCH, Hamming, and Reed-Solomon codes (which are all special kinds of linear block codes). Functions in the toolbox can accomplish these tasks:

- Encode or decode a message using one of the techniques mentioned above
- Determine characteristics of a technique, such as error-correction capability or valid message length
- Perform lower level computations associated with a technique, such as
  - Compute a decoding table
  - Compute a generator or parity-check matrix
  - Convert between generator and parity-check matrices
  - Compute a generator polynomial

Note The functions in this toolbox are designed for block codes that use an alphabet whose size is a power of 2.

The table below lists the functions that are related to each supported block coding technique.

Functions Related to Block Coding Techniques

Block Coding Technique	Toolbox Functions
Linear block	`encode`, `decode`, `gen2par`, `syndtable`
Cyclic	`encode`, `decode`, `cyclpoly`, `cyclgen`, `gen2par`, `syndtable`
BCH	`encode`, `decode`, `bchenco`, `bchdeco`, `bchpoly`, `cyclgen`, `gen2par`, `syndtable`
Hamming	`encode`, `decode`, `hammgen`, `gen2par`, `syndtable`
Reed-Solomon	`rsenc`, `rsdec`, `rsgenpoly`, `rsencof`, `rsdecof`

Block Coding Terminology

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

Each block of k message symbols is encoded into a codeword that consists of n symbols; in this context, k is called the message length, n is called the codeword length, and the code is called an $[n,k]$ code.

Representing Words for Binary Block Codes

Each message or codeword is an ordered grouping of symbols. The next few subsections illustrate the various ways that these symbols can be organized or interpreted as input and output. To process binary codes, see these topics:

- "Binary Vector Format" on page 2-27
- "Binary Matrix Format" on page 2-27
- "Decimal Vector Format" on page 2-28

Binary Vector Format

For binary codes, your messages and codewords can take the form of vectors containing 0s and 1s. For example, messages and codes might look like `msg` and `code` in the lines below.

```
n = 6; k = 4; % Set codeword length and message length
% for a [6,4] code.
msg = [1 0 0 1 1 0 1 0 1 0 1 1]'; % Message is a binary column.
code = encode(msg,n,k,'cyclic'); % Code will be a binary column.
msg'

ans =

     1     0     0     1     1     0     1     0     1     0     1     1

code'

ans =

  Columns 1 through 12

     0     0     1     0     0     1     1     0     1     0     1     0

  Columns 13 through 18

     0     1     1     0     1     1
```

In this example, `msg` consists of 12 entries, which are interpreted as three 4-digit (because `k = 4`) messages. The resulting vector `code` comprises three 6-digit (because `n = 6`) codewords, which are concatenated to form a vector of length 18. The parity bits are at the beginning of each codeword.

Binary Matrix Format

For binary codes, you can organize coding information so as to emphasize the grouping of digits into messages and codewords. If you use this approach, then each message or codeword occupies a row in a binary matrix. The example below illustrates this approach by listing each 4-bit message on a distinct row in `msg` and each 6-bit codeword on a distinct row in `code`.

```
n = 6; k = 4; % Set codeword length and message length.
```

```
msg = [1 0 0 1; 1 0 1 0; 1 0 1 1]; % Message is a binary matrix.
code = encode(msg,n,k,'cyclic'); % Code will be a binary matrix.
msg

msg =

     1     0     0     1
     1     0     1     0
     1     0     1     1

code

code =

     0     0     1     0     0     1
     1     0     1     0     1     0
     0     1     1     0     1     1
```

Note In the binary matrix format, the message matrix must have `k` columns. The corresponding code matrix has `n` columns. The parity bits are at the beginning of each row.

Decimal Vector Format

For binary codes, your messages and codewords can take the form of vectors containing integers. Each element of the vector gives the decimal representation of the bits in one message or one codeword.

Note If `2^n` or `2^k` is very large, then you should use the default binary format instead of the decimal format. This is because the function uses a binary format internally, while the roundoff error associated with converting many bits to large decimal numbers and back might be substantial.

Note When you use the decimal vector format to represent binary words, MATLAB expects the *leftmost* bit to be the least significant bit.

The syntax for the `encode` command must mention the decimal format explicitly, as in the example below. Notice that `/decimal` is appended to the fourth argument in the `encode` command.

```
n = 6; k = 4; % Set codeword length and message length.
msg = [9;5;13]; % Message is a decimal column vector.
% Code will be a decimal vector.
code = encode(msg,n,k,'cyclic/decimal')

code =

    36
    21
    54
```

Note The three examples above used cyclic coding. The formats for messages and codes are similar for Hamming, generic linear, and BCH codes.

Parameters for Binary Block Codes

This subsection describes the items that you might need in order to process $[n,k]$ binary linear block codes. The table below lists the items and the coding techniques for which they are most relevant.

Parameters Used in Block Coding Techniques

Parameter	Block Coding Technique
Generator Matrix	Generic linear block
Parity-Check Matrix	Generic linear block

Parameters Used in Block Coding Techniques (Continued)

Parameter	Block Coding Technique
Generator Polynomial	Cyclic, BCH
Decoding Table	Generic linear block, Hamming

Generator Matrix

The process of encoding a message into an [n,k] linear block code is determined by a k-by-n generator matrix G. Specifically, the 1-by-k message vector v is encoded into the 1-by-n codeword vector vG. If G has the form [I_k P] or [P I_k], where P is some k-by-(n-k) matrix and I_k is the k-by-k identity matrix, then G is said to be in *standard form*. (Some authors, e.g., Clark and Cain [1], use the first standard form, while others, e.g., Lin and Costello [2], use the second.) Most functions in this toolbox assume that a generator matrix is in standard form when you use it as an input argument.

Some examples of generator matrices are in the next section, "Parity-Check Matrix."

Parity-Check Matrix

Decoding an [n,k] linear block code requires an (n-k)-by-n parity-check matrix H. It satisfies $GH^{tr} = 0 \pmod 2$, where H^{tr} denotes the matrix transpose of H, G is the code's generator matrix, and this zero matrix is k-by-(n-k). If G = [I_k P] then H = [$-P^{tr}$ I_{n-k}]. Most functions in this toolbox assume that a parity-check matrix is in standard form when you use it as an input argument.

The table below summarizes the standard forms of the generator and parity-check matrices for an [n,k] binary linear block code.

Type of Matrix	Standard Form	Dimensions
Generator	[I_k P] or [P I_k]	k-by-n
Parity-check	[$-P'$ I_{n-k}] or [I_{n-k} $-P'$]	(n-k)-by-n

I_k is the identity matrix of size k and the ' symbol indicates matrix transpose. (For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, -1 = 1 in the binary field.)

Examples. In the command below, `parmat` is a parity-check matrix and `genmat` is a generator matrix for a Hamming code in which $[n,k] = [2^3\text{-}1, n\text{-}3] = [7,4]$. Notice that `genmat` has the standard form $[P\ I_k]$.

```
[parmat,genmat] = hammgen(3)

parmat =

     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1

genmat =

     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
```

The next example finds parity-check and generator matrices for a [7,3] cyclic code. The `cyclpoly` function is mentioned below in "Generator Polynomial."

```
genpoly = cyclpoly(7,3);
[parmat,genmat] = cyclgen(7,genpoly)

parmat =

     1     0     0     0     1     1     0
     0     1     0     0     0     1     1
     0     0     1     0     1     1     1
     0     0     0     1     1     0     1

genmat =

     1     0     1     1     1     0     0
     1     1     1     0     0     1     0
     0     1     1     1     0     0     1
```

The example below converts a generator matrix for a [5,3] linear block code into the corresponding parity-check matrix.

```
genmat = [1 0 0 1 0; 0 1 0 1 1; 0 0 1 0 1];
parmat = gen2par(genmat)
```

```
parmat =

     1     1     0     1     0
     0     1     1     0     1
```

The same function `gen2par` can also convert a parity-check matrix into a generator matrix.

Generator Polynomial

Cyclic codes, including the special case of BCH codes, have algebraic properties that allow a polynomial to determine the coding process completely. This so-called *generator polynomial* is a degree-(n-k) divisor of the polynomial x^n-1. Van Lint [4] explains how a generator polynomial determines a cyclic code.

The `cyclpoly` and `bchpoly` functions produce generator polynomials for generic cyclic codes and BCH codes, respectively. These functions represent a generator polynomial using a row vector that lists the polynomial's coefficients in order of *ascending* powers of the variable. For example, the command

```
genpoly = cyclpoly(7,3)

genpoly =

     1     0     1     1     1
```

finds that one valid generator polynomial for a [7,3] cyclic code is $1 + x^2 + x^3 + x^4$.

Decoding Table

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

This toolbox represents a decoding table as a matrix with `n` columns and `2^(n-k)` rows. Each row gives a correction vector for one received codeword vector. A Hamming decoding table has `n+1` rows. The `syndtable` function generates a decoding table for a given parity-check matrix.

Example: Using a Decoding Table. The script below shows how to use a Hamming decoding table to correct an error in a received message. The `hammgen` function produces the parity-check matrix, while the `syndtable` function produces the decoding table. The transpose of the parity-check matrix is multiplied on the left by the received codeword, yielding the *syndrome*. The decoding table helps determine the correction vector. The corrected codeword is the sum (modulo 2) of the correction vector and the received codeword.

```
% Use a [7,4] Hamming code.
m = 3; n = 2^m-1; k = n-m;
parmat = hammgen(m); % Produce parity-check matrix.
trt = syndtable(parmat); % Produce decoding table.
recd = [1 0 0 1 1 1 1] % Suppose this is the received vector.
syndrome = rem(recd * parmat',2);
syndrome_de = bi2de(syndrome,'left-msb'); % Convert to decimal.
disp(['Syndrome = ',num2str(syndrome_de),...
      ' (decimal), ',num2str(syndrome),' (binary)'])
corrvect = trt(1+syndrome_de,:) % Correction vector
% Now compute the corrected codeword.
correctedcode = rem(corrvect+recd,2)
```

The output is below.

```
recd =

     1     0     0     1     1     1     1

Syndrome = 3 (decimal), 0  1  1 (binary)

corrvect =

     0     0     0     0     1     0     0


correctedcode =

     1     0     0     1     0     1     1
```

Creating and Decoding Binary Block Codes

The functions for encoding and decoding linear block codes are `encode`, `decode`, `bchenco`, and `bchdeco`. The first two in this list are general-purpose functions that invoke other functions from the list when appropriate. This section discusses how to use these functions to create and decode generic linear block codes, cyclic codes, BCH codes, and Hamming codes.

Generic Linear Block Codes

Encoding a message using a generic linear block code requires a generator matrix. If you have defined variables `msg`, `n`, `k`, and `genmat`, then either of the commands

```
code = encode(msg,n,k,'linear',genmat);
code = encode(msg,n,k,'linear/decimal',genmat);
```

encodes the information in `msg` using the [n,k] code that the generator matrix `genmat` determines. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See "Representing Words for Binary Block Codes" on page 2-26 or the reference page for `encode` for a description of the formats of `msg` and `code`.

Decoding the code requires the generator matrix and possibly a decoding table. If you have defined variables `code`, `n`, `k`, `genmat`, and possibly also `trt`, then the commands

```
newmsg = decode(code,n,k,'linear',genmat);
newmsg = decode(code,n,k,'linear/decimal',genmat);
newmsg = decode(code,n,k,'linear',genmat,trt);
newmsg = decode(code,n,k,'linear/decimal',genmat,trt);
```

decode the information in `code`, using the [n,k] code that the generator matrix `genmat` determines. `decode` also corrects errors according to instructions in the decoding table that `trt` represents.

Example: Generic Linear Block Coding. The example below encodes a message, artificially adds some noise, decodes the noisy code, and keeps track of errors that the decoder detects along the way. Because the decoding table contains only zeros, the decoder does not correct any errors.

```
n = 4; k = 2;
genmat = [[1 1; 1 0], eye(2)]; % Generator matrix
```

```
msg = [0 1; 0 0; 1 0]; % Three messages, two bits each
% Create three codewords, four bits each.
code = encode(msg,n,k,'linear',genmat);
noisycode = rem(code + randerr(3,4,[0 1;.7 .3]),2); % Add noise.
trt = zeros(2^(n-k),n);  % No correction of errors
% Decode, keeping track of all detected errors.
[newmsg,err] = decode(noisycode,n,k,'linear',genmat,trt);
err_words = find(err~=0) % Find out which words had errors.
```

The output indicates that errors occurred in the first and second words. Your results might vary because this example uses random numbers as errors.

```
err_words =

     1
     2
```

Cyclic Codes

Encoding a message using a cyclic code requires a generator polynomial. If you have defined variables `msg`, `n`, `k`, and `genpoly`, then either of the commands

```
code = encode(msg,n,k,'cyclic',genpoly);
code = encode(msg,n,k,'cyclic/decimal',genpoly);
```

encodes the information in `msg` using the [`n`,`k`] code determined by the generator polynomial `genpoly`. `genpoly` is an optional argument for `encode`. The default generator polynomial is `cyclpoly(n,k)`. The `/decimal` option, suitable when `2^n` and `2^k` are not very large, indicates that `msg` contains nonnegative decimal integers rather than their binary representations. See “Representing Words for Binary Block Codes” on page 2-26 or the reference page for `encode` for a description of the formats of `msg` and `code`.

Decoding the code requires the generator polynomial and possibly a decoding table. If you have defined variables `code`, `n`, `k`, `genpoly`, and `trt`, then the commands

```
newmsg = decode(code,n,k,'cyclic',genpoly);
newmsg = decode(code,n,k,'cyclic/decimal',genpoly);
newmsg = decode(code,n,k,'cyclic',genpoly,trt);
newmsg = decode(code,n,k,'cyclic/decimal',genpoly,trt);
```

decode the information in code, using the [n,k] code that the generator matrix genmat determines. decode also corrects errors according to instructions in the decoding table that trt represents. genpoly is an optional argument in the first two syntaxes above. The default generator polynomial is cyclpoly(n,k).

Example. You can modify the example in the section "Generic Linear Block Codes" on page 2-34 so that it uses the cyclic coding technique, instead of the linear block code with the generator matrix genmat. Make the changes listed below:

- Replace the second line by

  ```
  genpoly = [1 0 1]; % generator poly is 1 + x^2
  ```

- In the fifth and ninth lines (encode and decode commands), replace genmat by genpoly and replace 'linear' by 'cyclic'.

Another example of encoding and decoding a cyclic code is on the reference page for encode.

BCH Codes

BCH codes are a special case of cyclic codes, though the decoding algorithm for BCH codes is more complicated than that for generic cyclic codes. The discussion in the section "Cyclic Codes" above applies almost exactly to the case of BCH codes. The only differences are

- **bch** replaces **cyclic** in the syntax for encode and decode.
- bchpoly(n,k) replaces cyclpoly(n,k) as the default generator polynomial.
- n and k must be valid codeword and message lengths for BCH code.

Valid codeword lengths for BCH code are those integers of the form 2^m-1 for some integer m greater than or equal to 3. Given a valid BCH codeword length, the corresponding valid BCH message lengths are those numbers in the second column of the output of the command below.

```
params = bchpoly(n); % Where n = 2^m-1 for some integer m >= 3
```

For example, the output of the command below shows that a BCH code with codeword length 15 can have message length 5, 7, or 11. No other message lengths are valid for this codeword length.

```
params = bchpoly(15)
```

```
params =

    15    11     1
    15     7     2
    15     5     3
```

The third column of the output above represents the error-correction capability for each pair of codeword length and message length.

Choice of Functions for BCH Coding. To process BCH codes, you can use either the `encode` and `decode` functions, or the lower level `bchenco` and `bchdeco` functions. The syntax of the lower level functions is slightly different from that of the higher level functions. The only difference in functionality is that the higher level functions prepare the input data (including default values of options that you omit) before invoking the lower level functions. The reference page for `encode` contains an example that uses `encode` and `decode`. The reference pages for `bchenco` and `bchdeco` contain other examples.

Hamming Codes

The reference pages for `encode` and `decode` contain examples of encoding and decoding Hamming codes. Also, the section “Decoding Table” on page 2-32 illustrates error correction in a Hamming code.

Performing Other Binary Block Code Tasks

This section describes functions that compute typical parameters associated with block codes and functions that convert information from one format to another. Specific tasks are

- Finding a generator polynomial
- Finding generator and parity-check matrices
- Converting between parity-check and generator matrices
- Finding the error-correction capability

Finding a Generator Polynomial

To find a generator polynomial for a cyclic or BCH code, use the `cyclpoly` or `bchpoly` function, respectively. The commands

```
genpolyCyclic = cyclpoly(7,4);
genpolyBCH = bchpoly(7,4);
```

represent valid ways to find one generator polynomial for a [7,4] code of the respective coding method. The result is suitable for use in other block coding functions, such as `encode` or `rsenc`.

Some pairs of message length and codeword length do not uniquely determine the generator polynomial. The syntax for `cyclpoly` includes ways to retrieve all valid generator polynomials or those that satisfy certain constraints that you specify. See the reference page for `cyclpoly` for details about syntax options.

For example, the command

```
genpolys = cyclpoly(7,4,'all')

genpolys =

     1     0     1     1
     1     1     0     1
```

shows that $1 + x^2 + x^3$ and $1 + x + x^3$ are two possible generator polynomials for a [7,4] cyclic code.

Finding Generator and Parity-Check Matrices

To find a parity-check and generator matrix for a Hamming code with codeword length `2^m-1`, use the `hammgen` function as below. `m` must be at least three.

```
[parmat,genmat] = hammgen(m); % Hamming
```

To find a parity-check and generator matrix for a cyclic code, use the `cyclgen` function. You must provide the codeword length and a valid generator polynomial. You can use the `cyclpoly` function to produce one possible generator polynomial after you provide the codeword length and message length. For example,

```
[parmat,genmat] = cyclgen(7,cyclpoly(7,4)); % Cyclic
```

To find a parity-check and generator matrix for a BCH code, use the same `cyclgen` function mentioned above. Because the generator polynomial must now be valid for BCH code, the `bchpoly` function replaces `cyclpoly`.

```
[parmat,genmat] = cyclgen(7,bchpoly(7,4)); % BCH
```

Converting Between Parity-Check and Generator Matrices

The `gen2par` function converts a generator matrix into a parity-check matrix, and vice versa. Examples to illustrate this are on the reference page for `gen2par`.

Finding the Error-Correction Capability of a BCH Code

The `bchpoly` function can compute the error-correction capability of a BCH code, which depends on the codeword length and message length. To retrieve the error-correction capability `t` of a BCH code, use the command below.

```
[temp1,temp2,temp3,temp4,t] = bchpoly(n,k);
```

Representing Words for Reed-Solomon Codes

This toolbox supports Reed-Solomon codes that use m-bit symbols instead of bits. If you want to create an [n,k] Reed-Solomon code, then your message must be a k-column Galois array in the field GF(2^m). Each array entry must be an integer between 0 and 2^m-1. The code corresponding to that message is an n-column Galois array in GF(2^m). The codeword length `n` must be between 3 and 2^m-1.

Note For information about Galois arrays and how to create them, see "Representing Elements of Galois Fields" on page 2-94 or the reference page for the `gf` function.

The example below illustrates how to represent words for a [7,3] Reed-Solomon code.

```
n = 7; k = 3; % Codeword length and message length
m = 3; % Number of bits in each symbol
msg = gf([1 6 4; 0 4 3],m); % Message is a Galois array.
c = rsenc(msg,n,k) % Code will be a Galois array.
```

The output is

```
c = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     1     6     4     4     3     6     3
     0     4     3     3     7     4     7
```

Parameters for Reed-Solomon Codes

This section describes several integers related to Reed-Solomon codes and discusses how to find generator polynomials.

Allowable Values of Integer Parameters

The table below summarizes the meanings and allowable values of some positive integer quantities related to Reed-Solomon codes as supported in this toolbox. The quantities `n` and `k` are input parameters for Reed-Solomon functions in this toolbox.

Symbol	Meaning	Value or Range
m	Number of bits per symbol	Integer between 3 and 16
`n`	Number of symbols per codeword	Integer between 3 and 2^m-1
`k`	Number of symbols per message	Positive integer less than `n`, such that `n-k` is even
t	Error-correction capability of the code	`(n-k)/2`

Generator Polynomial

The `rsgenpoly` function produces generator polynomials for Reed-Solomon codes. It is useful if you want to use `rsenc` and `rsdec` with a generator polynomial other than the default, or if you want to examine or manipulate a generator polynomial. `rsgenpoly` represents a generator polynomial using a Galois row vector that lists the polynomial's coefficients in order of *descending* powers of the variable. If each symbol has m bits, then the Galois row vector is in the field GF(2^m). For example, the command

```
r = rsgenpoly(15,13)

r = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1     6     8
```

finds that one generator polynomial for a [15,13] Reed-Solomon code is $X^2 + (A^2 + A)X + (A^3)$, where A is a root of the default primitive polynomial for GF(16).

Algebraic Expression for Generator Polynomials. The generator polynomials that `rsgenpoly` produces have the form $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$, where b is an integer, A is a root of the primitive polynomial for the Galois field, and t is `(n-k)/2`. The default value of b is 1. The output from `rsgenpoly` is the result of multiplying the factors and collecting like powers of X. The example below checks this formula for the case of a [15,13] Reed-Solomon code, using b = 1.

```
n = 15;
a = gf(2,log2(n+1)); % Root of primitive polynomial
f1 = [1 a]; f2 = [1 a^2]; % Factors that form generator polynomial
f = conv(f1,f2) % Generator polynomial, same as r above.
```

Creating and Decoding Reed-Solomon Codes

The `rsenc` and `rsdec` functions create and decode Reed-Solomon codes, using the data described in "Representing Words for Reed-Solomon Codes" on page 2-39 and "Parameters for Reed-Solomon Codes" on page 2-40.

This section illustrates how to use `rsenc` and `rsdec`. The topics are

Example: Reed-Solomon Coding Syntaxes

The example below illustrates multiple ways to encode and decode data using a [15,13] Reed-Solomon code. The example shows that you can

- Vary the generator polynomial for the code, using `rsgenpoly` to produce a different generator polynomial
- Vary the primitive polynomial for the Galois field that contains the symbols, using an input argument in `gf`.
- Vary the position of the parity symbols within the codewords, choosing either the end (default) or beginning

The example also shows that corresponding syntaxes of `rsenc` and `rsdec` use the same input arguments, except for the first input argument.

```
m = 4; % Number of bits in each symbol
n = 2^m-1; k = 13; % Codeword length and message length
data = randint(4,k,2^m); % Four random integer messages
msg = gf(data,m); % Represent data using a Galois array.

% Simplest syntax for encoding
c1 = rsenc(msg,n,k);
d1 = rsdec(c1,n,k);

% Vary the generator polynomial for the code.
c2 = rsenc(msg,n,k,rsgenpoly(n,k,19,2));
d2 = rsdec(c2,n,k,rsgenpoly(n,k,19,2));

% Vary the primitive polynomial for GF(16).
msg2 = gf(data,m,25);
c3 = rsenc(msg2,n,k);
d3 = rsdec(c3,n,k);

% Prepend the parity symbols instead of appending them.
c4 = rsenc(msg,n,k,'beginning');
d4 = rsdec(c4,n,k,'beginning');

% Check that the decoding worked correctly.
chk = isequal(d1,msg) & isequal(d2,msg) & isequal(d3,msg2) &...
isequal(d4,msg)

chk =

     1
```

Example: Detecting and Correcting Errors

The example below illustrates the decoding results for a corrupted code. The example encodes some data, introduces errors in each codeword, and invokes `rsdec` to attempt to decode the noisy code. It uses additional output arguments in `rsdec` to gain information about the success of the decoding process.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Codeword length and message length
t = (n-k)/2; % Error-correction capability of the code
nw = 4; % Number of words to process
msgw = gf(randint(nw,k,2^m),m); % Random k-symbol messages
c = rsenc(msgw,n,k); % Encode the data.
noise = (1+randint(nw,n,2^m-1)).*randerr(nw,n,t); % t errors/row
cnoisy = c + noise; % Add noise to the code.
[dc,nerrs,corrcode] = rsdec(cnoisy,n,k); % Decode the noisy code.
% Check that the decoding worked correctly.
isequal(dc,msgw) & isequal(corrcode,c)
nerrs % Find out how many errors rsdec corrected.
```

Notice that the array of noise values contains integers between 1 and `2^m`, and that the addition operation `c + noise` takes place in the Galois field GF(`2^m`) because `c` is a Galois array in GF(`2^m`).

The output from the example is below. The nonzero value of `ans` indicates that the decoder was able to correct the corrupted codewords and recover the original message. The values in the vector `nerrs` indicates that the decoder corrected `t` errors in each codeword.

```
ans =

     1


nerrs =

     2
     2
     2
     2
```

Excessive Noise in Reed-Solomon Codewords

In the previous example, `rsdec` corrected all of the errors. However, each Reed-Solomon code has a finite error-correction capability. If the noise is so great that the corrupted codeword is too far in Hamming distance from the correct codeword, then either

- The corrupted codeword is close to a valid codeword *other than* the correct codeword. The decoder returns the message that corresponds to the other codeword.
- The corrupted codeword is not close enough to any codeword for successful decoding. This situation is called a *decoding failure*. The decoder removes the symbols in parity positions from the corrupted codeword and returns the remaining symbols.

In both cases, the decoder returns the wrong message. However, you can tell when a decoding failure occurs because `rsdec` also returns a value of `-1` in its second output.

To examine cases in which codewords are too noisy for successful decoding, change the previous example so that the definition of `noise` is

```
noise = (1+randint(nw,n,n)).*randerr(nw,n,t+1); % t+1 errors/row
```

Creating Shortened Reed-Solomon Codes

Every Reed-Solomon encoder uses a codeword length that equals 2^m-1 for an integer m. A shortened Reed-Solomon code is one in which the codeword length is not 2^m-1. A shortened [n,k] Reed-Solomon code implicitly uses an $[n_1,k_1]$ encoder, where

$$n_1 = 2^m - 1 \text{ where m is the number of bits per symbol}$$

$$k_1 = k + (n_1 - n)$$

The `rsenc` and `rsdec` functions support shortened codes using the same syntaxes that they use for nonshortened codes. You do not need to indicate explicitly that you want to use a shortened code. For example, compare the two similar-looking commands below. The first creates a (nonshortened) [7,5] code. The second causes `rsenc` to create a [5,3] shortened code by implicitly using a [7,5] encoder.

```
m = 3; ordinarycode = rsenc(gf([1 1 1 1 1],m),7,5);
m = 3; shortenedcode = rsenc(gf([1 1 1],m),5,3);
```

How rsenc Creates a Shortened Code. When creating a shortened code, `rsenc` performs these steps:

- Pads each message by prepending zeros
- Encodes each padded message using a Reed-Solomon encoder having an allowable codeword length and the desired error-correction capability
- Removes the extra zeros from the nonparity symbols of each codeword

The example below illustrates this process. Note that forming a [12,8] Reed-Solomon code actually uses a [15,11] Reed-Solomon encoder. Also note that you do not have to indicate in the `rsenc` syntax that this is a shortened code or that the proper encoder to use is [15,11].

```
n = 12; k = 8; % Lengths for the shortened code
m = ceil(log2(n+1)); % Number of bits per symbol
msg = gf(randint(3,k,2^m),m); % Random array of 3 k-symbol words
code = rsenc(msg,n,k); % Create a shortened code.

% Do the shortening manually, just to show how it works.
n_pad = 2^m-1; % Codeword length in the actual encoder
k_pad = k+(n_pad-n); % Message length in the actual encoder
msg_pad=[zeros(3, n_pad-n), msg]; % Prepend zeros to each word.
code_pad = rsenc(msg_pad,n_pad,k_pad); % Encode padded words.
code_eqv = code_pad(:,n_pad-n+1:n_pad); % Remove extra zeros.
ck = isequal(code_eqv,code); % Returns true (1).
```

Selected Bibliography for Block Coding

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

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

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

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

Convolutional Coding

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

This section

- Outlines the convolutional coding features of the Communications Toolbox
- Defines the two supported ways to describe a convolutional encoder:
  - Polynomial description
  - Trellis description
- Describes how to encode and decode using the `convenc` and `vitdec` functions
- Gives additional examples of convolutional coding

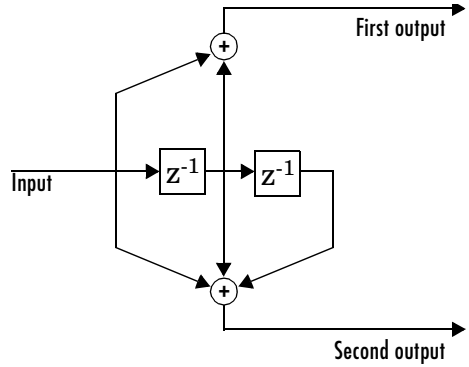
Convolutional Coding Features of the Toolbox

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

For background information about convolutional coding, see the works listed in "Selected Bibliography for Convolutional Coding" on page 2-58.

Polynomial Description of a Convolutional Encoder

A polynomial description of a convolutional encoder describes the connections among shift registers and modulo-2 adders. For example, the figure below depicts a feedforward convolutional encoder that has one input, two outputs, and two shift registers.



A polynomial description of a convolutional encoder has either two or three components, depending on whether the encoder is a feedforward or feedback type:

- Constraint lengths
- Generator polynomials
- Feedback connection polynomials (for feedback encoders only)

Constraint Lengths

The constraint lengths of the encoder form a vector whose length is the number of inputs in the encoder diagram. The elements of this vector indicate the number of bits stored in each shift register, *including* the current input bits.

In the figure above, the constraint length is three. It is a scalar because the encoder has one input stream, and its value is one plus the number of shift registers for that input.

Generator Polynomials

If the encoder diagram has k inputs and n outputs, then the code generator matrix is a k-by-n matrix. The element in the ith row and jth column indicates how the ith input contributes to the jth output.

For *systematic* bits of a systematic feedback encoder, match the entry in the code generator matrix with the corresponding element of the feedback connection vector. See “Feedback Connection Polynomials” below for details.

In other situations, you can determine the (i,j) entry in the matrix as follows:

1. Build a binary number representation by placing a 1 in each spot where a connection line from the shift register feeds into the adder, and a 0 elsewhere. The leftmost spot in the binary number represents the current input, while the rightmost spot represents the oldest input that still remains in the shift register.

2. Convert this binary representation into an octal representation by considering consecutive triplets of bits, starting from the rightmost bit. The rightmost bit in each triplet is the least significant. If the number of bits is not a multiple of three, then place zero bits at the left end as necessary. (For example, interpret 1101010 as 001 101 010 and convert it to 152.)

For example, the binary numbers corresponding to the upper and lower adders in the figure above are 110 and 111, respectively. These binary numbers are equivalent to the octal numbers 6 and 7, respectively. Thus the generator polynomial matrix is [6 7].

Note You can perform the binary-to-octal conversion in MATLAB by using code like `str2num(dec2base(bin2dec('110'),8))`.

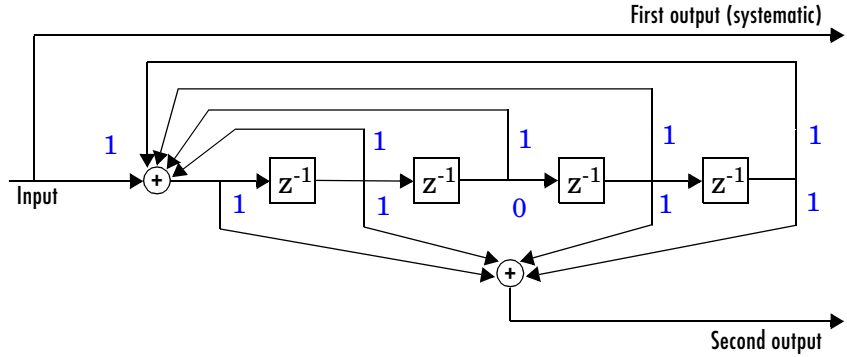
For a table of some good convolutional code generators, refer to [1] in the section "Selected Bibliography for Block Coding" on page 2-45, especially that book's appendices.

Feedback Connection Polynomials

If you are representing a feedback encoder, then you need a vector of feedback connection polynomials. The length of this vector is the number of inputs in the encoder diagram. The elements of this vector indicate the feedback connection for each input, using an octal format. First build a binary number representation as in step 1 above. Then convert the binary representation into an octal representation as in step 2 above.

If the encoder has a feedback configuration and is also systematic, then the code generator and feedback connection parameters corresponding to the systematic bits must have the same values.

For example, the diagram below shows a rate 1/2 systematic encoder with feedback.



This encoder has a constraint length of 5, a generator polynomial matrix of `[37 33]`, and a feedback connection polynomial of `37`.

The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits. The feedback polynomial is represented by the binary vector [1 1 1 1 1], corresponding to the upper row of binary digits in the diagram. These digits indicate connections from the outputs of the registers to the adder. Note that the initial 1 corresponds to the input bit. The octal representation of the binary number 11111 is 37.

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

Using the Polynomial Description in MATLAB

To use the polynomial description with the functions `convenc` and `vitdec`, first convert it into a trellis description using the `poly2trellis` function. For example, the command below computes the trellis description of the encoder pictured in the section "Polynomial Description of a Convolutional Encoder" on page 2-46.

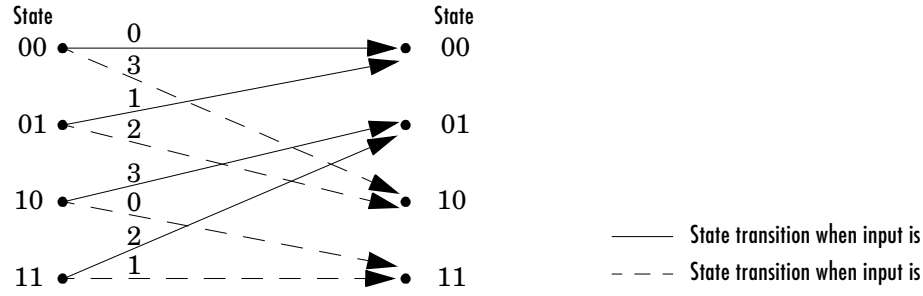
```
trellis = poly2trellis(3,[6 7]);
```

The MATLAB structure `trellis` is a suitable input argument for `convenc` and `vitdec`.

Trellis Description of a Convolutional Encoder

A trellis description of a convolutional encoder shows how each possible input to the encoder influences both the output and the state transitions of the encoder. This section describes trellises, describes how to represent trellises in MATLAB, and gives an example of a MATLAB trellis.

The figure below depicts a trellis for the convolutional encoder from the previous section. The encoder has four states (numbered in binary from 00 to 11), a one-bit input, and a two-bit output. (The ratio of input bits to output bits makes this encoder a rate-1/2 encoder.) Each solid arrow shows how the encoder changes its state if the current input is zero, and each dashed arrow shows how the encoder changes its state if the current input is one. The octal numbers above each arrow indicate the current output of the encoder.



As an example of interpreting this trellis diagram, if the encoder is in the 10 state and receives an input of zero, then it outputs the code symbol 3 and changes to the 01 state. If it is in the 10 state and receives an input of one, then it outputs the code symbol 0 and changes to the 11 state.

Note that any polynomial description of a convolutional encoder is equivalent to some trellis description, although some trellises have no corresponding polynomial descriptions.

Specifying a Trellis in MATLAB

To specify a trellis in MATLAB, use a specific form of a MATLAB structure called a trellis structure. A trellis structure must have five fields, as in the table below.

Fields of a Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputsymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in decimal) for all combinations of current state and current input

Note While your trellis structure can have any name, its fields must have the *exact* names as in the table. Field names are case sensitive.

In the `nextStates` matrix, each entry is an integer between 0 and `numStates`-1. The element in the ith row and jth column denotes the next state when the starting state is i-1 and the input bits have decimal representation j-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the `nextStates` matrix stores the next states when the current set of input values is {0,...,0,1}. To learn how to assign numbers to states, see the reference page for `istrellis`.

In the `outputs` matrix, the element in the ith row and jth column denotes the encoder's output when the starting state is i-1 and the input bits have decimal representation j-1. To convert to decimal value, use the first output bit as the MSB.

How to Create a MATLAB Trellis Structure

Once you know what information you want to put into each field, you can create a trellis structure in any of these ways:

- Define each of the five fields individually, using `structurename.fieldname` notation. For example, set the first field of a structure called `s` using the command below. Use additional commands to define the other fields.

  ```
  s.numInputSymbols = 2;
  ```

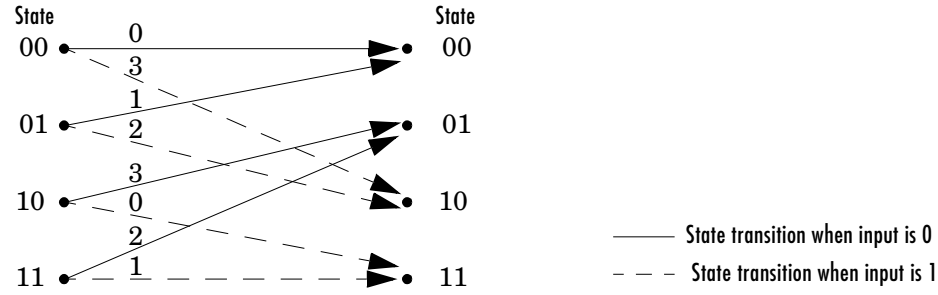
  The reference page for the `istrellis` function illustrates this approach.
- Collect all field names and their values in a single `struct` command. For example:

  ```
  s = struct('numInputSymbols',2,'numOutputSymbols',2,...
  'numStates',2,'nextStates',[0 1;0 1],'outputs',[0 0;1 1]);
  ```
- Start with a polynomial description of the encoder and use the `poly2trellis` function to convert it to a valid trellis structure. The polynomial description of a convolutional encoder is described in "Polynomial Description of a Convolutional Encoder" on page 2-46.

To check whether your structure is a valid trellis structure, use the `istrellis` function.

Example: A MATLAB Trellis Structure

Consider the trellis shown below.



To build a trellis structure that describes it, use the command below.

```
trellis = struct('numInputSymbols',2,'numOutputSymbols',4,...
'numStates',4,'nextStates',[0 2;0 2;1 3;1 3],...
'outputs',[0 3;1 2;3 0;2 1]);
```

The number of input symbols is 2 because the trellis diagram has two types of input path, the solid arrow and the dashed arrow. The number of output symbols is 4 because the numbers above the arrows can be either 0, 1, 2, or 3. The number of states is 4 because there are four bullets on the left side of the trellis diagram (equivalently, four on the right side). To compute the matrix of next states, create a matrix whose rows correspond to the four current states on the left side of the trellis, whose columns correspond to the inputs of 0 and 1, and whose elements give the next states at the end of the arrows on the right side of the trellis. To compute the matrix of outputs, create a matrix whose rows and columns are as in the next states matrix, but whose elements give the octal outputs shown above the arrows in the trellis.

Creating and Decoding Convolutional Codes

The functions for encoding and decoding convolutional codes are `convenc` and `vitdec`. This section discusses using these functions to create and decode convolutional codes.

Encoding

A simple way to use `convenc` to create a convolutional code is shown in the commands below.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Define trellis.
code = convenc(ones(100,1),t); % Encode a string of ones.
```

The first command converts a polynomial description of a feedforward convolutional encoder to the corresponding trellis description. The second command encodes 100 bits, or 50 two-bit symbols. Because the code rate in this example is 2/3, the output vector `code` contains 150 bits (that is, 100 input bits times 3/2).

Hard-Decision Decoding

To decode using hard decisions, use the `vitdec` function with the flag `'hard'` and with *binary* input data. Because the output of `convenc` is binary, hard-decision decoding can use the output of `convenc` directly, without additional processing. This example extends the previous example and implements hard decision decoding.

```
t = poly2trellis([4 3],[4 5 17;7 4 2]); % Define trellis.
code = convenc(ones(100,1),t); % Encode a string of ones.
tb = 2; % Traceback length for decoding
```

```
decoded = vitdec(code,t,tb,'trunc','hard'); % Decode.
```

Soft-Decision Decoding

To decode using soft decisions, use the `vitdec` function with the flag `'soft'`. You must also specify the number, `nsdec`, of soft-decision bits and use input data consisting of integers between 0 and `2^nsdec-1`.

An input of 0 represents the most confident 0, while an input of `2^nsdec-1` represents the most confident 1. Other values represent less confident decisions. For example, the table below lists interpretations of values for 3-bit soft decisions.

Input Values for 3-bit Soft Decisions

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

Example: Soft-Decision Decoding. The script below illustrates decoding with 3-bit soft decisions. First it creates a convolutional code with `convenc` and adds white Gaussian noise to the code with `awgn`. Then, to prepare for soft-decision decoding, the example uses `quantiz` to map the noisy data values to appropriate decision-value integers between 0 and 7. The second argument in `quantiz` is a partition vector that determines which data values map to 0, 1, 2, etc. The partition is chosen so that values near 0 map to 0, and values near 1 map to 7. (You can refine the partition to obtain better decoding performance if your application requires it.) Finally, the example decodes the code and computes the bit error rate. Notice that when comparing the decoded data with

the original message, the example must take the decoding delay into account. The continuous operation mode of `vitdec` causes a delay equal to the traceback length, so `msg(1)` corresponds to `decoded(tblen+1)` rather than to `decoded(1)`.

```
msg = randint(4000,1,2,139); % Random data
t = poly2trellis(7,[171 133]); % Define trellis.
code = convenc(msg,t); % Encode the data.
ncode = awgn(code,6,'measured',244); % Add noise.

% Quantize to prepare for soft-decision decoding.
qcode = quantiz(ncode,[0.001,.1,.3,.5,.7,.9,.999]);

tblen = 48; delay = tblen; % Traceback length
decoded = vitdec(qcode,t,tblen,'cont','soft',3); % Decode.

% Compute bit error rate.
[number,ratio] = biterr(decoded(delay+1:end),msg(1:end-delay))
```

The output is below.

```
number =

     5


ratio =


    0.0013
```

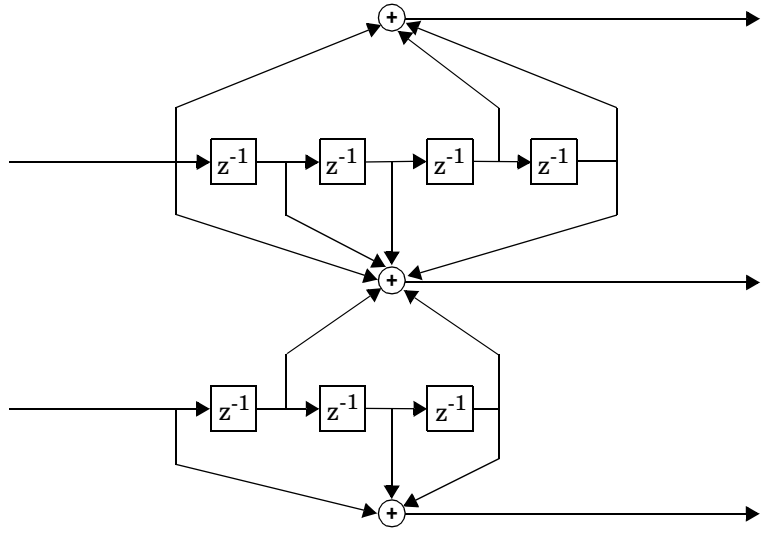
Examples of Convolutional Coding

This section contains more examples of convolutional coding:

- The first example determines the correct trellis parameter for its encoder and then uses it to process a code. The decoding process uses hard decisions and the continuous operation mode. This operation mode causes a decoding delay, which the error rate computation takes into account.
- The second example processes a punctured convolutional code. The decoding process uses the unquantized decision type.

Example: A Rate-2/3 Feedforward Encoder

The example below uses the rate 2/3 feedforward encoder depicted in the schematic below. The accompanying description explains how to determine the trellis structure parameter from a schematic of the encoder and then how to perform coding using this encoder.



Determining Coding Parameters. The `convenc` and `vitdec` functions can implement this code if their parameters have the appropriate values.

The encoder's constraint length is a vector of length 2 because the encoder has two inputs. The elements of this vector indicate the number of bits stored in each shift register, including the current input bits. Counting memory spaces in each shift register in the diagram and adding one for the current inputs leads to a constraint length of [5 4].

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

which is equivalent to the octal number 13. The full value of the code generator matrix is [23 35 0; 0 5 13].

To use the constraint length and code generator parameters in the `convenc` and `vitdec` functions, use the `poly2trellis` function to convert those parameters into a trellis structure. The command to do this is below.

```
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Define trellis.
```

Using the Encoder. Below is a script that uses this encoder.

```
len = 1000;
msg = randint(2*len,1); % Random binary message of 2-bit symbols
trel = poly2trellis([5 4],[23 35 0;0 5 13]); % Trellis
code = convenc(msg,trel); % Encode the message.
ncode = rem(code + randerr(3*len,1,[0 1;.96 .04]),2); % Add noise.
decoded = vitdec(ncode,trel,34,'cont','hard'); % Decode.
[number,ratio] = biterr(decoded(68+1:end),msg(1:end-68));
```

Notice that `convenc` accepts a vector containing 2-bit symbols and produces a vector containing 3-bit symbols, while `vitdec` does the opposite. Also notice that `biterr` ignores the first 68 elements of `decoded`. That is, the decoding delay is 68, which is the number of bits per symbol (2) of the recovered message times the traceback depth value (34) in the `vitdec` function. The first 68 elements of `decoded` are 0s, while subsequent elements represent the decoded messages.

Example: A Punctured Convolutional Code

This example processes a punctured convolutional code. It begins by generating 3000 random bits and encoding them using a rate-1/2 convolutional encoder. The resulting vector contains 6000 bits, which are mapped to values of -1 and 1 for transmission. The puncturing process removes every third value and results in a vector of length 4000. The punctured code, `punctcode`, passes through an additive white Gaussian noise channel. Afterwards, the example inserts values to reverse the puncturing process. While the puncturing process removed both -1s and 1s from code, the insertion process inserts zeros. Then `vitdec` decodes the vector of -1s, 1s, and 0s using the `'unquant'` decision type. This unquantized decision type is appropriate here for these reasons:

- `tcode` uses -1 to represent the 1s in `code`.
- `tcode` uses 1 to represent the 0s in `code`.

- The inserted 0s are acceptable for the `'unquant'` decision type, which allows any real values as input.

Finally, the example computes the bit error rate and the number of bit errors.

```
len = 3000; msg = randint(len,1,2,94384); % Random data
t = poly2trellis(7,[171 133]); % Define trellis.
code = convenc(msg,t); % Length is 2*len.
tcode = -2*code+1; % Transmit -1s and 1s.

% Puncture by removing every third value.
punctcode = tcode;
punctcode(3:3:end)=[]; % Length is (2*len)*2/3.

ncode = awgn(punctcode,8,'measured',1234); % Add noise.

% Insert zeros.
nicode = zeros(2*len,1); % Zeros represent inserted data.
nicode(1:3:end) = ncode(1:2:end); % Write actual data.
nicode(2:3:end) = ncode(2:2:end); % Write actual data.

decoded = vitdec(nicode,t,96,'trunc','unquant'); % Decode.
[number,ratio]=biterr(decoded,msg); % Bit error rate
```

Selected Bibliography for Convolutional Coding

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

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

Modulation

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

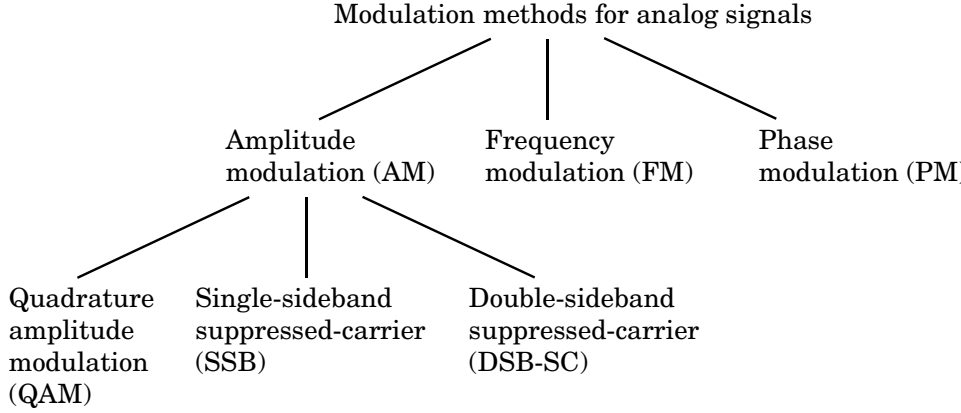
The table shows how this section is organized.

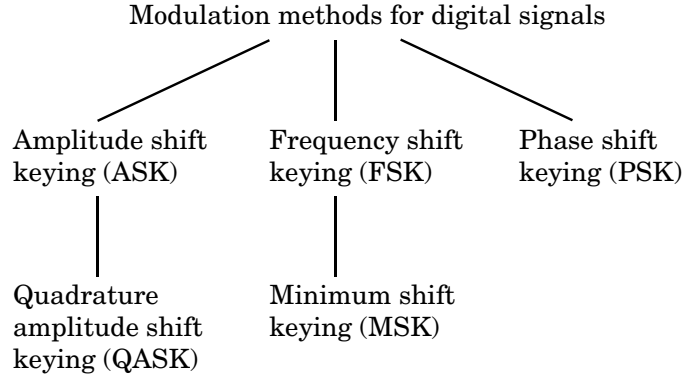
Subject	Topics
General modulation	"Modulation Features of the Toolbox" on page 2-60
	"Modulation Terminology" on page 2-61
Analog modulation	"Representing Analog Signals" on page 2-62
	"Simple Analog Modulation Example" on page 2-64
	"Other Options in Analog Modulation" on page 2-65
	"Filter Design Issues" on page 2-65
Digital modulation	"Digital Modulation Overview" on page 2-69
	"Representing Digital Signals" on page 2-70
	"Significance of Sampling Rates" on page 2-73
	"Representing Signal Constellations" on page 2-74
	"Simple Digital Modulation Example" on page 2-77
	"Customizing the Modulation Process" on page 2-79
	"Other Options in Digital Modulation" on page 2-81

For background information about modulation and demodulation, see the works listed in "Selected Bibliography for Modulation" on page 2-81.

Modulation Features of the Toolbox

The available methods of modulation depend on whether the input signal is analog or digital. The figures below show the modulation techniques that the Communications Toolbox supports for analog and digital signals, respectively. As the figures suggest, some categories of techniques include named special cases.





Baseband Versus Passband Simulation

For a given modulation technique, two ways to simulate modulation techniques are called *baseband* and *passband*. Baseband simulation, also known as the *lowpass equivalent method*, requires less computation. This toolbox supports both baseband and passband simulation. Because baseband simulation is more prevalent, this guide focuses more on baseband simulation.

Note To use this toolbox for passband simulation, see the reference pages for the functions `amod`, `ademod`, `dmod`, and `ddemod`.

Supported Modulation Tasks

Functions in the toolbox can accomplish these tasks:

- Modulate a signal using one of the techniques shown in the figures above
- Demodulate a signal using one of the techniques shown in the figures above
- Map a digital signal to an analog signal, before modulation
- Demap an analog signal to a digital signal, after demodulation
- Map, demap, and plot constellations for QASK modulation

The modulation and demodulation functions also let you control such features as the initial phase of the modulated signal, post-demodulation filtering, and the decision timing for digital demodulation.

Modulation Terminology

Modulation is a process by which a *carrier signal* is altered according to information in a *message signal*. The *carrier frequency*, denoted `Fc`, is the frequency of the carrier signal. The *sampling rate* is the rate at which the message signal is sampled during the simulation.

The frequency of the carrier signal is usually much greater than the highest frequency of the input message signal. The Nyquist sampling theorem requires that the simulation sampling rate `Fs` be greater than two times the highest frequency of the modulated signal, in order for the demodulator to recover the message correctly. The sampling rate `Fs` of a modulated digital signal is greater than or equal to the sampling rate `Fd` of the original message signal before modulation.

The table below lists the requirements in terms of the input arguments for this toolbox's modulation and demodulation functions. Note that the situations are not mutually exclusive.

Situation	Requirement
Passband simulation	`2*(highest frequency of modulated signal) < Fs`
Digital signals	`Fd ≤ Fs`
Passband simulation, digital signals	`Fd < Fc`

Representing Analog Signals

To perform baseband modulation of an analog signal using this toolbox, start with a real message signal and a sampling rate `Fs` in hertz. For modulation techniques *other than* quadrature amplitude modulation (QAM), represent the signal using a vector `x`, the entries of which give the signal's values in time increments of `1/Fs`. Baseband modulation (using a technique other than QAM) produces a complex vector.

For example, if `t` measures time in seconds, then the vector `x` below is the result of sampling a frequency-one sine wave 100 times per second for 2 seconds. The vector `y` represents the modulated signal. The output shows that `y` is complex.

```
Fs = 100; % Sampling rate is 100 samples per second.
t = [0:1/Fs:2]'; % Sampling times for 2 seconds
x = sin(2*pi*t); % Representation of the signal
y = amodce(x,Fs,'pm'); % Modulate x to produce y.
whos
  Name      Size            Bytes  Class

  Fs        1x1                 8  double array
  t       201x1              1608  double array
  x       201x1              1608  double array
  y       201x1              3216  double array (complex)

Grand total is 604 elements using 6440 bytes
```

Baseband Modulated Signals Defined

This section explains the connection between this complex vector y and the real signal that you might expect to get after modulating a real signal. If the modulated signal has the waveform

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

where f_c is the carrier frequency and θ is the carrier signal's initial phase, then a baseband simulation recognizes that this equals the real part of

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

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

$$(Y_1(t) + jY_2(t))\exp(j\theta)$$

Note You can also simultaneously process several signals of equal length. To do this, make x a matrix in which each signal occupies one column. The corresponding modulated signal y is a complex matrix whose kth column is the modulation of the kth column of x.

Changes for QAM

The case for quadrature amplitude modulation (QAM) is similar, except that the message signal has in-phase and quadrature components. Represent the signal using a matrix x that has an even number of columns. The odd-indexed columns represent in-phase components of the signal and the even-indexed columns represent quadrature components. If the message signal is a $2n$-by-m matrix, then the modulated signal is an n-by-m matrix. As in the other methods, baseband modulation turns a real message signal into a complex modulated signal.

For example, the code below implements QAM on a set of sinusoidal input signals.

```
Fs = 100; % Sampling rate is 100 samples per second.
t = [0:1/Fs:2]'; % Sampling times
% Signal is a four column matrix.
```

```
% Each column models 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]);
y = amodce(x,Fs,'qam'); % Modulate x to produce y.
```

The output below shows the sizes and types of `x` and `y`.

```
whos
  Name      Size            Bytes  Class

  Fs        1x1                 8  double array
  t       201x1              1608  double array
  x       201x4              6432  double array
  y       201x2              6432  double array (complex)

Grand total is 1408 elements using 14480 bytes
```

Simple Analog Modulation Example

This example illustrates the basic format of the baseband modulation and demodulation commands, `amodce` and `ademodce`. Although the example uses the AMDSB-TC method, most elements of this example apply to other analog modulation techniques as well. The example samples an analog signal and modulates it. Then it demodulates it and displays the order of magnitude of the variance between the original and demodulated signals.

```
% Sample the signal for two seconds,
% at a rate of 100 samples per second.
Fs = 100;
t = [0:1/Fs:2]';
% The signal is a sum of sinusoids.
x = sin(2*pi*t) + sin(4*pi*t);
% Use AMDSB-TC modulation to produce y.
y = amodce(x,Fs,'amdsb-tc');
% Demodulate y to recover the message.
z = ademodce(y,Fs,'amdsb-tc');
v = floor(log10(var(x-z)))

v =

   -33
```

Other Options in Analog Modulation

The table below lists a few ways in which you might vary the commands in "Simple Analog Modulation Example" on page 2-64 in order to perform the modulation and demodulation slightly differently. See the reference pages for full details about options.

Substitutions in Simple Analog Modulation Example

Modification of Process	Modifications in the Code
Set the carrier signal's initial phase to `phs`, measured in radians.	`y = amodce(x,[Fs phs],'amdsb-tc');` `z = ademodce(y,[Fs phs],'amdsb-tc');`
Use a lowpass filter after demodulating. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function.	`z = ademodce(y,Fs,'amdsb-tc',0,num,den);` (For other demodulation methods, the `0` in the statement above would be unnecessary. See the reference page for `ademodce` for details.)
(*AM-SSB only*) Use a Hilbert filter in the time domain. `num` and `den` are as above.	`y = amodce(x,Fs,'amssb/time',num,den);` `z = ademodce(y,Fs,'amssb');`

(*AMDSB only*) Use a Costas phase-locked loop.	`z = ademodce(y,Fs,'amdsb-tc/costas');`
	
	or
	
	`y = amodce(x,Fs,'amdsb-sc');`
	`z = ademodce(y,Fs,'amdsb-sc/costas');`

(*AMDSB-TC only*) Shift the signal values by `offset` before modulating and after demodulating.	`y = amodce(x,Fs,'amdsb-tc',offset);` `z = ademodce(y,Fs,'amdsb-tc',offset);`

Filter Design Issues

After demodulating, you might want to filter out the carrier signal, especially if you are using passband simulation. The Signal Processing Toolbox provides functions that can help you design your filter, such as `butter`, `cheby1`, `cheby2`,

and `ellip`. Different demodulation methods have different properties, and you might need to test your application with several filters before deciding which is most suitable. This subsection mentions two issues that relate to the use of filters: cutoff frequency and time lag.

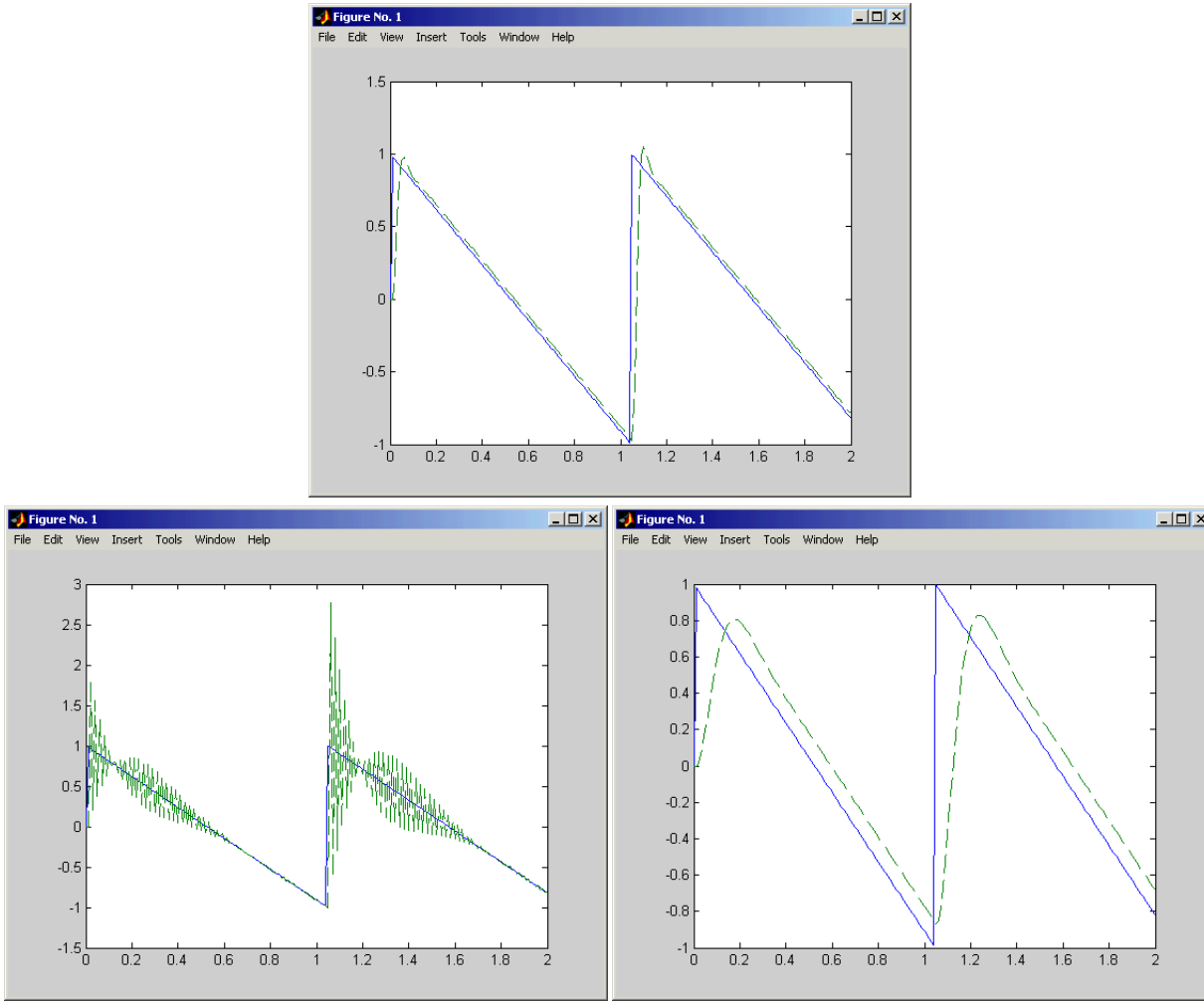
Example: Varying the Filter's Cutoff Frequency

In many situations, a suitable cutoff frequency is half the carrier frequency. Because the carrier frequency must be higher than the bandwidth of the message signal, a cutoff frequency chosen in this way limits the bandwidth of the message signal. If the cutoff frequency is too high, then the carrier frequency might not be filtered out. If the cutoff frequency is too low, then it might narrow the bandwidth of the message signal.

The code below modulates a sawtooth message signal, demodulates the resulting signal using a Butterworth filter, and plots the original and recovered signals. Note that the scaling in the `butter` function causes the cutoff frequency of the filter to be `F*Fs/2`, not `F` itself.

```
Fc = 25; % Carrier frequency
Fs = 100; % Signal sampling rate
t = [0:1/Fs:2]'; % Times to sample the signal
x = sawtooth(6*t,0); % Signal is a sawtooth.
y = amod(x,Fc,Fs,'amssb'); % Modulate.
F = Fc/Fs; % Change F to vary the filter's cutoff frequency.
[num,den] = butter(2,F); % Design Butterworth filter.
z = ademod(y,Fc,Fs,'amssb',num,den); % Demodulate and filter.
plot(t,x,'-',t,z,'--') % Plot original and recovered signals.
```

The plots below show the effects of three lowpass filters with different cutoff frequencies. In each plot, the dotted curve is the demodulated signal and the solid curve is the original message signal. The top plot uses the suggested cutoff frequency (`F = Fc/Fs`). The lower left plot uses a higher cutoff frequency (`F = 3.9*Fc/Fs`), which allows the carrier signal to interfere with the demodulated signal. The lower right plot uses a lower cutoff frequency (`F = Fc/Fs/4`), which narrows the bandwidth of the demodulated signal.



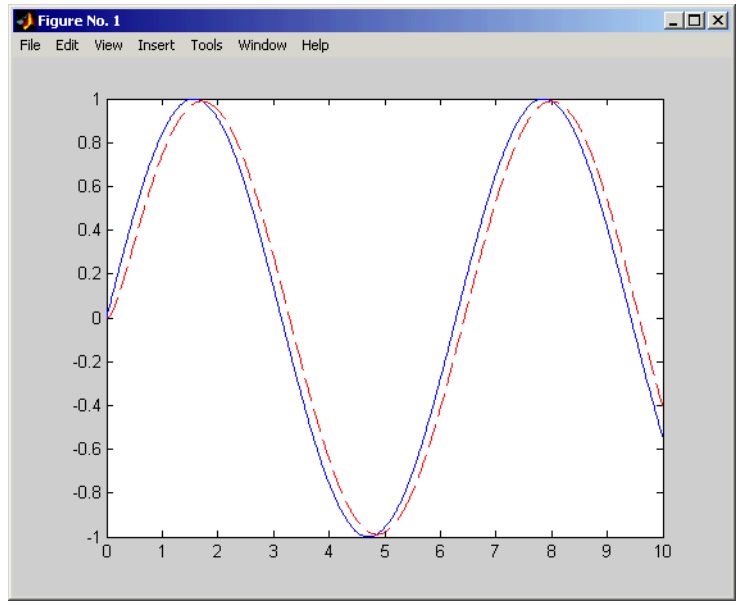
Original and Recovered Signals, with Filter Cutoff F = Fc/Fs, 3.9*Fc/Fs, and Fc/Fs/4

Example: Time Lag From Filtering

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. The example below illustrates the time delay by

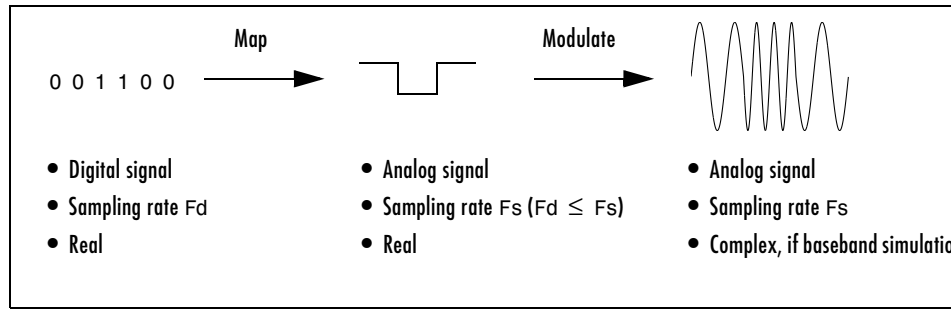
plotting a signal before and after the modulation, demodulation, and filtering processes. The solid curve is the original sine wave and the dashed curve is the recovered signal.

```
Fs = 100; % Sampling rate of signal
[num,den] = butter(2,0.8); % Design Butterworth filter.
t = [0:1/Fs:10]'; % Times to sample the signal
x = sin(t); % Signal is a sine wave.
y = amodce(x,Fs,'pm'); % Modulate.
z = ademodce(y,Fs,'pm',num,den); % Demodulate and filter.
plot(t,x,t,z,'r--') % Plot original signal and recovered signal.
```



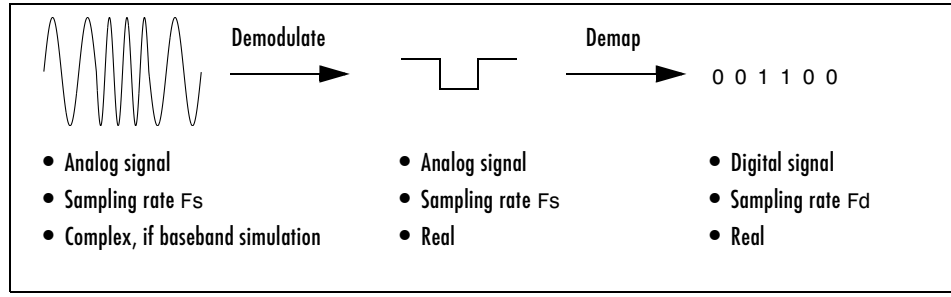
Digital Modulation Overview

Modulating a digital signal can be interpreted as a combination of two steps: mapping the digital signal to an analog signal and modulating the analog signal. These are depicted in the schematic below.



Two Steps of Digital Modulation

Except for FSK and MSK methods, when the receiver tries to recover a digital message from the analog signal that it receives, it performs two steps: demodulating the analog signal and demapping the demodulated analog signal to produce a digital message. These are depicted in the schematic below.



Two Steps of Digital Demodulation

For FSK and MSK methods, the demodulator uses correlation techniques instead of the two-stage process above.

The mapping process increases the sampling rate of the signal from `Fd` to `Fs`, whereas the demapping process decreases the sampling rate from `Fs` to `Fd`.

Functions in this toolbox can perform any of these steps, as summarized in the table below.

Functions for the Steps of Digital Modulation and Demodulation

Step	Function
Mapping and modulation	`dmodce` or `dmod`
Mapping only	`modmap`
Modulation without mapping	`dmodce` or `dmod`, with `/nomap` flag
Demodulation and demapping	`ddemodce` or `ddemod`
Demodulation without demapping (ASK, PSK, or QASK)	`ddemodce` or `ddemod`, with `/nomap` flag
Demapping only	`demodmap`

The functions are described in more detail in the sections that follow.

Representing Digital Signals

This section describes the formats for digital message signals, the analog signals to which they map, and the analog signals that result from the two-stage baseband digital modulation process. The last part, “Constellations and Mapped Signals (PSK, QASK)” on page 2-72, discusses some special formats that apply to the PSK and QASK modulation methods.

Message Signals

To perform M-ary baseband modulation of a digital signal using this toolbox, start with a message signal consisting of integers in the range [0, M-1]. Represent the signal using a vector `x`. Associate with the message signal a sampling rate `Fd`, which means that the entries of `x` give the signal's values in time increments of 1/`Fd`.

Mapped Signals

Mapping produces a real signal `y` whose sampling rate `Fs` must satisfy

```
Fs > Fd
```

(For passband simulation, in which the carrier frequency `Fc` appears explicitly, both of the relations `Fs > Fc > Fd` and `Fs > 2Fc` must hold.) If `x` consists of n samples, then `y` contains n`*Fs/Fd` samples. The actual dimensions of `y` depend on the modulation scheme, as described in "To Map a Digital Signal (General Information)" on page 3-171.

For example, the vector `x` below samples a random digital signal 100 times per second for 2 seconds. The vector `y` represents the mapped signal, sampled three times as frequently. The output shows that `y` contains three times as many samples as `x`.

```
Fd = 100; % Sampling rate of x
M = 32; % Digital symbols are 0,1,2,...,31
x = randint(2*Fd,1,M); % Representation of the digital signal
Fs = 3*Fd; % Sampling rate of mapped signal
y = modmap(x,Fd,Fs,'ask',M); % Mapped signal
r = [size(x,1) size(y,1)] % Number of rows in x and y

r =

   200   600
```

Modulated Signals

Baseband modulation produces a complex signal with sampling rate `Fs`. Notice that this is the same sampling rate as the mapped signal. Baseband signals are explained briefly in "Representing Analog Signals" on page 2-62; for more details, see the works listed in "Selected Bibliography for Modulation" on page 2-81. To illustrate the size and nature of the modulated signal, supplement the example in the paragraph above with these commands.

```
z = dmodce(x,Fd,[Fs pi/2],'ask',M);
whos
  Name      Size            Bytes  Class

  Fd        1x1                 8  double array
  Fs        1x1                 8  double array
  M         1x1                 8  double array
  r         1x2                16  double array
  x       200x1              1600  double array
  y       600x1              4800  double array
```

```
  z       600x1             9600  double array (complex)

Grand total is 1405 elements using 16040 bytes
```

Constellations and Mapped Signals (PSK, QASK)

If you map a digital message using the phase shift keying (PSK) or quadrature amplitude shift keying (QASK) modulation method, then `modmap` describes the amplitude and phase of the resulting analog signal using an in-phase part and a quadrature part. For this reason, one column in the original message signal vector corresponds to *two* columns in the mapped signal matrix.

For example, compare the code below with the example in "Mapped Signals" above. The mapped signal `ypsk` is a two-column matrix, whereas the earlier ASK example produced a column vector. The first column of `ypsk` gives the in-phase components of the samples and the second column gives the quadrature components.

```
Fd = 100; % Sampling rate of x
M = 32; % Digital symbols are 0,1,2,...,31.
x = randint(2*Fd,1,M); % Representation of the digital signal
Fs = 3*Fd; % Sampling rate of mapped signal
ypsk = modmap(x,Fd,Fs,'psk',M); % PSK mapped signal
s = size(ypsk)

s =

   600     2
```

Using Signal Constellation Plots. To understand the in-phase and quadrature description more easily, refer to a signal constellation plot. Each point in the constellation represents an analog signal to which `modmap` can map the digital message data. Each row of `y` in the example above gives the two rectangular coordinates of some point in the constellation. To produce a signal constellation plot that corresponds to the example above, use the command

```
modmap('psk',M) % Using M = 32 from before
```

More about creating signal constellation plots is in the section "Representing Signal Constellations" on page 2-74.

Significance of Sampling Rates

The vectors and matrices that form the input and output of the modulation and demodulation functions do not have a built-in notion of time. That is, MATLAB does not know whether the digital signal `[0 1 2 3 4 5 6 7]` represents an 8-second signal sampled once per second, or a 1-second signal sampled eight times, or something else. However, many functions appearing in this "Modulation" section ask for one or more sampling rates. This subsection discusses the significance of these sampling rates.

If your application has a natural notion of time, then you are free to use it in the modulation and demodulation functions. For example, if you generate the digital signal `[0 1 2 3 4 5 6 7]` and know that it represents a 1-second signal sampled eight times, then set `Fd = 8`. On the other hand, if you know that the signal represents a 2-second signal sampled four times per second, then set `Fd = 4`. You can also use the formula

```
Fd = size(x,1) / (max(t)-min(t)); % if x=signal, t=sample times
```

for a signal `x` sampled at times `t`. Here `x` is a matrix or vector and `t` is a vector whose length is the number of rows of `x`.

For most digital modulation computations, MATLAB does not directly use the sampling rates `Fd` and `Fs` of digital message signals and mapped signals, respectively. What it uses is their *ratio* `Fs/Fd`. For example, the two commands below produce exactly the same result, because 3/1 equals 6/2.

```
y13 = dmodce([0 1 2 3 4 5 6 7]',1,3,'ask',8);
y26 = dmodce([0 1 2 3 4 5 6 7]',2,6,'ask',8);
```

One exceptional situation in which the individual value of `Fd` matters occurs in the MSK and M-ary FSK methods. The default separations between successive frequencies are `Fd/2` and `Fd` for these two methods, respectively.

Choosing Sampling Rates for Passband FSK Modulation

If you use the `dmod` and `ddemod` functions to perform passband FSK modulation, then your choice of the `Fc`, `Fd`, and `Fs` parameters influences the accuracy of the results. The table below lists suggested minimum values for small alphabets. The minimum values yield performance results that match theoretical data to within 0.1 dB when tested using an AWGN channel. To get closer to theoretical

data, you should use values of `Fc` and `Fs` that exceed the minimum values listed in the table.

Alphabet Size, M	Carrier Frequency, Fc	Sampling Rate of Unmodulated Data, Fd	Sampling Rate of Modulated Data, Fs
2	≥8	1	≥8*Fc
4	≥16	1	≥8*Fc
8	≥32	1	≥8*Fc
16	≥64	1	≥8*Fc

Representing Signal Constellations

The QASK method depends on a choice of a signal constellation. The QASK mapping and demapping functions in this toolbox can process two special types of signal constellations, as well as a general type of constellation that you can define as you choose. The special types are called square and circle constellations and the general type is called an arbitrary constellation. This section describes how you can tell MATLAB what signal constellation you want to use, and how you can plot signal constellations.

Square Constellations

To use a square constellation, you only need to tell MATLAB the number of points in the constellation. This number, M, must be a power of two. For example, to map the digital signal [3 8 15 30 28] to a square constellation having 32 points, use the `qaskenco` function as below.

```
[inphase,quadr] = qaskenco([3 8 15 30 28],32);
```

The returned vectors `inphase` and `quadr` give the in-phase and quadrature components, respectively, of the mapped signal. The command

```
msg = qaskdeco(inphase,quadr,32);
```

demaps to recover the original message [3 8 15 30 28]. Notice that in both cases, the square constellation is described only by the number 32.

The modulation and demodulation functions use the M-ary number and the method string `'qask'` to specify the square constellation. The command below

implements QASK modulation on the message [3 8 15 30 28], using a 32-point square constellation. The command assumes that the sampling rates are 1 Hz before modulating and 2 Hz after modulating.

```
y = dmodce([3 8 5 30 28],1,2,'qask',32);
```

Plotting Square Constellations. To plot a square constellation with `M` points, use one of these commands:

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

Circle Constellations

To use a circle constellation having equally spaced points on each circle, you need to give MATLAB this information, in this order:

1. The number of points on each circle
2. The radius of each circle
3. The phase of one point on each circle

The three types of information occupy three vectors of the same length. The first entries of the three vectors determine one circle, the second entries of the three vectors determine another circle, and so on.

For example, the `apkconst` command below returns the complex coordinates of the points on a circle constellation that contains sixteen points on each of two circles. The inner circle has radius one, and one of the constellation points has zero phase. The outer circle has radius three and a constellation point at 10 degrees.

```
y = apkconst([16 16],[1 3],[0 10*pi/180]);
```

The constellation contains two circles because each vector has length two. The constellation has 32 points in total because the sum of entries in the first vector is 32.

The modulation and demodulation functions use three equal-length vectors and the method string `'qask/cir'` to specify the circle constellation. The command below implements QASK modulation on the message [3 8 15 30 28], using the circle constellation described above.

```
y = dmodce([3 8 5 30 28],1,2,'qask/cir',[16 16],[1 3],...
[0 10*pi/180]);
```

Default Values. If you do not provide the phase vector, then by default one constellation point on each circle will have zero phase. If you provide neither the phase vector nor the radius vector, then by default the kth circle will have radius k, and one of the constellation points will have zero phase. You must provide the vector that specifies how many points are on each circle.

Plotting Circle Constellations. To plot a circle constellation in which `numsig` gives the number of points on each circle, `amp` gives the radius of each circle, and `phs` gives the phase of one point on each circle, use one of these commands:

```
apkconst(numsig,amp,phs)
modmap('qask/cir',numsig,amp,phs);
```

To label the constellation points by number, use this syntax instead:

```
apkconst(numsig,amp,phs,'n')
```

Arbitrary Constellations

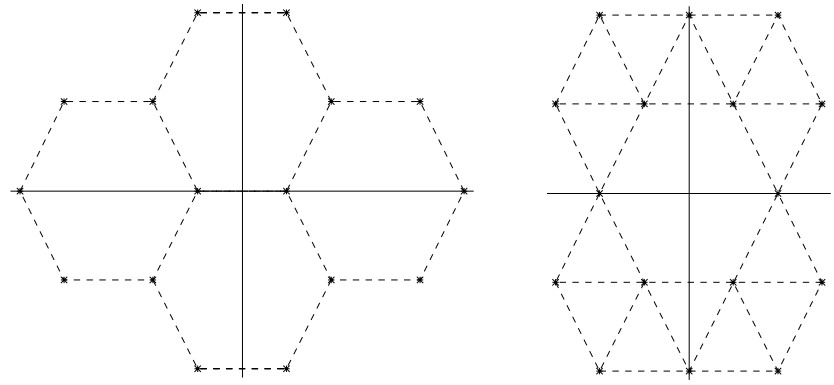
You can also use a signal constellation that does not fit into the categories above. To do this, you need to specify two real vectors of equal length, one that contains the in-phase components of the constellation point and one that contains the corresponding quadrature components. You also need to use the method string `'qask/arb'` in the modulation, demodulation, mapping, and demapping functions.

For example, the code examples below plot signal constellations that have a hexagonal and triangular structure, respectively. They use the `modmap` function.

```
% Example #1: A hexagonal constellation
inphase = [1/2 1 1 1/2 1/2 2 2 5/2];
quadr = [0 1 -1 2 -2 1 -1 0];
inphase = [inphase;-inphase]; inphase = inphase(:);
quadr = [quadr;quadr]; quadr = quadr(:);
modmap('qask/arb',inphase,quadr);

% Example #2: A triangular constellation
figure;
inphase = [1/2 -1/2 1 0 3/2 -3/2 1 -1];
```

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

The figure below shows plots of the hexagonal and triangular signal constellations on the left and right, respectively. The dashed lines are not part of the MATLAB output, and appear below only to suggest the hexagonal and triangular structures.



The modulation and demodulation functions also use the method string `'qask/arb'` and a pair of equal-length vectors like `inphase` and `quadr` to determine your constellation. For example, to modulate the message [3 8 5 10 7] using the QASK method with one of the constellations described in the examples above, supplement the example code with this command:

```
y = dmodce([3 8 5 10 7],1,2,'qask/arb',inphase,quadr);
```

Simple Digital Modulation Example

This example illustrates the basic format of the baseband modulation and demodulation commands, `dmodce` and `ddemodce`. Although the example uses the PSK method, most elements of this example apply to digital modulation techniques other than PSK.

The example generates a random digital signal, modulates it, and adds noise. Then it creates a scatter plot, demodulates the noisy signal, and computes the

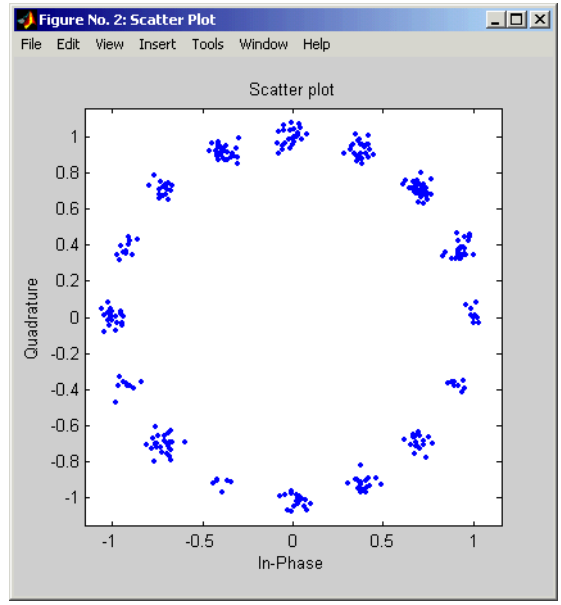
symbol error rate. The `ddemodce` function demodulates the analog signal `y` and then demaps to produce the digital signal `z`.

Notice that the scatter plot does not look exactly like a signal constellation. Whereas the signal constellation would have 16 precisely located points, the noise causes the scatter plot to have a small cluster of points approximately where each constellation point would be. However, the noise is sufficiently small that the signal can be recovered perfectly.

Note Because some options vary by method, you should check the reference pages before adapting the code here for other uses.

Below are the code and the scatter plot.

```
M = 16; % Use 16-ary modulation.
Fd = 1; % Assume the original message is sampled
% at a rate of 1 sample per second.
Fs = 3; % The modulated signal will be sampled
% at a rate of 3 samples per second.
x = randint(100,1,M); % Random digital message
% Use M-ary PSK modulation to produce y.
y = dmodce(x,Fd,Fs,'psk',M);
% Add some Gaussian noise.
ynoisy = y + .04*randn(300,1) + .04*j*randn(300,1);
% Create scatter plot from noisy data.
scatterplot(ynoisy,1,0,'b.');
% Demodulate y to recover the message.
z = ddemodce(ynoisy,Fd,Fs,'psk',M);
s = symerr(x,z) % Check symbol error rate.

s =

     0
```



Customizing the Modulation Process

Recall from “Digital Modulation Overview” on page 2-69 that the modulation and demodulation processes each consist of two steps. You can tell the toolbox functions to carry out only selected steps in the processes. For example, this might be useful if you want to use standard mapping and demapping techniques along with unusual or proprietary modulation and demodulation techniques.

Mapping Without Modulating and Demapping Without Demodulating

To map the digital signal to an analog signal without modulating the analog signal, use the `modmap` function instead of the `dmodce` function. To demap the analog signal to a digital signal without demodulating the analog signal, use the `demodmap` function instead of the `ddemodce` function.

To alter the basic example so that it does not modulate or demodulate the analog signals at all, replace the "old commands" listed in the first column of the table below with the "new commands" listed in the second column.

Changes in "Simple Digital Modulation Example" to Avoid Modulating

Old Command	New Command
`y = dmodce(x,Fd,Fs,'psk',M);`	`y = modmap(x,Fd,Fs,'psk',M);`
`ynoisy = y + .04*randn(300,1) + .04*j*randn(300,1);`	`ynoisy = y + .04*randn(300,2) + .04*j*randn(300,2);`
`z = ddemodce(y,Fd,Fs,'psk',M);`	`z = demodmap(y,Fd,Fs,'psk',M);`

Modulating Without Mapping and Demodulating Without Demapping

To carry out the analog modulation step on a signal that has already been mapped from a digital signal to an analog signal, use the `dmodce` function with the extra word `/nomap` appended to the method string. To carry out the analog demodulation step but avoid demapping the resulting signal to a digital signal, use the `ddemodce` function with the extra word `/nomap` appended to the method string.

If you substituted your own mapping and demapping steps into the basic example then it would look something like the code below. The lines in the second grouping differ from the original example.

```
M = 16; % Use 16-ary modulation.
Fd = 1; % Assume the original message is sampled
% at a rate of 1 sample per second.
Fs = 3; % The modulated signal will be sampled
% at a rate of 3 samples per second.
x = randint(100,1,M); % Random digital message

% Important changes are below.
mapx = mymappingfunction(x); % Use your own function here.
y = dmodce(mapx,Fd,Fs,'psk/nomap',M); % Modulate without mapping.
% Demodulate y without demapping.
demody = ddemodce(y,Fd,Fs,'psk/nomap',M);
% Now demap.
z = mydemappingfunction(demody); % Use your own function here.
```

Other Options in Digital Modulation

The table below lists a few ways in which you might vary the example in "Simple Digital Modulation Example" on page 2-77 in order to perform the modulation and demodulation slightly differently. See the reference pages for full details about options.

Substitutions in the Digital Example

Modification of Process	Modifications in the Code
Set the carrier signal's initial phase to `phs`, measured in radians.	`y = dmodce(x,Fd,[Fs phs],'psk',M);` `z = ddemodce(y,Fd,[Fs phs],'psk',M);`
Use a lowpass filter after demodulating but before demapping. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function.	`z = ddemodce(y,Fd,Fs,'psk',M,num,den);` (See also "Filter Design Issues" on page 2-65 if you plan to use filters.)
(*ASK only*) Use a Costas phase-locked loop.	`y = dmodce(x,Fd,Fs,'ask',M);` `z = ddemodce(y,Fd,Fs,'ask/costas',M);`
(*FSK only*) Use noncoherent demodulation.	`y = dmodce(x,Fd,Fs,'fsk',M);` `z = ddemodce(y,Fd,Fs,'fsk/noncoherence',M);`

Selected Bibliography for Modulation

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

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

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

Special Filters

The Communications Toolbox includes several functions that can help you design and use filters. Other filtering capabilities are in the Signal Processing Toolbox.

Special Filter Features of the Toolbox

Filtering tasks supported in the Communications Toolbox include

- Designing a Hilbert transform filter
- Filtering data using a raised cosine filter
- Designing a raised cosine filter

Besides discussing an implementation issue relating to filters' group delays and some background information about the role of raised cosine filters in communications, this section describes the toolbox functions that accomplish filtering tasks: `hilbiir`, `rcosflt`, `rcosine`, and the lower-level functions `rcosfir` and `rcosiir`.

For more background information about Hilbert filters and raised cosine filters, see the works listed in "Selected Bibliography for Special Filters" on page 2-92. For a demonstration involving raised cosine filters, type `playshow rcosdemo`.

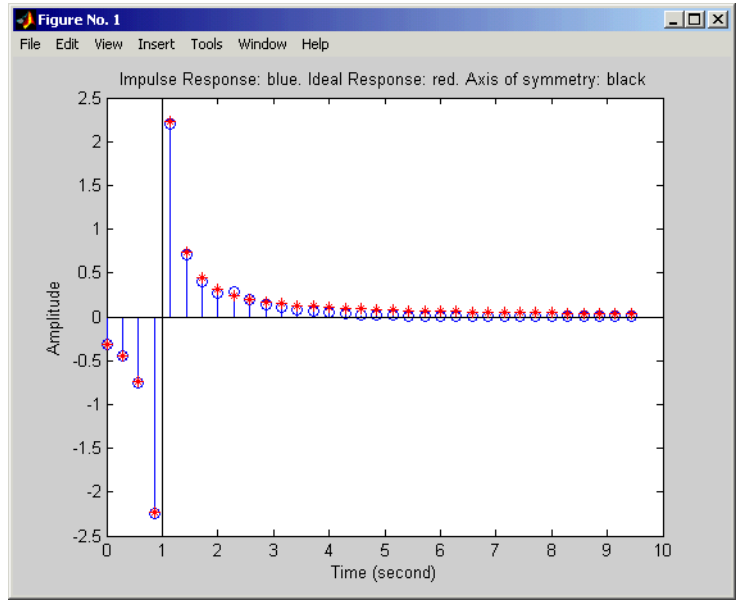
Noncausality and the Group Delay Parameter

Without propagation delays, both Hilbert filters and raised cosine filters are noncausal. This means that the current output depends on the system's future input. In order to design only *realizable* filters, the `hilbiir`, `rcosine`, and `rcosflt` functions delay the input signal before producing an output. This delay, known as the filter's *group delay*, is the time between the filter's initial response and its peak response. The group delay is defined as

$$-\frac{d}{d\omega}\theta(\omega)$$

where θ is the phase of the filter and ω is the frequency in radians. This delay is set so that the impulse response before time zero is negligible and can safely be ignored by the function.

For example, the Hilbert filter whose impulse is shown below uses a group delay of 1 second. Notice in the figure that the impulse response near time 0 is small and that the large impulse response values occur near time 1.



Impulse Response of a Hilbert Filter

Example: Compensating for Group Delays When Analyzing Data

Comparing filtered with unfiltered data might be easier if you delay the unfiltered signal by the filter's group delay. For example, suppose you use the code below to filter `x` and produce `y`.

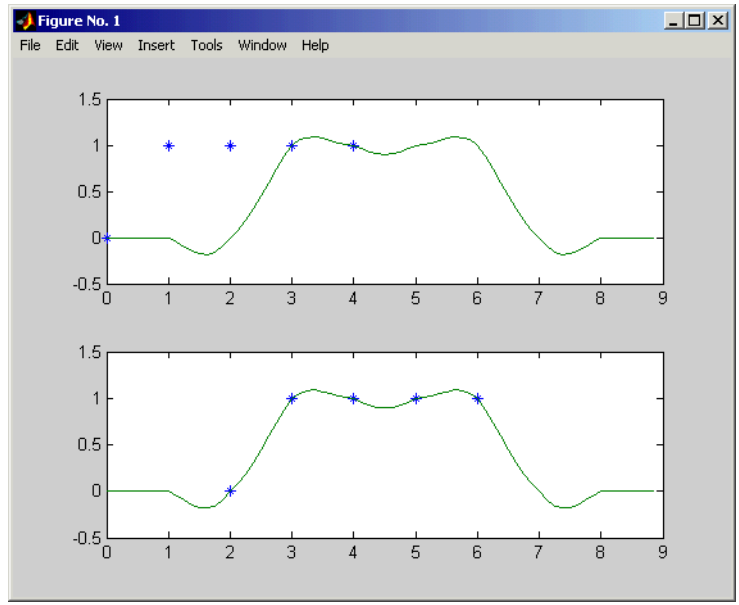
```
tx = 0:4; % Times for data samples
x = [0 1 1 1 1]'; % Binary data samples
% Filter the data and use a delay of 2 seconds.
delay = 2;
[y,ty] = rcosflt(x,1,8,'fir',.3,delay);
```

Here, the elements of `tx` and `ty` represent the times of each sample of `x` and `y`, respectively. However, `y` is delayed relative to `x`, so corresponding elements of

`x` and `y` do not have the same time values. Plotting `y` against `ty` and `x` against `tx` is less useful than plotting `y` against `ty` and `x` against a *delayed version* of `tx`.

```
% Top plot
subplot(2,1,1), plot(tx,x,'*',ty,y);
% Bottom plot delays tx.
subplot(2,1,2), plot(tx+delay,x,'*',ty,y);
```

For another example of compensating for group delay, see the raised-cosine filter demo by typing `playshow rcosdemo`.



Designing Hilbert Transform Filters

The `hilbiir` function designs a Hilbert transform filter and produces either

- A plot of the filter's impulse response
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Example with Default Parameters

For example, typing simply

```
hilbiir
```

plots the impulse response of a fourth-order digital Hilbert transform filter having a 1-second group delay. The sample time is 2/7 seconds. In this particular design, the tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter having a 1-second group delay. The plot is in the figure "Impulse Response of a Hilbert Filter" on page 2-83.

To compute this filter's transfer function, use the command below.

```
[num,den] = hilbiir

num =

   -0.3183   -0.3041   -0.5160   -1.8453    3.3105


den =

    1.0000   -0.4459   -0.1012   -0.0479   -0.0372
```

Here, the vectors `num` and `den` contain the coefficients of the numerator and denominator, respectively, of the transfer function in ascending order of powers of z^{-1}.

The commands in this section use the function's default parameters. You can also control the filter design by specifying the sample time, group delay, bandwidth, and tolerance index. The reference entry for `hilbiir` explains these parameters. The group delay is also mentioned above in "Noncausality and the Group Delay Parameter" on page 2-82.

Raised Cosine Filters in Communication Systems

Raised cosine filters reduce the spectral side lobes of a transmitted signal while introducing controlled intersymbol interference. The interference is controlled in the sense that it exists only at sample times other than the original signal's sample times.

The filtered signal

- Has the same bandwidth as the original signal
- Has a higher sampling rate than the original signal
- Has a spectral rolloff that conforms to the raised cosine spectrum
- Is identical to the original signal at sample times matching those of the original signal

Typically, the receiver decimates the received signal by sampling it precisely at the sample times of the original signal. Because these are times at which the filtered signal is identical to the original signal, the receiver can recover the values of the original signal without intersymbol interference.

Filtering with Raised Cosine Filters

The `rcosflt` function applies a raised cosine filter to data. Because `rcosflt` is a versatile function, you can

- Use `rcosflt` to both design and implement the filter
- Specify a raised cosine filter and use `rcosflt` only to filter the data
- Design and implement either raised cosine filters or square-root raised cosine filters
- Specify the rolloff factor and/or group delay of the filter, if `rcosflt` designs the filter
- Design and implement either FIR or IIR filters

This section discusses the use of sampling rates in filtering and then covers these options. For an additional example, type `playshow rcosdemo` in the MATLAB Command Window.

Sampling Rates

The basic `rcosflt` syntax

```
y = rcosflt(x,Fd,Fs...) % Basic syntax
```

assumes by default that you want to apply the filter to a digital signal `x` whose sampling rate is `Fd`. The filter's sampling rate is `Fs`. The ratio of `Fs` to `Fd` must be an integer. By default, the function upsamples the input data by a factor of `Fs/Fd` before filtering. It upsamples by inserting `Fs/Fd-1` zeros between

consecutive input data samples. The upsampled data consists of `Fs/Fd` samples per symbol and has sampling rate `Fs`.

An example using this syntax is below. The output sampling rate is four times the input sampling rate.

```
y1 = rcosflt([1;0;0],1,4,'fir'); % Upsample by factor of 4/1.
```

Maintaining the Input Sampling Rate. You can also override the default upsampling behavior. In this case, the function assumes that the input signal already has sampling rate `Fs` and consists of `Fs/Fd` samples per symbol. You might want to maintain the sampling rate in a receiver's filter if the corresponding transmitter's filter has already upsampled sufficiently.

To maintain the sampling rate, modify the fourth input argument in `rcosflt` to include the string `Fs`. For example, in the first command below, `rcosflt` uses its default upsampling behavior and the output sampling rate is four times the input sampling rate. By contrast, the second command below uses `Fs` in the string argument and thus maintains the sampling rate throughout.

```
y1 = rcosflt([1;0;0],1,4,'fir'); % Upsample by factor of 4/1.
y2 = rcosflt([1;0;0],1,4,'fir/Fs'); % Maintain sampling rate.
```

The second command assumes that the sampling rate of the input signal is 4, and that the input signal contains 4/1 samples per symbol.

An example that uses the `'Fs'` option at the receiver is in "Combining Two Square-Root Raised Cosine Filters" on page 2-90.

Designing Filters Automatically

The simplest syntax of `rcosflt` assumes that the function should both design and implement the raised cosine filter. For example, the command below designs an FIR raised cosine filter and then filters the input vector [1;0;0] with it. The second and third input arguments indicate that the function should upsample the data by a factor of 8 (that is, 8/1) during the filtering process.

```
y = rcosflt([1;0;0],1,8);
```

Types of Raised Cosine Filters. You can have `rcosflt` design other types of raised cosine filters by using a fourth input argument. Variations on the previous example are below.

```
y = rcosflt([1;0;0],1,8,'fir'); % Same as original example
```

```
y = rcosflt([1;0;0],1,8,'fir/sqrt'); % FIR square-root RC filter
y = rcosflt([1;0;0],1,8,'iir'); % IIR raised cosine filter
y = rcosflt([1;0;0],1,8,'iir/sqrt'); % IIR square-root RC filter
```

Specifying Filters Using Input Arguments

If you have a transfer function for a raised cosine filter, then you can provide it as an input to `rcosflt` so that `rcosflt` does not design its own filter. This is useful if you want to use `rcosine` to design the filter once and then use the filter many times. For example, the `rcosflt` command below uses the `'filter'` flag to indicate that the transfer function is an input argument. The input `num` is a vector that represents the FIR transfer function by listing its coefficients.

```
num = rcosine(1,8); y = rcosflt([1;0;0],1,8,'filter',num);
```

This syntax for `rcosflt` works whether num represents the transfer function for a square-root raised cosine FIR filter or an ordinary raised cosine FIR filter. For example, the code below uses a square-root raised cosine FIR filter. Only the definition of `num` is different.

```
num = rcosine(1,8,'sqrt'); y = rcosflt([1;0;0],1,8,'filter',num);
```

You can also use a raised cosine IIR filter. To do this, modify the fourth input argument of the `rcosflt` command above so that it contains the string `'iir'` and provide a denominator argument. An example is below.

```
delay = 8;
[num,den] = rcosine(1,8,'iir',.5,delay);
y = rcosflt([1;0;0],1,8,'iir/filter',num,den,delay);
```

Controlling the Rolloff Factor

If `rcosflt` designs the filter automatically, then you can control the rolloff factor of the filter, as described below. If you specify your own filter, then `rcosflt` does not need to know its rolloff factor.

The rolloff factor determines the excess bandwidth of the filter. For example, a rolloff factor of .5 means that the bandwidth of the filter is 1.5 times the input sampling frequency, `Fd`. This also means that the transition band of the filter extends from `.5 * Fd` to `1.5 * Fd`.

The default rolloff factor is .5, but if you want to use a value of .2, then you can use a command such as the one below. Typical values for the rolloff factor are between .2 and .5.

```
y = rcosflt([1;0;0],1,8,'fir',.2); % Rolloff factor is .2.
```

Controlling the Group Delay

If `rcosflt` designs the filter automatically, then you can control the group delay of the filter, as described below. If you specify your own FIR filter, then `rcosflt` does not need to know its group delay.

The filter's group delay is the time between the filter's initial response and its peak response. The default group delay in the implementation is three input samples. To specify a different value, measure it in input symbol periods and provide it as the sixth input argument. For example, the command below specifies a group delay of six input samples, which is equivalent to 6 *8 /1 output samples.

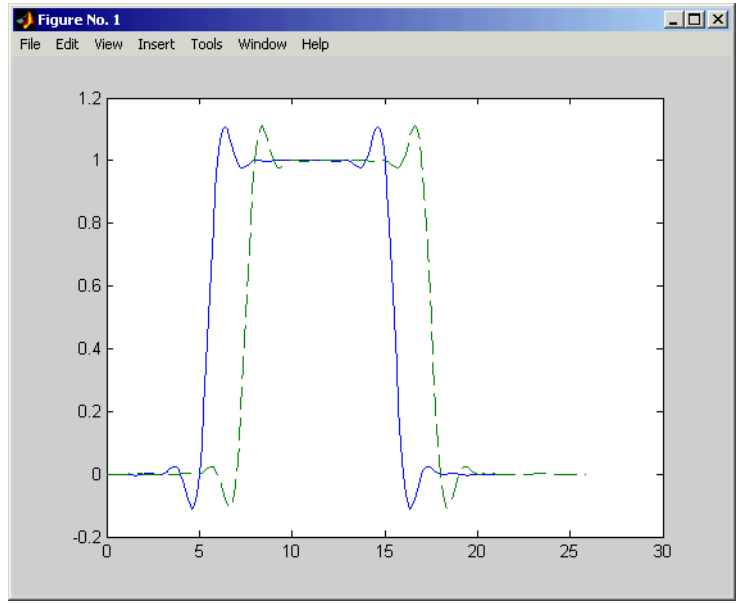
```
y = rcosflt([1;0;0],1,8,'fir',.2,6); % Delay is 6 input samples.
```

The group delay influences the size of the output, as well as the order of the filter if `rcosflt` designs the filter automatically. See the reference page for `rcosflt` for details that relate to the syntax you want to use.

Example: Raised Cosine Filter Delays. The code below filters a signal using two different group delays. A larger delay results in a smaller error in the frequency response of the filter. The plot shows how the two filtered signals differ, and the output `pt` indicates that the first peak occurs at different times for the two filtered signals. In the plot, the solid line corresponds to a delay of six samples, while the dashed line corresponds to a delay of eight samples.

```
[y,t] = rcosflt(ones(10,1),1,8,'fir',.5,6); % Delay = 6 samples
[y1,t1] = rcosflt(ones(10,1),1,8,'fir',.5,8); % Delay = 8 samples
plot(t,y,t1,y1,'--') % Two curves indicate the different delays.
peak = t(find(y == max(y))); % Times where first curve peaks
peak1 = t1(find(y1 == max(y1))); % Times where second curve peaks
pt = [min(peak), min(peak1)] % First peak time for both curves

pt =

   14.6250   16.6250
```

If `Fs/Fd` is at least 4, then a group delay value of at least 8 works well in many cases. In the examples of this section, `Fs/Fd` is 8.



Delays of Six Samples (Solid) and Eight Samples (Dashed)

Combining Two Square-Root Raised Cosine Filters

If you want to split the filtering equally between the transmitter's filter and the receiver's filter, then you can use a pair of square-root raised cosine filters. In theory, the combination of two square-root raised cosine filters is equivalent to a single normal raised cosine filter. However, the limited impulse response of practical square-root raised cosine filters causes a slight difference between the response of two successive square-root raised cosine filters and the response of one raised cosine filter.

Using rcosine and rcosflt to Implement Square-Root Raised Cosine Filters. One way to implement the pair of square-root raised cosine filters is to follow these steps:

1. Use `rcosine` with the `'sqrt'` flag to design a square-root raised cosine filter.
2. Use `rcosflt` in the transmitter section of code to upsample and filter the data.

3 Use `rcosflt` in the receiver section of code to filter the received data *without upsampling* it. Use the `'Fs'` flag to avoid upsampling.

An example of this approach is below. Notice that the syntaxes for `rcosflt` use the `'filter'` flag to indicate that you are providing the filter's transfer function as an input.

```
% First approach
x = randint(100,1,2,1234); % Data
num = rcosine(1,8,'sqrt'); % Transfer function of filter
y = rcosflt(x,1,8,'filter',num); % Filter the data.
z = rcosflt(y,1,8,'Fs/filter',num); % Filter the received data
% but do not upsample it.
```

Using rcosflt Alone. Another way to implement the pair of square-root raised cosine filters is to have `rcosflt` both design and use the square-root raised cosine filter. This approach avoids using `rcosine`. The corresponding example code is below. Notice that the syntaxes for `rcosflt` use the `'sqrt'` flag to indicate that you want it to design a square-root raised cosine filter.

```
% Second approach
x = randint(100,1,2,1234); % Data (again)
y1 = rcosflt(x,1,8,'sqrt'); % Design and use a filter.
z1 = rcosflt(y1,1,8,'sqrt/Fs'); % Design and use a filter
% but do not upsample the data.
```

Because these two approaches are equivalent, `y` is the same as `y1` and `z` is the same as `z1`.

Designing Raised Cosine Filters

The `rcosine` function designs (but does not apply) filters of these types:

- Finite impulse response (FIR) raised cosine filter
- Infinite impulse response (IIR) raised cosine filter
- FIR square-root raised cosine filter
- IIR square-root raised cosine filter

The function returns the transfer function as output. To learn about applying raised cosine filters, see "Filtering with Raised Cosine Filters" on page 2-86.

Sampling Rates

The `rcosine` function assumes that you want to apply the filter to a digital signal whose sampling rate is `Fd`. The function also requires you to provide the filter's sampling rate, `Fs`. The ratio of `Fs` to `Fd` must be an integer.

Example Designing a Square-Root Raised Cosine Filter

For example, the command below designs a square-root raised cosine FIR filter with a sampling rate of 2, for use with a digital signal whose sampling rate is 1.

```
num = rcosine(1,2,'fir/sqrt')

num =

  Columns 1 through 7

   0.0021   -0.0106    0.0300   -0.0531   -0.0750    0.4092    0.8037

  Columns 8 through 13

    0.4092   -0.0750   -0.0531    0.0300   -0.0106    0.0021
```

Here, the vector `num` contains the coefficients of the filter, in ascending order of powers of z^{-1}.

Other Options in Filter Design

You can also control the filter design by specifying the rolloff factor, group delay, and (for IIR filters) tolerance index explicitly, instead of having `rcosine` use its default values. The reference entry for `rcosine` explains these parameters. The group delay is also mentioned above in "Noncausality and the Group Delay Parameter" on page 2-82.

Selected Bibliography for Special Filters

[1] Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

[2] Oppenheim, Alan V., and Ronald W. Schafer, *Discrete-Time Signal Processing*, Englewood Cliffs, N.J., Prentice Hall, 1989.

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

Galois Field Computations

A *Galois field* is an algebraic field that has a finite number of members. This section describes how to work with fields that have 2^m members, where m is an integer between 1 and 16. Such fields are denoted GF(2^m). Galois fields having 2^m members are used in error-control coding. If you need to use Galois fields having an odd number of elements, see "Appendix: Galois Fields of Odd Characteristic" in the online documentation for the Communications Toolbox.

Galois Field Features of the Toolbox

The Communications Toolbox facilitates computations in Galois fields that have 2^m members. You can create array variables whose values are in GF(2^m) and use these variables to perform computations in the Galois field. Most computations use the same syntax that you would use to manipulate ordinary MATLAB arrays of real numbers, making the GF(2^m) capabilities of the toolbox easy to learn and use.

The topics in this section are

For background information about Galois fields or their use in error-control coding, see the works listed in "Selected Bibliography for Galois Fields" on page 2-124.

For more details about specific functions that process arrays of Galois field elements, see the online reference entries in the documentation for MATLAB

or for the Communications Toolbox. Functions whose behavior is identical to the corresponding MATLAB function, except for the ability to handle Galois field members, do not have reference entries in this manual because the entries would be identical to those in the MATLAB manual.

Galois Field Terminology

The discussion of Galois fields in this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [4].

- A *primitive element* of $GF(2^m)$ is a cyclic generator of the group of nonzero elements of $GF(2^m)$. This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power.
- A *primitive polynomial* for $GF(2^m)$ is the minimal polynomial of some primitive element of $GF(2^m)$. That is, it is the binary-coefficient polynomial of smallest nonzero degree having a certain primitive element as a root in $GF(2^m)$. As a consequence, a primitive polynomial has degree m and is irreducible.

The definitions imply that a primitive element is a root of a corresponding primitive polynomial.

Representing Elements of Galois Fields

This section describes how to create a *Galois array*, which is a MATLAB expression that represents elements of a Galois field. This section also describes how MATLAB interprets the numbers that you use in the representation, and includes several examples. The topics are

Creating a Galois Array

To begin working with data from a Galois field GF(`2^m`), you must set the context by associating the data with crucial information about the field. The `gf`

function performs this association and creates a Galois array in MATLAB. This function accepts as inputs

- The Galois field data, `x`, which is a MATLAB array whose elements are integers between 0 and `2^m-1`.
- (*Optional*) An integer, `m`, that indicates that `x` is in the field GF(`2^m`). Valid values of `m` are between 1 and 16. The default is 1, which means that the field is GF(2).
- (*Optional*) A positive integer that indicates which primitive polynomial for GF(`2^m`) you are using in the representations in `x`. If you omit this input argument, then `gf` uses a default primitive polynomial for GF(`2^m`). For information about this argument, see "Specifying the Primitive Polynomial" on page 2-99.

The output of the `gf` function is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate the variable, MATLAB works within the Galois field you have specified. For example, if you apply the `log` function to a Galois array, then MATLAB computes the logarithm in the Galois field and *not* in the field of real or complex numbers.

When MATLAB Implicitly Creates a Galois Array. Some operations on Galois arrays require multiple arguments. If you specify one argument that is a Galois array and another that is an ordinary MATLAB array, then MATLAB interprets both as Galois arrays in the same field. That is, it implicitly invokes the `gf` function on the ordinary MATLAB array. This implicit invocation simplifies your syntax because you can omit some references to the `gf` function. For an example of the simplification, see "Example: Addition and Subtraction" on page 2-103.

Example: Creating Galois Field Variables

The code below creates a row vector whose entries are in the field GF(4), and then adds the row to itself.

```
x = 0:3; % A row vector containing integers
m = 2; % Work in the field GF(2^2), or, GF(4).
a = gf(x,m) % Create a Galois array in GF(2^m).
b = a + a % Add a to itself, creating b.
```

The output is

```
a = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)
```

```
Array elements =

     0     1     2     3


b = GF(2^2) array. Primitive polynomial = D^2+D+1 (7 decimal)

Array elements =

     0     0     0     0
```

The output shows the values of the Galois arrays named `a` and `b`. Notice that each output section indicates

- The field containing the variable, namely, GF(2^2) = GF(4).
- The primitive polynomial for the field. In this case, it is the toolbox's default primitive polynomial for GF(4).
- The array of Galois field values that the variable contains. In particular, the array elements in `a` are exactly the elements of the vector `x`, while the array elements in `b` are four instances of the zero element in GF(4).

The command that creates `b` shows how, having defined the variable `a` as a Galois array, you can add `a` to itself by using the ordinary + operator. MATLAB performs the vectorized addition operation in the field GF(4). Notice from the output that

- Compared to `a`, `b` is in the same field and uses the same primitive polynomial. It is not necessary to indicate the field when defining the sum, `b`, because MATLAB remembers that information from the definition of the addends, `a`.
- The array elements of `b` are zeros because the sum of any value with itself, in a Galois field *of characteristic two*, is zero. This result differs from the sum `x + x`, which represents an addition operation in the infinite field of integers.

Example: Representing Elements of GF(8)

To illustrate what the array elements in a Galois array mean, the table below lists the elements of the field GF(8) as integers and as polynomials in a primitive element, A. The table should help you interpret a Galois array like

```
gf8 = gf([0:7],3); % Galois vector in GF(2^3)
```

Integer Representation	Binary Representation	Element of GF(8)
0	000	0
1	001	1
2	010	A
3	011	A + 1
4	100	A^2
5	101	A^2 + 1
6	110	A^2 + A
7	111	A^2 + A + 1

How Integers Correspond to Galois Field Elements

Building on the GF(8) example above, this section explains the interpretation of array elements in a Galois array in greater generality. The field GF(`2^m`) has `2^m` distinct elements, which this toolbox labels as 0, 1, 2,..., `2^m-1`. These integer labels correspond to elements of the Galois field via a polynomial expression involving a primitive element of the field. More specifically, each integer between 0 and `2^m-1` has a binary representation in `m` bits. Using the bits in the binary representation as coefficients in a polynomial, where the least significant bit is the constant term, leads to a binary polynomial whose order is at most `m-1`. Evaluating the binary polynomial at a primitive element of GF(`2^m`) leads to an element of the field.

Conversely, any element of GF(`2^m`) can be expressed as a binary polynomial of order at most `m-1`, evaluated at a primitive element of the field. The `m`-tuple of coefficients of the polynomial corresponds to the binary representation of an integer between 0 and `2^m`.

Below is a symbolic illustration of the correspondence of an integer X, representable in binary form, with a Galois field element. Each b_k is either zero or one, while A is a primitive element.

$$X = b_{m-1} \cdot 2^{m-1} + \ldots + b_2 \cdot 4 + b_1 \cdot 2 + b_0$$
$$\leftrightarrow b_{m-1}A^{m-1} + \ldots + b_2A^2 + b_1A + b_0$$

Example: Representing a Primitive Element

The code below defines a variable `alph` that represents a primitive element of the field GF(2^4).

```
m = 4; % Or choose any positive integer value of m.
alph = gf(2,m) % Primitive element in GF(2^m)
```

The output is

```
alph = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     2
```

The Galois array `alph` represents a primitive element because of the correspondence between

- The integer 2, specified in the `gf` syntax
- The binary representation of 2, which is 10 (or 0010 using four bits)
- The polynomial A + 0, where A is a primitive element in this field (or $0A^3 + 0A^2 + A + 0$ using the four lowest powers of A)

Primitive Polynomials and Element Representations

This section builds on the discussion in "Representing Elements of Galois Fields" on page 2-94 by describing how to specify your own primitive polynomial when you create a Galois array. The topics are

- "Specifying the Primitive Polynomial" on page 2-99
- "Finding Primitive Polynomials" on page 2-100
- "Effect of Nondefault Primitive Polynomials on Numerical Results" on page 2-101

If you perform many computations using a nondefault primitive polynomial, then see "Speed and Nondefault Primitive Polynomials" on page 2-123 as well.

Specifying the Primitive Polynomial

The discussion in "How Integers Correspond to Galois Field Elements" on page 2-97 refers to a primitive element, which is a root of a primitive polynomial of the field. When you use the `gf` function to create a Galois array, the function interprets the integers in the array with respect to a specific default primitive polynomial for that field, unless you explicitly provide a different primitive polynomial. A list of the default primitive polynomials is on the reference page for the `gf` function.

To specify your own primitive polynomial when creating a Galois array, use a syntax like

```
c = gf(5,4,25) % 25 indicates the primitive polynomial for GF(16).
```

instead of

```
c1= gf(5,4); % Use default primitive polynomial for GF(16).
```

The extra input argument, `25` in this case, specifies the primitive polynomial for the field GF(`2^m`) in a way similar to the representation described in "How Integers Correspond to Galois Field Elements" on page 2-97. In this case, the integer 25 corresponds to a binary representation of 11001, which in turn corresponds to the polynomial $D^4 + D^3 + 1$.

Note When you specify the primitive polynomial, the input argument must have a binary representation using exactly `m+1` bits, not including unnecessary leading zeros. In other words, a primitive polynomial for GF(`2^m`) always has order `m`.

When you use an input argument to specify the primitive polynomial, the output reflects your choice by showing the integer value as well as the polynomial representation.

```
d = gf([1 2 3],4,25)

d = GF(2^4) array. Primitive polynomial = D^4+D^3+1 (25 decimal)

Array elements =

     1     2     3
```

Note After you have defined a Galois array, you cannot change the primitive polynomial with respect to which MATLAB interprets the array elements.

Finding Primitive Polynomials

You can use the `primpoly` function to find primitive polynomials for GF(`2^m`) and the `isprimitive` function to determine whether a polynomial is primitive for GF(`2^m`). The code below illustrates.

```
m = 4;
defaultprimpoly = primpoly(m) % Default primitive poly for GF(16)

Primitive polynomial(s) =

D^4+D^1+1

defaultprimpoly =

    19

allprimpolys = primpoly(m,'all') % All primitive polys for GF(16)

Primitive polynomial(s) =

D^4+D^1+1
D^4+D^3+1

allprimpolys =

    19
    25

i1 = isprimitive(25) % Can 25 be the prim_poly input in gf(...)?

i1 =

     1
```

```
i2 = isprimitive(21) % Can 21 be the prim_poly input in gf(...)?

i2 =

     0
```

Effect of Nondefault Primitive Polynomials on Numerical Results

Most fields offer multiple choices for the primitive polynomial that helps define the representation of members of the field. When you use the `gf` function, changing the primitive polynomial changes the interpretation of the array elements and, in turn, changes the results of some subsequent operations on the Galois array. For example, exponentiation of a primitive element makes it easy to see how the primitive polynomial affects the representations of field elements.

```
a11 = gf(2,3); % Use default primitive polynomial of 11.
a13 = gf(2,3,13); % Use D^3+D^2+1 as the primitive polynomial.
z = a13.^3 + a13.^2 + 1 % 0 because a13 satisfies the equation
nz = a11.^3 + a11.^2 + 1 % Nonzero. a11 does not satisfy equation.
```

The output below shows that when the primitive polynomial has integer representation `13`, the Galois array satisfies a certain equation. By contrast, when the primitive polynomial has integer representation `11`, the Galois array fails to satisfy the equation.

```
z = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     0


nz = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     6
```

The output when you try this example might also include a warning about lookup tables. This is normal if you did not use the `gftable` function to optimize computations involving a nondefault primitive polynomial of 13.

Arithmetic in Galois Fields

You can perform arithmetic operations on Galois arrays by using the same MATLAB operators that work on ordinary integer arrays. The table below lists the available arithmetic operations as well as the operators that perform them. Whenever you operate on a pair of Galois arrays, both arrays must be in the same Galois field.

Operation	Operator
Addition	`+`
Subtraction	`-`
Elementwise multiplication	`.*`
Matrix multiplication	`*`
Elementwise left division	`./`
Elementwise right division	`.\`
Matrix left division	`/`
Matrix right division	`\`
Elementwise exponentiation	`.^`
Elementwise logarithm	`log()`
Exponentiation of a square Galois matrix by a scalar integer	`^`

Note For multiplication and division of polynomials over a Galois field, see "Addition and Subtraction of Polynomials" on page 2-117.

Examples of these operations are in the sections that follow:

- "Example: Addition and Subtraction" on page 2-103
- "Example: Multiplication" on page 2-104
- "Example: Division" on page 2-105

Example: Addition and Subtraction

The code below adds two Galois arrays to create an addition table for GF(8). Addition uses the ordinary + operator. The code below also shows how to index into the array `addtb` to find the result of adding 1 to the elements of GF(8).

```
m = 3;
e = repmat([0:2^m-1],2^m,1);
f = gf(e,m); % Create a Galois array.
addtb = f + f' % Add f to its own matrix transpose.

addtb = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     0     1     2     3     4     5     6     7
     1     0     3     2     5     4     7     6
     2     3     0     1     6     7     4     5
     3     2     1     0     7     6     5     4
     4     5     6     7     0     1     2     3
     5     4     7     6     1     0     3     2
     6     7     4     5     2     3     0     1
     7     6     5     4     3     2     1     0

addone = addtb(2,:); % Assign 2nd row to the Galois vector addone.
```

As an example of reading this addition table, the (7,4) entry in the `addtb` array shows that `gf(6,3)` plus `gf(3,3)` equals `gf(5,3)`. Equivalently, the element A^2+A plus the element A+1 equals the element A^2+1. The equivalence arises from the binary representation of 6 as 110, 3 as 011, and 5 as 101.

The subtraction table, which you can obtain by replacing + by -, would be the same as `addtb`. This is because subtraction and addition are identical operations in a field *of characteristic two*. In fact, the zeros along the main diagonal of `addtb` illustrate this fact for GF(8).

Simplifying the Syntax. The code below illustrates scalar expansion and the implicit creation of a Galois array from an ordinary MATLAB array. The Galois arrays `h` and `h1` are identical, but the creation of `h` uses a simpler syntax.

```
g = gf(ones(2,3),4); % Create a Galois array explicitly.
h = g + 5; % Add gf(5,4) to each element of g.
h1 = g + gf(5*ones(2,3),4) % Same as h.

h1 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     4     4     4
     4     4     4
```

Notice that 1+5 is reported as 4 in the Galois field. This is true because the 5 represents the polynomial expression A^2+1, and $1+(A^2+1)$ in GF(16) is A^2. Furthermore, the integer that represents the polynomial expression A^2 is 4.

Example: Multiplication

The example below multiplies individual elements in a Galois array using the `.*` operator. It then performs matrix multiplication using the `*` operator. The elementwise multiplication produces an array whose size matches that of the inputs. By contrast, the matrix multiplication produces a Galois scalar because it is the matrix product of a row vector with a column vector.

```
m = 5;
row1 = gf([1:2:9],m); row2 = gf([2:2:10],m);
col = row2'; % Transpose to create a column array.
ep = row1 .* row2; % Elementwise product.
mp = row1 * col; % Matrix product.
```

Multiplication Table for GF(8). As another example, the code below multiplies two Galois vectors using matrix multiplication. The result is a multiplication table for GF(8).

```
m = 3;
els = gf([0:2^m-1]',m);
multb = els * els' % Multiply els by its own matrix transpose.

multb = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)
```

```
Array elements =

     0     0     0     0     0     0     0     0
     0     1     2     3     4     5     6     7
     0     2     4     6     3     1     7     5
     0     3     6     5     7     4     1     2
     0     4     3     7     6     2     5     1
     0     5     1     4     2     7     3     6
     0     6     7     1     5     3     2     4
     0     7     5     2     1     6     4     3
```

Example: Division

The examples below illustrate the four division operators in a Galois field by computing multiplicative inverses of individual elements and of an array. You can also compute inverses using `inv` or using exponentiation by -1.

Elementwise Division. This example divides 1 by each of the individual elements in a Galois array using the `./` and `.\` operators. These two operators differ only in their sequence of input arguments. Each quotient vector lists the multiplicative inverses of the nonzero elements of the field. In this example, MATLAB expands the scalar 1 to the size of `nz` before computing; alternatively, you can use as arguments two arrays of the same size.

```
m = 5;
nz = gf([1:2^m-1],m); % Nonzero elements of the field
inv1 = 1 ./ nz; % Divide 1 by each element.
inv2 = nz .\ 1; % Obtain same result using .\ operator.
```

Matrix Division. This example divides the identity array by the square Galois array `mat` using the `/` and `\` operators. Each quotient matrix is the multiplicative inverse of `mat`. Notice how the transpose operator (`'`) appears in the equivalent operation using `\`. For square matrices, the sequence of transpose operations is unnecessary, but for nonsquare matrices, it is necessary.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minv1 = eye(3) / mat; % Compute matrix inverse.
minv2 = (mat' \ eye(3)')'; % Obtain same result using \ operator.
```

Example: Exponentiation

The examples below illustrate how to compute integer powers of a Galois array. To perform matrix exponentiation on a Galois array, you must use a square Galois array as the base and an ordinary (not Galois) integer scalar as the exponent.

Elementwise Exponentiation. This example computes powers of a primitive element, A, of a Galois field. It then uses these separately computed powers to evaluate the default primitive polynomial at A. The answer of zero shows that A is a root of the primitive polynomial. Notice that the `.^` operator exponentiates each array element independently.

```
m = 3;
av = gf(2*ones(1,m+1),m); % Row containing primitive element
expa = av .^ [0:m]; % Raise element to different powers.
evp = expa(4)+expa(2)+expa(1) % Evaluate D^3 + D + 1.

evp = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     0
```

Matrix Exponentiation. This example computes the inverse of a square matrix by raising the matrix to the power -1. It also raises the square matrix to the powers 2 and -2.

```
m = 5;
mat = gf([1 2 3; 4 5 6; 7 8 9],m);
minvs = mat ^ (-1); % Matrix inverse
matsq = mat^2; % Same as mat * mat
matinvssq = mat^(-2); % Same as minvs * minvs
```

Example: Elementwise Logarithm

The code below computes the logarithm of the elements of a Galois array. The output indicates how to express each *nonzero* element of GF(8) as a power of the primitive element. The logarithm of the zero element of the field is undefined.

```
gf8_nonzero = gf([1:7],3); % Vector of nonzero elements of GF(8)
expformat = log(gf8_nonzero) % Logarithm of each element
```

```
expformat =

     0     1     3     2     6     4     5
```

As an example of how to interpret the output, consider the last entry in each vector in this example. You can infer that the element `gf(7,3)` in GF(8) can be expressed as either

- A^5, using the last element of `expformat`
- A^2+A+1, using the binary representation of 7 as 111. See "Example: Representing Elements of GF(8)" on page 2-96 for more details.

Logical Operations in Galois Fields

You can apply logical tests to Galois arrays and obtain a logical array. Some important types of tests are testing for equality of two Galois arrays and testing for nonzero values in a Galois array.

Testing for Equality

To compare corresponding elements of two Galois arrays that have the same size, use the operators == and ~=. The result is a logical array, each element of which indicates the truth or falsity of the corresponding elementwise comparison. If you use the same operators to compare a scalar with a Galois array, then MATLAB compares the scalar with each element of the array, producing a logical array of the same size.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg1 = (r1 .* r2 == [1 1 1]) % Does each element equal one?
lg2 = (r1 .* r2 == 1) % Same as above, using scalar expansion
lg3 = (r1 ~= r2) % Does each element differ from its inverse?
```

The output is below.

```
lg1 =

     1     1     1
```

```
lg2 =

     1     1     1


lg3 =

     0     1     1
```

Comparison of isequal and ==. To compare entire arrays and obtain a logical *scalar* result rather than a logical array, you can use the built-in `isequal` function. Note, however, that `isequal` uses strict rules for its comparison, and returns a value of `0` (false) if you compare

- A Galois array with an ordinary MATLAB array, even if the values of the underlying array elements match
- A scalar with a nonscalar array, even if all elements in the array match the scalar

The example below illustrates this difference between == and `isequal`.

```
m = 5; r1 = gf([1:3],m); r2 = 1 ./ r1;
lg4 = isequal(r1 .* r2, [1 1 1]); % False
lg5 = isequal(r1 .* r2, gf(1,m)); % False
lg6 = isequal(r1 .* r2, gf([1 1 1],m)); % True
```

Testing for Nonzero Values

To test for nonzero values in a Galois vector, or in the columns of a Galois array that has more than one row, use the `any` or `all` function. These two functions behave just like the ordinary MATLAB functions `any` and `all`, except that they consider only the underlying array elements while ignoring information about which Galois field the elements are in. Examples are below.

```
m = 3; randels = gf(randint(6,1,2^m),m);
if all(randels) % If all elements are invertible
    invels = randels .\ 1; % Compute inverses of elements.
else
    disp('At least one element was not invertible.');
end

alph = gf(2,4);
```

```
poly = 1 + alph + alph^3;
if any(poly) % If poly contains a nonzero value
    disp('alph is not a root of 1 + D + D^3.');
end

code = rsenc(gf([0:4;3:7],3),7,5); % Each row is a code word.
if all(code,2) % Is each row entirely nonzero?
    disp('Both code words are entirely nonzero.');
else
    disp('At least one code word contains a zero.');
end
```

Matrix Manipulation in Galois Fields

Some basic operations that you would perform on an ordinary MATLAB array are available for Galois arrays. This section illustrates how to perform basic manipulations and how to get basic information.

Basic Manipulations of Galois Arrays

Basic array operations that you can perform on a Galois array include those in the table below. The results of these operations are Galois arrays in the same field. The functionality of these operations is analogous to the MATLAB operations having the same syntax.

Operation	Syntax
Index into array, possibly using colon operator instead of a vector of explicit indices	`a(vector)` or `a(vector,vector1)`, where `vector` and/or `vector1` can be ":" instead of a vector
Transpose array	`a'`
Concatenate matrices	`[a,b]` or `[a;b]`
Create array having specified diagonal elements	`diag(vector)` or `diag(vector,k)`
Extract diagonal elements	`diag(a)` or `diag(a,k)`
Extract lower triangular part	`tril(a)` or `tril(a,k)`

Operation (Continued)	Syntax (Continued)
Extract upper triangular part	`triu(a)` or `triu(a,k)`
Change shape of array	`reshape(a,k1,k2)`

The code below uses some of these syntaxes.

```
m = 4; a = gf([0:15],m);
a(1:2) = [13 13]; % Replace some elements of the vector a.
b = reshape(a,2,8); % Create 2-by-8 matrix.
c = [b([1 1 2],1:3); a(4:6)]; % Create 4-by-3 matrix.
d = [c, a(1:4)']; % Create 4-by-4 matrix.
dvec = diag(d); % Extract main diagonal of d.
dmat = diag(a(5:9)); % Create 5-by-5 diagonal matrix
dtril = tril(d); % Extract upper and lower triangular
dtriu = triu(d); % parts of d.
```

Basic Information About Galois Arrays

You can determine the length of a Galois vector or the size of any Galois array using the `length` and size functions. The functionality for Galois arrays is analogous to that of the MATLAB operations on ordinary arrays, except that the output arguments from `size` and `length` are always integers, not Galois arrays. The code below illustrates the use of these functions.

```
m = 4; e = gf([0:5],m); f = reshape(e,2,3);
lne = length(e); % Vector length of e
szf = size(f); % Size of f, returned as a two-element row
[nr,nc] = size(f); % Size of f, returned as two scalars
nc2 = size(f,2); % Another way to compute number of columns
```

Positions of Nonzero Elements. Another type of information you might want to determine from a Galois array is the positions of nonzero elements. For an ordinary MATLAB array, you might use the `find` function. However, for a Galois array you should use `find` in conjunction with the ~= operator, as illustrated.

```
x = [0 1 2 1 0 2]; m = 2; g = gf(x,m);
nzx = find(x); % Find nonzero values in the ordinary array x.
nzg = find(g~=0); % Find nonzero values in the Galois array g.
```

Linear Algebra in Galois Fields

You can do linear algebra in a Galois field using Galois arrays. Important categories of computations are inverting matrices, computing determinants, computing ranks, factoring square matrices, and solving linear equations.

Inverting Matrices and Computing Determinants

To invert a square Galois array, use the `inv` function. Related is the `det` function, which computes the determinant of a Galois array. Both `inv` and `det` behave like their ordinary MATLAB counterparts, except that they perform computations in the Galois field instead of in the field of complex numbers.

Note A Galois array is singular if and only if its determinant is exactly zero. It is not necessary to consider roundoff errors, as in the case of real and complex arrays.

The code below illustrates matrix inversion and determinant computation.

```
m = 4;
randommatrix = gf(randint(4,4,2^m),m);
gfid = gf(eye(4),m);
if det(randommatrix) ~= 0
    invmatrix = inv(randommatrix);
    check1 = invmatrix * randommatrix;
    check2 = randommatrix * invmatrix;
    if (isequal(check1,gfid) & isequal(check2,gfid))
        disp('inv found the correct matrix inverse.');
    end
else
    disp('The matrix is not invertible.');
end
```

The output from this example is either of these two messages, depending on whether the randomly generated matrix is nonsingular or singular.

```
inv found the correct matrix inverse.
```

```
The matrix is not invertible.
```

Computing Ranks

To compute the rank of a Galois array, use the `rank` function. It behaves like the ordinary MATLAB `rank` function when given exactly one input argument. The example below illustrates how to find the rank of square and nonsquare Galois arrays.

```
m = 3;
asquare = gf([4 7 6; 4 6 5; 0 6 1],m);
r1 = rank(asquare);
anonsquare = gf([4 7 6 3; 4 6 5 1; 0 6 1 1],m);
r2 = rank(anonsquare);
[r1 r2]

ans =

     2     3
```

The values of `r1` and `r2` indicate that `asquare` has less than full rank but that `anonsquare` has full rank.

Factoring Square Matrices

To express a square Galois array (or a permutation of it) as the product of a lower triangular Galois array and an upper triangular Galois array, use the `lu` function. This function accepts one input argument and produces exactly two or three output arguments. It behaves like the ordinary MATLAB `lu` function when given the same syntax. The example below illustrates how to factor using `lu`.

```
tofactor = gf([6 5 7 6; 5 6 2 5; 0 1 7 7; 1 0 5 1],3);
[L,U]=lu(tofactor); % lu with two output arguments
c1 = isequal(L*U, tofactor) % True
tofactor2 = gf([1 2 3 4;1 2 3 0;2 5 2 1; 0 5 0 0],3);
[L2,U2,P] = lu(tofactor2); % lu with three output arguments
c2 = isequal(L2*U2, P*tofactor2) % True
```

Solving Linear Equations

To find a particular solution of a linear equation in a Galois field, use the `\` or `/` operator on Galois arrays. The table below indicates the equation that each

operator addresses, assuming that `A` and `B` are previously defined Galois arrays.

	Backslash Operator (\)	**Slash Operator (/)**
Linear Equation	`A * x = B`	`x * A = B`
Syntax	`x = A \ B`	`x = B / A`
Equivalent Syntax Using `\`	Not applicable	`x = (A' \ B')'`

The results of the syntax in the table depend on characteristics of the Galois array A:

- If `A` is square and nonsingular, then the output `x` is the unique solution to the linear equation.
- If `A` is square and singular, then the syntax in the table produces an error.
- If `A` is not square, then MATLAB attempts to find a particular solution. If `A'*A` or `A*A'` is a singular array, or if `A` is a tall matrix that represents an overdetermined system, then the attempt might fail.

Note An error message does not necessarily indicate that the linear equation has no solution. You might be able to find a solution by rephrasing the problem. For example, `gf([1 2; 0 0],3) \ gf([1; 0],3)` produces an error but the mathematically equivalent `gf([1 2],3) \ gf([1],3)` does not. The first syntax fails because `gf([1 2; 0 0],3)` is a singular square matrix.

Example: Solving Linear Equations. The examples below illustrate how to find particular solutions of linear equations over a Galois field.

```
m = 4;
A = gf(magic(3),m); % Square nonsingular matrix
Awide=[A, 2*A(:,3)]; % 3-by-4 matrix with redundancy on the right
Atall = Awide'; % 4-by-3 matrix with redundancy at the bottom
B = gf([0:2]',m);
C = [B; 2*B(3)];
D = [B; B(3)+1];
```

```
thesolution = A \ B; % Solution of A * x = B
thesolution2 = B' / A; % Solution of x * A = B'
ck1 = all(A * thesolution == B) % Check validity of solutions.
ck2 = all(thesolution2 * A == B')

% Awide * x = B has infinitely many solutions. Find one.
onesolution = Awide \ B;
ck3 = all(Awide * onesolution == B) % Check validity of solution.

% Atall * x = C has a solution.
asolution = Atall \ C;
ck4 = all(Atall * asolution == C) % Check validity of solution.

% Atall * x = D has no solution.
notasolution = Atall \ D;
ck5 = all(Atall * notasolution == D) % It is not a valid solution.
```

The output from this example indicates that the validity checks are all true (`1`), except for `ck5`, which is false (`0`).

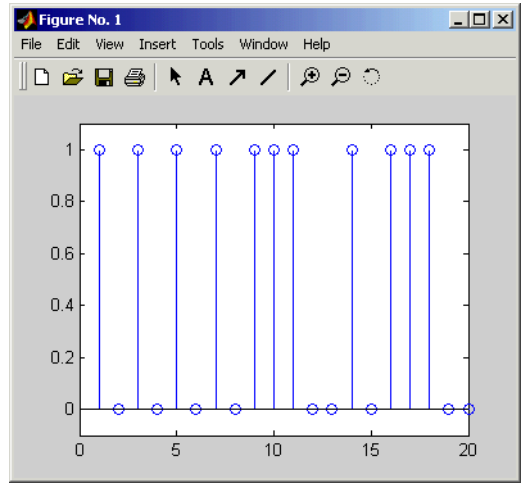
Signal Processing Operations in Galois Fields

You can perform some signal-processing operations on Galois arrays, such as filtering, convolution, and the discrete Fourier transform. This section describes how to perform these operations. Other information about the corresponding operations for ordinary real vectors is in the Signal Processing Toolbox documentation.

Filtering

To filter a Galois vector, use the `filter` function. It behaves like the ordinary MATLAB `filter` function when given exactly three input arguments. The code and diagram below give the impulse response of a particular filter over GF(2).

```
m = 1; % Work in GF(2).
b = gf([1 0 0 1 0 1 0 1],m); % Numerator
a = gf([1 0 1 1],m); % Denominator
x = gf([1,zeros(1,19)],m);
y = filter(b,a,x); % Filter x.
figure; stem(y.x); % Create stem plot.
axis([0 20 -.1 1.1])
```



Convolution

This toolbox offers two equivalent ways to convolve a pair of Galois vectors:

- Use the `conv` function, as described in "Multiplication and Division of Polynomials" on page 2-117. This works because convolving two vectors is equivalent to multiplying the two polynomials whose coefficients are the entries of the vectors.
- Use the `convmtx` function to compute the convolution matrix of one of the vectors, and then multiply that matrix by the other vector. This works because convolving two vectors is equivalent to filtering one of the vectors by the other. The equivalence permits the representation of a digital filter as a convolution matrix, which you can then multiply by any Galois vector of appropriate length.

Tip If you need to convolve large Galois vectors, then multiplying by the convolution matrix might be faster than using `conv`.

Example. The example below computes the convolution matrix for a vector `b` in GF(4), representing the numerator coefficients for a digital filter. It then illustrates the two equivalent ways to convolve `b` with `x` over the Galois field.

```
m = 2; b = gf([1 2 3]',m);
n = 3; x = gf(randint(n,1,2^m),m);
C = convmtx(b,n); % Compute convolution matrix.
v1 = conv(b,x); % Use conv to convolve b with x
v2 = C*x; % Use C to convolve b with x.
```

Discrete Fourier Transform

The discrete Fourier transform is an important tool in digital signal processing. This toolbox offers these tools to help you process discrete Fourier transforms:

- `fft`, which transforms a Galois vector
- `ifft`, which inverts the discrete Fourier transform on a Galois vector
- `dftmtx`, which returns a Galois array that you can use to perform or invert the discrete Fourier transform on a Galois vector

In all cases, the vector being transformed must be a Galois vector of length 2^m-1 in the field GF(2^m). The examples below illustrate the use of these functions. You can check, using the `isequal` function, that `y` equals `y1`, `z` equals `z1`, and `z` equals `x`.

```
m = 4;
x = gf(randint(2^m-1,1,2^m),m); % A vector to transform
alph = gf(2,m);
dm = dftmtx(alph);
idm = dftmtx(1/alph);
y = dm*x; % Transform x using the result of dftmtx.
y1 = fft(x); % Transform x using fft.
z = idm*y; % Recover x using the result of dftmtx(1/alph).
z1 = ifft(y1); % Recover x using ifft.
```

Tip If you have many vectors that you want to transform (in the same field), then it might be faster to use `dftmtx` once and matrix multiplication many times, instead of using `fft` many times.

Polynomials over Galois Fields

You can use Galois vectors to represent polynomials in an indeterminate quantity x, with coefficients in a Galois field. Form the representation by

listing the coefficients of the polynomial in a vector in order of descending powers of x. For example, the vector

```
gf([2 1 0 3],4)
```

represents the polynomial $Ax^3 + 1x^2 + 0x + (A+1)$, where

- A is a primitive element in the field $GF(2^4)$.
- x is the indeterminate quantity in the polynomial.

You can then use such a Galois vector to perform arithmetic with, evaluate, and find roots of polynomials. You can also find minimal polynomials of elements of a Galois field.

Addition and Subtraction of Polynomials

To add and subtract polynomials, use the ordinary + and - operators on equal-length Galois vectors that represent the polynomials. If one polynomial has lower degree than the other, then you must pad the shorter vector with zeros at the beginning so that the two vectors have the same length. The example below shows how to add a degree-one polynomial in x to a degree-two polynomial in x.

```
lin = gf([4 2],3); % A^2 x + A, which is linear in x
linpadded = gf([0 4 2],3); % The same polynomial, zero-padded
quadr = gf([1 4 2],3); % x^2 + A^2 x + A, which is quadratic in x
% Can't do lin + quadr because they have different vector lengths.
sumpoly = [0, lin] + quadr; % Sum of the two polynomials
sumpoly2 = linpadded + quadr; % The same sum
```

Multiplication and Division of Polynomials

To multiply and divide polynomials, use the `conv` and `deconv` functions on Galois vectors that represent the polynomials. Multiplication and division of polynomials is equivalent to convolution and deconvolution of vectors. The `deconv` function returns the quotient of the two polynomials as well as the remainder polynomial. Examples are below.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
bpoly = gf([1 1],m); % x + 1
xpoly = gf([1 0],m); % x
% Product is A^2 x^3 + x^2 + (A^2 + A) x + (A + 1).
```

```
cpoly = conv(apoly,bpoly);
[a2,remd] = deconv(cpoly,bpoly); % a2==apoly. remd is zero.
[otherpol,remd2] = deconv(cpoly,xpoly); % remd is nonzero.
```

Note that the multiplication and division operators described in "Arithmetic in Galois Fields" on page 2-102 multiply elements or matrices, but not polynomials.

Evaluating Polynomials

To evaluate a polynomial at an element of a Galois field, use the `polyval` function. It behaves like the ordinary MATLAB `polyval` function when given exactly two input arguments. The example below illustrates how to evaluate a polynomial at several elements in a field. It also checks the results using the operators `.^` and `.*` in the field.

```
m = 4;
apoly = gf([4 5 3],m); % A^2 x^2 + (A^2 + 1) x + (A + 1)
x0 = gf([0 1 2],m); % Points at which to evaluate the polynomial
y = polyval(apoly,x0)

y = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10

a = gf(2,m); % Primitive element of the field, corresponding to A.
y2 = a.^2.*x0.^2 + (a.^2+1).*x0 + (a+1) % Check the result.

y2 = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     3     2    10
```

The first element of `y` evaluates the polynomial at `0` and, therefore, returns the polynomial's constant term of `3`.

Roots of Polynomials

To find the roots of a polynomial in a Galois field, use the `roots` function on a Galois vector that represents the polynomial. This function finds roots that are in the same field that the Galois vector is in. The number of times an entry appears in the output vector from `roots` is exactly its multiplicity as a root of the polynomial.

Note If the Galois vector is in GF(2^m), then the polynomial it represents might have additional roots in some extension field GF($(2^m)^k$). However, `roots` does not find those additional roots or indicate their existence.

The examples below find roots of cubic polynomials in GF(8).

```
m = 3;
cubicpoly1 = gf([2 7 3 0],m); % A polynomial divisible by x
cubicpoly2 = gf([2 7 3 1],m);
cubicpoly3 = gf([2 7 3 2],m);
zeroandothers = roots(cubicpoly1); % Zero is among the roots.
multipleroots = roots(cubicpoly2); % One root has multiplicity 2.
oneroot = roots(cubicpoly3); % Only one root is in GF(2^m).
```

Roots of Binary Polynomials

In the special case of a polynomial having binary coefficients, it is also easy to find roots that exist in an extension field. This because the elements `0` and `1` have the same unambiguous representation in all fields of characteristic two. To find roots of a binary polynomial in an extension field, apply the `roots` function to a Galois vector in the extension field whose array elements are the binary coefficients of the polynomial.

The example below seeks roots of a binary polynomial in various fields.

```
gf2poly = gf([1 1 1],1); % x^2 + x + 1 in GF(2)
noroots = roots(gf2poly); % No roots in the ground field, GF(2)
gf4poly = gf([1 1 1],2); % x^2 + x + 1 in GF(4)
roots4 = roots(gf4poly); % The roots are A and A+1, in GF(4).
gf16poly = gf([1 1 1],4); % x^2 + x + 1 in GF(16)
roots16 = roots(gf16poly); % Roots in GF(16)
checkanswer4 = polyval(gf4poly,roots4); % Zero vector
checkanswer16 = polyval(gf16poly,roots16); % Zero vector
```

The roots of the polynomial do not exist in GF(2), so `noroots` is an empty array. However, the roots of the polynomial exist in GF(4) as well as in GF(16), so `roots4` and `roots16` are nonempty.

Notice that `roots4` and `roots16` are not equal to each other. They differ in these ways:

- `roots4` is a GF(4) array, while `roots16` is a GF(16) array. MATLAB keeps track of the underlying field of a Galois array.
- The array elements in `roots4` and `roots16` differ because they use representations with respect to different primitive polynomials. For example, `2` (which represents a primitive element) is an element of the vector `roots4` because the default primitive polynomial for GF(4) is the same polynomial that `gf4poly` represents. On the other hand, `2` is not an element of `roots16` because the primitive element of GF(16) is not a root of the polynomial that `gf16poly` represents.

Minimal Polynomials

The minimal polynomial of an element of $GF(2^m)$ is the smallest-degree nonzero binary-coefficient polynomial having that element as a root in $GF(2^m)$. To find the minimal polynomial of an element or a column vector of elements, use the `minpol` function.

The code below finds that the minimal polynomial of `gf(6,4)` is $D^2 + D + 1$ and then checks that `gf(6,4)` is indeed among the roots of that polynomial in the field GF(16).

```
m = 4;
e = gf(6,4);
em = minpol(e) % Find minimal polynomial of e. em is in GF(2).

em = GF(2) array.

Array elements =

     0     0     1     1     1

emr = roots(gf([0 0 1 1 1],m)) % Roots of D^2+D+1 in GF(2^m)

emr = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)
```

```
Array elements =

     6
     7
```

To find out which elements of a Galois field share the same minimal polynomial, use the `cosets` function.

Manipulating Galois Variables

This section describes techniques for manipulating Galois variables or for transferring information between Galois arrays and ordinary MATLAB arrays.

Note These techniques are particularly relevant if you write M-file functions that process Galois arrays. For an example of this type of usage, enter `edit gf/conv` in the Command Window and examine the first several lines of code in the editor window.

Determining Whether a Variable Is a Galois Array

To find out whether a variable is a Galois array rather than an ordinary MATLAB array, use the `isa` function. An illustration is below.

```
mlvar = eye(3);
gfvar = gf(mlvar,3);
no = isa(mlvar,'gf'); % False because mlvar is not a Galois array
yes = isa(gfvar,'gf'); % True because gfvar is a Galois array
```

Extracting Information From a Galois Array

To extract the array elements, field order, or primitive polynomial from a variable that is a Galois array, append a suffix to the name of the variable. The table below lists the exact suffixes, which are independent of the name of the variable.

Information	Suffix	Output Value
Array elements	`.x`	MATLAB array of type `uint16` that contains the data values from the Galois array
Field order	`.m`	Integer of type `double` that indicates that the Galois array is in GF(`2^m`)
Primitive polynomial	`.prim_poly`	Integer of type `uint32` that represents the primitive polynomial. The representation is similar to the description in "How Integers Correspond to Galois Field Elements" on page 2-97.

Note If the output value is an integer data type and you want to convert it to `double` for later manipulation, use the `double` function.

The code below illustrates the use of these suffixes. The definition of `empr` uses a vector of binary coefficients of a polynomial to create a Galois array in an extension field. Another part of the example retrieves the primitive polynomial for the field and converts it to a binary vector representation having the appropriate number of bits.

```
% Check that e solves its own minimal polynomial.
e = gf(5,4); % An element of GF(16)
emp = minpol(e); % The minimal polynomial, emp, is in GF(2).
empr = roots(gf(emp.x,e.m)) % Find roots of emp in GF(16).

% Check that the primitive element gf(2,m) is
% really a root of the primitive polynomial for the field.
primpoly_int = double(e.prim_poly);
mval = e.m;
primpoly_vect = gf(de2bi(primpoly_int,mval+1,'left-msb'),mval);
containstwo = roots(primpoly_vect); % Output vector includes 2.
```

Speed and Nondefault Primitive Polynomials

The section "Specifying the Primitive Polynomial" on page 2-99 described how you can represent elements of a Galois field with respect to a primitive polynomial of your choice. This section describes how you can increase the speed of computations involving a Galois array that uses a primitive polynomial other than the default primitive polynomial. The technique is recommended if you perform many such computations.

The mechanism for increasing the speed is a data file, `userGftable.mat`, that some computational functions use to avoid performing certain computations repeatedly. To take advantage of this mechanism for your combination of field order (`m`) and primitive polynomial (`prim_poly`):

1. Navigate in MATLAB to a directory to which you have write permission. You can use either the `cd` function or the Current Directory feature to navigate.
2. Define `m` and `prim_poly` as workspace variables. For example:

   ```
   m = 3; prim_poly = 13; % Examples of valid values
   ```

3. Invoke the `gftable` function:

   ```
   gftable(m,prim_poly); % If you previously defined m and prim_poly
   ```

The function revises or creates `userGftable.mat` in your current working directory to include data relating to your combination of field order and primitive polynomial. After you initially invest the time to invoke `gftable`, subsequent computations using those values of `m` and `prim_poly` should be faster.

Note If you change your current working directory after invoking `gftable`, then you must place `userGftable.mat` on your MATLAB path to ensure that MATLAB can see it. Do this by using the `addpath` command to prefix the directory containing `userGftable.mat` to your MATLAB path. If you have multiple copies of `userGftable.mat` on your path, then use `which('userGftable.mat','-all')` to find out where they are and which one MATLAB is using.

To see how much `gftable` improves the speed of your computations, you can surround your computations with the `tic` and `toc` functions. See the `gftable` reference page for an example.

Selected Bibliography for Galois Fields

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, Mass., Addison-Wesley, 1983, p. 105.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, Mass., Addison-Wesley, 1993.

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

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

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

3

Function Reference

Functions - By Category

The Communications Toolbox contains the following categories of functions:

- Signal Sources
- Signal Analysis Functions
- Source Coding
- Error-Control Coding
- Lower-Level Functions for Error-Control Coding
- Modulation and Demodulation
- Special Filters
- Lower-Level Functions for Special Filters
- Channel Functions
- Galois Field Computations
- Computations in Galois Fields of Odd Characteristic
- Utilities

Signal Sources

`randerr`	Generate bit error patterns
`randint`	Generate matrix of uniformly distributed random integers
`randsrc`	Generate random matrix using prescribed alphabet
`wgn`	Generate white Gaussian noise

Signal Analysis Functions

`biterr`	Compute number of bit errors and bit error rate
`eyediagram`	Generate an eye diagram
`scatterplot`	Generate a scatter plot
`symerr`	Compute number of symbol errors and symbol error rate

Source Coding

`arithdeco`	Decode binary code using arithmetic decoding
`arithenco`	Encode a sequence of symbols using arithmetic coding
`compand`	Source code mu-law or A-law compressor or expander
`dpcmdeco`	Decode using differential pulse code modulation
`dpcmenco`	Encode using differential pulse code modulation
`dpcmopt`	Optimize differential pulse code modulation parameters
`lloyds`	Optimize quantization parameters using the Lloyd algorithm
`quantiz`	Produce a quantization index and a quantized output value

Error-Control Coding

`bchpoly`	Produce parameters or generator polynomial for binary BCH code
`convenc`	Convolutionally encode binary data
`cyclgen`	Produce parity-check and generator matrices for cyclic code

`cyclpoly`	Produce generator polynomials for a cyclic code
`decode`	Block decoder
`encode`	Block encoder
`gen2par`	Convert between parity-check and generator matrices
`gfweight`	Calculate the minimum distance of a linear block code
`hammgen`	Produce parity-check and generator matrices for Hamming code
`rsdec`	Reed-Solomon decoder
`rsdecof`	Decode an ASCII file that was encoded using Reed-Solomon code
`rsenc`	Reed-Solomon encoder
`rsencof`	Encode an ASCII file using Reed-Solomon code
`rsgenpoly`	Generator polynomial of Reed-Solomon code
`syndtable`	Produce syndrome decoding table
`vitdec`	Convolutionally decode binary data using the Viterbi algorithm

Lower-Level Functions for Error-Control Coding

`bchdeco`	BCH decoder
`bchenco`	BCH encoder

Modulation and Demodulation

`ademod`	Analog passband demodulator
`ademodce`	Analog baseband demodulator
`amod`	Analog passband modulator
`amodce`	Analog baseband modulator
`apkconst`	Plot a combined circular ASK-PSK signal constellation
`ddemod`	Digital passband demodulator
`ddemodce`	Digital baseband demodulator
`demodmap`	Demap a digital message from a demodulated signal
`dmod`	Digital passband modulator
`dmodce`	Digital baseband modulator
`modmap`	Map a digital signal to an analog signal
`qaskdeco`	Demap a message from a QASK square signal constellation
`qaskenco`	Map a message to a QASK square signal constellation

Special Filters

`hank2sys`	Convert a Hankel matrix to a linear system model
`hilbiir`	Design a Hilbert transform IIR filter
`rcosflt`	Filter the input signal using a raised cosine filter
`rcosine`	Design a raised cosine filter

Lower-Level Functions for Special Filters

`rcosfir`	Design a raised cosine FIR filter
`rcosiir`	Design a raised cosine IIR filter

Channel Functions

`awgn`	Add white Gaussian noise to a signal

Galois Field Computations

`+ -`	Addition and subtraction of Galois arrays
`* / \`	Matrix multiplication and division of Galois arrays
`.* ./ .\`	Elementwise multiplication and division of Galois arrays
`^`	Matrix exponentiation of Galois array
`.^`	Elementwise exponentiation of Galois array
`' .'`	Transpose of Galois array
`==, ~=`	Relational operators for Galois arrays
`all`	True if all elements of a Galois vector are nonzero
`any`	True if any element of a Galois vector is nonzero
`conv`	Convolution of Galois vectors
`convmtx`	Convolution matrix of Galois field vector
`cosets`	Produce cyclotomic cosets for a Galois field
`deconv`	Deconvolution and polynomial division
`det`	Determinant of square Galois matrix
`dftmtx`	Discrete Fourier transform matrix in a Galois field
`diag`	Diagonal Galois matrices and diagonals of a Galois matrix
`fft`	Discrete Fourier transform
`filter`	One-dimensional digital filter over a Galois field

`gf`	Create a Galois field array
`gftable`	Generate a file to accelerate Galois field computations
`ifft`	Inverse discrete Fourier transform
`inv`	Inverse of Galois matrix
`isempty`	True for empty Galois arrays
`isprimitive`	True for a primitive polynomial for a Galois field
`length`	Length of Galois vector
`log`	Logarithm in a Galois field
`lu`	Lower-Upper triangular factorization of Galois array
`minpol`	Find the minimal polynomial of an element of a Galois field
`mldivide`	Matrix left division \ of Galois arrays
`polyval`	Evaluate polynomial in Galois field
`primpoly`	Find primitive polynomials for a Galois field
`rank`	Rank of a Galois array
`reshape`	Reshape Galois array
`roots`	Find polynomial roots across a Galois field
`size`	Size of Galois array
`tril`	Extract lower triangular part of Galois array
`triu`	Extract upper triangular part of Galois array

Computations in Galois Fields of Odd Characteristic

gfadd	Add polynomials over a Galois field
gfconv	Multiply polynomials over a Galois field
gfcosets	Produce cyclotomic cosets for a Galois field
gfdeconv	Divide polynomials over a Galois field
gfdiv	Divide elements of a Galois field
gffilter	Filter data using polynomials over a prime Galois field
gflineq	Find a particular solution of Ax = b over a prime Galois field
gfminpol	Find the minimal polynomial of an element of a Galois field
gfmul	Multiply elements of a Galois field
gfpretty	Display a polynomial in traditional format
gfprimck	Check whether a polynomial over a Galois field is primitive
gfprimdf	Provide default primitive polynomials for a Galois field
gfprimfd	Find primitive polynomials for a Galois field
gfrank	Compute the rank of a matrix over a Galois field
gfrepcov	Convert one binary polynomial representation to another
gfroots	Find the roots of a polynomial over a prime Galois field
gfsub	Subtract polynomials over a Galois field
gftrunc	Minimize the length of a polynomial representation
gftuple	Simplify or convert the format of elements of a Galois field

Utilities

bi2de	Convert binary vectors to decimal numbers
de2bi	Convert decimal numbers to binary vectors
erf	Error function
erfc	Complementary error function

`istrellis`	Check if the input is a valid trellis structure
`marcumq`	Generalized Marcum Q function
`mask2shift`	Convert mask vector to shift for a shift register configuration
`oct2dec`	Convert octal numbers to decimal numbers
`poly2trellis`	Convert convolutional code polynomials to trellis description
`shift2mask`	Convert shift to mask vector for a shift register configuration
`vec2mat`	Convert a vector into a matrix

Functions - Alphabetical List

Purpose

Analog passband demodulator

Syntax

```
z = ademod(y,Fc,Fs,'amdsb-tc',offset,num,den);
z = ademod(y,Fc,Fs,'amdsb-tc/costas',offset,num,den);
z = ademod(y,Fc,Fs,'amdsb-sc',num,den);
z = ademod(y,Fc,Fs,'amdsb-sc/costas',num,den);
z = ademod(y,Fc,Fs,'amssb',num,den);
z = ademod(y,Fc,Fs,'qam',num,den);
z = ademod(y,Fc,Fs,'fm',num,den,vcoconst);
z = ademod(y,Fc,Fs,'pm',num,den,vcoconst);
z = ademod(y,Fc,[Fs initphase],...);
```

Optional Inputs

Input	Default Value
`offset`	Appropriate value so that each output signal has zero mean
`num`, `den`	`[num,den] = butter(5,Fc*2/Fs);`
`vcoconst`	1

Description

The function `ademod` performs analog passband demodulation. The corresponding modulation function is `amod`. The table below lists the demodulation schemes that `ademod` supports.

Demodulation Scheme	Fourth Input Argument
Amplitude demodulation	'**amdsb-tc**' or '**amdsb-tc/costas**'
Amplitude demodulation, double sideband suppressed carrier	'**amdsb-sc**' or '**amdsb-sc/costas**'
Amplitude demodulation, single sideband suppressed carrier	'**amssb**'
Quadrature amplitude demodulation	'**qam**'
Frequency demodulation	'**fm**'
Phase demodulation	'**pm**'

For All Syntaxes

The generic syntax `z = ademod(y,Fc,Fs,...)` demodulates the received signal that `y` represents. `Fc` is the carrier frequency in hertz, and `Fs` is the sampling rate in hertz. The initial phase of the carrier signal is zero.

`y` and `z` are real matrices whose sizes depend on the demodulation method:

- **(QAM method)** If `y` is a length-n vector, then `z` is an n-by-2 matrix. Otherwise, if `y` is n-by-m, then `z` is n-by-2m and each column of `y` is processed separately. The odd-numbered columns in `z` represent in-phase components and the even-numbered columns represent quadrature components.
- **(Other methods)** `y` and `z` have the same dimensions. If `y` is a two-dimensional matrix, then each column of `y` is processed separately.

The generic syntax `z = ademod(y,Fc,[Fs initphase],...)` is the same, except that the third input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

`ademod` uses a lowpass filter with sample time `1/Fs` while demodulating, in order to filter out the carrier signal. To specify the lowpass filter, include `num` and `den` in the list of input arguments. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. If `num` is empty, zero, or absent, then the default filter is a Butterworth filter whose parameters come from the command below. `butter` is in the Signal Processing Toolbox.

```
[num,den] = butter(5,Fc*2/Fs);
```

For Specific Syntaxes

`z = ademod(y,Fc,Fs,'amdsb-tc',offset,num,den)` implements double-sideband amplitude demodulation. `offset` is a vector whose kth entry is subtracted from the kth signal after the demodulation. If `offset` is empty, then by default `z` is adjusted so that each column has mean zero (or so that `z` has mean zero in case `z` is a vector).

`z = ademod(y,Fc,Fs,'amdsb-tc/costas',offset,num,den)` is the same as the syntax above, except that the algorithm includes a Costas phase-locked loop.

`z = ademod(y,Fc,Fs,'amdsb-sc',num,den)` implements double-sideband suppressed-carrier amplitude demodulation.

`z = ademod(y,Fc,Fs,'amdsb-sc/costas',num,den)` is the same as the syntax above, except that the algorithm includes a Costas phase-locked loop.

`z = ademod(y,Fc,Fs,'amssb',num,den)` implements single-sideband suppressed-carrier amplitude demodulation.

`z = ademod(y,Fc,Fs,'qam',num,den)` implements quadrature amplitude demodulation.

`z = ademod(y,Fc,Fs,'fm',num,den,vcoconst)` implements frequency demodulation. The spectrum of the demodulated signal is between `min(y) + Fc` and `max(y) + Fc`. The demodulation process uses a phase-locked loop composed of a multiplier (as a phase detector), a lowpass filter, and a voltage-controlled oscillator (VCO). If `Fs` is a two-element vector, then its second element is the initial phase of the VCO, in radians. The optional argument `vcoconst` is a scalar that represents the VCO constant in Hz/V.

`z = ademod(y,Fc,Fs,'pm',num,den,vcoconst)` implements phase demodulation. The demodulation process uses a phase-locked loop (which acts as an FM demodulator) cascaded with an integrator. The phase-locked loop consists of a multiplier (as a phase detector), a lowpass filter, and a voltage-controlled oscillator (VCO). If `Fs` is a two-element vector, then its second element is the initial phase of the VCO, in radians. The optional argument `vcoconst` is a scalar that represents the input signal's sensitivity.

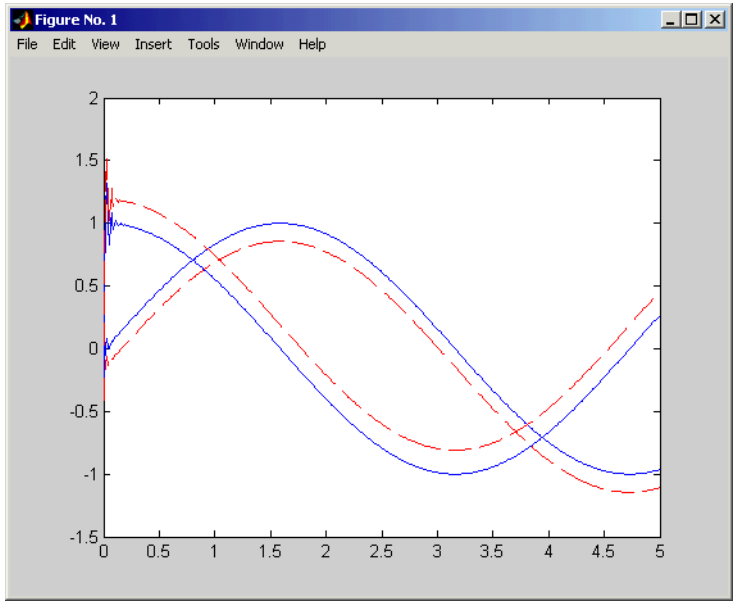
Examples

This example illustrates the use of the `offset` argument. Because the first `ademod` command uses the same `offset` value of .3 that the `amod` command used, `z1` is similar to the original message signal. Because the second `ademod` command omits `offset`, `z2` has mean close to zero (not exactly zero because of roundoff error).

```
Fc = 25; % Carrier signal frequency
Fs = 100; % Sampling rate of signal
t = [0:1/Fs:5]'; % Times to sample the signals
x = [cos(t), sin(t)]; % Cosine signal and sine signal
y = amod(x,Fc,Fs,'amdsb-tc',.3); % Modulate
% and shift the values up by .3.
```

```
z1 = ademod(y,Fc,Fs,'amdsb-tc',.3); % Demodulate.
z2 = ademod(y,Fc,Fs,'amdsb-tc'); % Demodulate.
plot(t,z1,'b',t,z2,'r--') % Plot recovered signal.
```

The plot shows `z1` as a solid line and `z2` as a dashed line.



Other examples using `ademod` are the Hilbert Filter Example on the reference page for `amod`, and in “Example: Varying the Filter’s Cutoff Frequency” on page 2-66.

See Also

`amod`, `dmod`, `ddemod`, `amodce`, `ademodce`

Purpose

Analog baseband demodulator

Syntax

```
z = ademodce(y,Fs,'amdsb-tc',offset,num,den);
z = ademodce(y,Fs,'amdsb-tc/costas',offset,num,den);
z = ademodce(y,Fs,'amdsb-sc',num,den);
z = ademodce(y,Fs,'amdsb-sc/costas',num,den);
z = ademodce(y,Fs,'amssb',num,den);
z = ademodce(y,Fs,'qam',num,den);
z = ademodce(y,Fs,'fm',num,den,vcoconst);
z = ademodce(y,Fs,'pm',num,den,vcoconst);
z = ademodce(y,[Fs initphase],...);
```

Optional Inputs

Input	Default Value, or Default Behavior If Input Is Omitted
`offset`	Appropriate value so that each output signal has zero mean
`num`, `den`	Omitting these arguments prevents `ademodce` from using a filter.
`vcoconst`	1

Description

The function `ademodce` performs analog baseband demodulation. The corresponding modulation function is `amodce`. The table below lists the demodulation schemes that `ademodce` supports.

Demodulation Scheme	Third Input Argument
Amplitude demodulation	'**amdsb-tc**' or '**amdsb-tc/costas**'
Amplitude demodulation, double sideband suppressed carrier	'**amdsb-sc**' or '**amdsb-sc/costas**'
Amplitude demodulation, single sideband suppressed carrier	'**amssb**'
Quadrature amplitude demodulation	'**qam**'
Frequency demodulation	'**fm**'
Phase demodulation	'**pm**'

For All Syntaxes

The generic syntax `z = ademodce(y,Fs,...)` demodulates the received signal that `y` represents. `Fs` is the sampling rate in hertz. The initial phase of the carrier signal is zero. `y` is a complex matrix and `z` is a real matrix. Their sizes depend on the demodulation method:

- **(QAM method)** If `y` is a vector of length n, then `z` is an n-by-2 matrix. Otherwise, if `y` is n-by-m, then `z` is n-by-2m and each column of `y` is processed separately. The odd-numbered columns in `z` represent in-phase components and the even-numbered columns represent quadrature components.
- **(Other methods)** `y` and `z` have the same dimensions. If `y` is a two-dimensional matrix, then each column of `y` is processed separately.

The generic syntax `z = ademodce(y,[Fs initphase],...)` is the same, except that the second input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate as described in the paragraph above. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

To use a lowpass filter in the demodulation, include `num` and `den` in the list of input arguments. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. If `num` is empty, zero, or absent, then `ademodce` does not use a filter.

For Specific Syntaxes

`z = ademodce(y,Fs,'amdsb-tc',offset,num,den)` implements double-sideband amplitude demodulation. `offset` is a vector whose kth entry is subtracted from the kth column of demodulated data. If `offset` is empty, then by default `z` is adjusted so that each column has mean zero (or so that `z` has mean zero in case `z` is a vector).

`z = ademodce(y,Fs,'amdsb-tc/costas',offset,num,den)` is the same as the syntax above, except that the algorithm includes a Costas phase-locked loop.

`z = ademodce(y,Fs,'amdsb-sc',num,den)` implements double-sideband suppressed-carrier amplitude demodulation.

`z = ademodce(y,Fs,'amdsb-sc/costas',num,den)` is the same as the syntax above, except that the algorithm includes a Costas phase-locked loop.

`z = ademodce(y,Fs,'amssb',num,den)` implements single-sideband suppressed-carrier amplitude demodulation.

`z = ademodce(y,Fs,'qam',num,den)` implements quadrature amplitude demodulation.

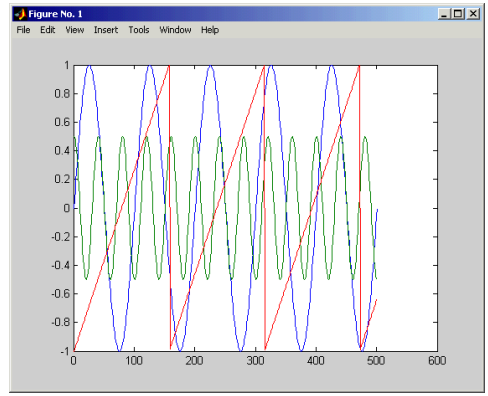
`z = ademodce(y,Fs,'fm',num,den,vcoconst)` implements frequency demodulation. The optional argument `vcoconst` is a scalar that represents the VCO constant in the demodulation.

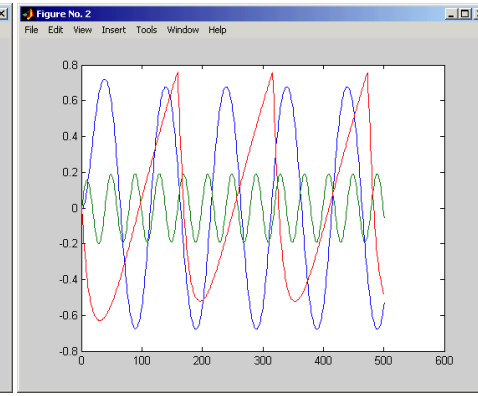
`z = ademodce(y,Fs,'pm',num,den,vcoconst)` implements phase demodulation. The optional argument `vcoconst` specifies the VCO constant in the demodulation.

Examples

The example below processes sine, cosine, and sawtooth signals simultaneously. All three signals have the same sampling rate and the same number of samples. The code also plots the original and demodulated signals.

```
Fs = 100; % Sampling rate of signal
t = [0:1/Fs:5]'; % Times to sample the signals
% Combine three signals into a three-column matrix.
% Each signal occupies one column.
x = [sin(2*pi*t), .5*cos(5*pi*t), sawtooth(4*t)];
y = amodce(x,Fs,'fm'); % Modulate.
z = ademodce(y,Fs,'fm'); % Demodulate.
plot(x); figure; plot(z); % Original and demodulated signals
```





Other examples using `ademodce` are in the sections "Simple Analog Modulation Example" on page 2-64 and "Example: Time Lag From Filtering" on page 2-67.

See Also

`amodce`, `dmodce`, `ddemodce`, `amod`, `ademod`

Purpose

Analog passband modulator

Syntax

```
y = amod(x,Fc,Fs,'amdsb-tc',offset);
y = amod(x,Fc,Fs,'amdsb-sc');
y = amod(x,Fc,Fs,'amssb/opt');
y = amod(x,Fc,Fs,'amssb/opt',num,den);
y = amod(x,Fc,Fs,'amssb/opt',hilbertflag);
y = amod(x,Fc,Fs,'qam');
y = amod(x,Fc,Fs,'fm',deviation);
y = amod(x,Fc,Fs,'pm',deviation);
y = amod(x,Fc,[Fs initphase],...);
[y,t] = amod(...);
```

Optional Inputs

Input	Default Value, or Default Behavior If Input Is Omitted
`offset`	`-min(min(x))`
`opt`	Omitting this argument causes `amod` to produce the lower sideband instead of the upper sideband.
`deviation`	1

Description

The function `amod` performs analog passband modulation. The corresponding demodulation function is `ademod`. The table below lists the modulation schemes that `amod` supports.

Modulation Scheme	Fourth Input Argument
Amplitude modulation, double sideband with transmission carrier	`'amdsb-tc'`
Amplitude modulation, double sideband suppressed carrier	`'amdsb-sc'`
Amplitude modulation, single sideband suppressed carrier	`'amssb'` or `'amssb/up'`
Quadrature amplitude modulation	`'qam'`

Modulation Scheme (Continued)	Fourth Input Argument (Continued)
Frequency modulation	`'fm'`
Phase modulation	`'pm'`

For All Syntaxes

The generic syntax `y = amod(x,Fc,Fs,...)` modulates the message signal that `x` represents. `Fc` is the carrier frequency in hertz, and `Fs` is the sampling rate in hertz. (Thus 1/`Fs` represents the time interval between two consecutive samples in `x`.) The initial phase of the carrier signal is zero. By the Nyquist theorem, the sampling rate must be at least twice as large as the modulation carrier frequency. `x` and `y` are real matrices whose sizes depend on the demodulation method:

- **(QAM method)** `x` must have an even number of columns. The odd-numbered columns in `x` represent in-phase components and the even-numbered columns represent quadrature components. If `x` is n-by-2m, then `y` is n-by-m and each *pair* of columns of `x` is processed separately.
- **(Other methods)** `x` and `y` have the same dimensions. If `x` is a two-dimensional matrix, then each column of `x` is processed separately.

The generic syntax `y = amod(x,Fc,[Fs initphase],...)` is the same, except that the third input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate as described in the paragraph above. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

For Specific Syntaxes

`y = amod(x,Fc,Fs,'amdsb-tc',offset)` implements double-sideband amplitude modulation. `offset` is the value added to `x` prior to the modulation. If you omit `offset`, then its default value is `-min(min(x))`. This default value produces 100% modulation.

`y = amod(x,Fc,Fs,'amdsb-sc')` implements double-sideband suppressed-carrier amplitude modulation.

`y = amod(x,Fc,Fs,'amssb/opt')` implements single-sideband suppressed-carrier amplitude modulation. By default, it produces the lower

sideband; if *opt* is **up**, then the function produces the upper sideband. This syntax does a Hilbert transform in the frequency domain.

`y = amod(x,Fc,Fs,'amssb/opt',num,den)` is the same as the syntax above, except that it specifies a time-domain Hilbert filter. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. You can use the function `hilbiir` to design the Hilbert filter.

`y = amod(x,Fc,Fs,'amssb/opt',hilbertflag)` is the same as the syntax above, except that it uses a default time-domain Hilbert filter. The filter's transfer function is defined by `[num,den] = hilbiir(1/Fs)`, where `num` and `den` are as in the paragraph above. The input argument `hilbertflag` can have any value.

`y = amod(x,Fc,Fs,'qam')` implements quadrature amplitude modulation. `x` is a two-column matrix whose first column represents the in-phase signal and whose second column represents the quadrature signal. `y` is a column vector.

`y = amod(x,Fc,Fs,'fm',deviation)` implements frequency modulation. The spectrum of the modulated signal is between `min(x) + Fc` and `max(x) + Fc`. The optional argument `deviation` is a scalar that represents the frequency deviation constant of the modulation. The command `y = amod(x,Fc,Fs,'fm',deviation)` is equivalent to the command `y = amod(x*deviation,Fc,Fs,'fm')`.

`y = amod(x,Fc,Fs,'pm',deviation)` implements phase modulation. The optional argument `deviation` is a scalar that represents the phase deviation constant of the modulation. The command `y = amod(x,Fc,Fs,'pm',deviation)` is equivalent to the command `y = amod(x*deviation,Fc,Fs,'pm')`.

`[y,t] = amod(...)` returns the computation time in `t`.

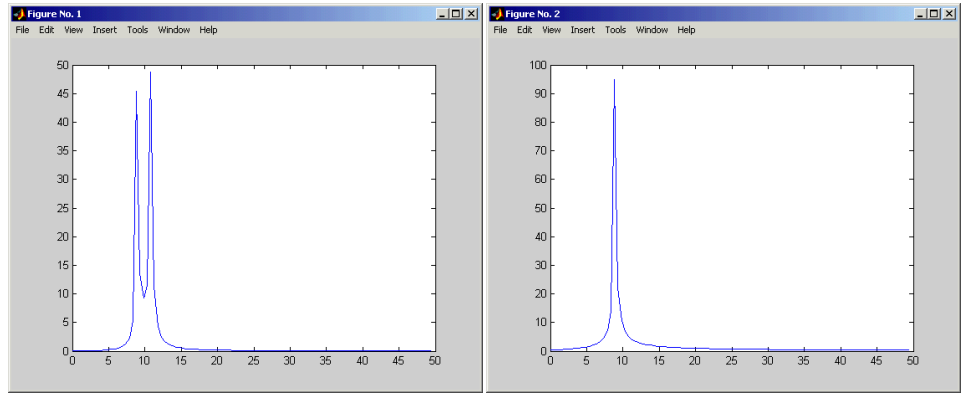
Examples

Double- and Single-Sideband Comparison Example

The first example compares the spectra of signals after modulation using the double-sideband and single-sideband techniques. The message signal is a frequency-one sine wave and the carrier signal is a 10 Hz sine wave. The script below uses the '**amdsb-sc**' and '**amssb**' arguments in the `amod` function to

produce modulated signals `ydouble` and `ysingle`, respectively. It then plots the spectra of both modulated signals.

```
% Sample the signal 100 times per second, for 2 seconds.
Fs = 100;
t = [0:2*Fs+1]'/Fs;
Fc = 10; % Carrier frequency
x = sin(2*pi*t); % Sinusoidal signal
% Modulate x using single- and double-sideband AM.
ydouble = amod(x,Fc,Fs,'amdsb-sc');
ysingle = amod(x,Fc,Fs,'amssb');
% Plot spectra of both modulated signals.
zdouble = fft(ydouble);
zdouble = abs(zdouble(1:length(zdouble)/2+1));
frqdouble = [0:length(zdouble)-1]*Fs/length(zdouble)/2;
plot(frqdouble,zdouble); % The plot on the left-hand side below
figure;
zsingle = fft(ysingle);
zsingle = abs(zsingle(1:length(zsingle)/2+1));
frqsingle = [0:length(zsingle)-1]*Fs/length(zsingle)/2;
plot(frqsingle,zsingle); % The plot on the right-hand side below
```



Notice that the spectrum in the left plot has two peaks; these 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.

In the right plot, there is one peak because the SSB AM technique requires `amod` to transmit only one sideband.

Hilbert Filter Example

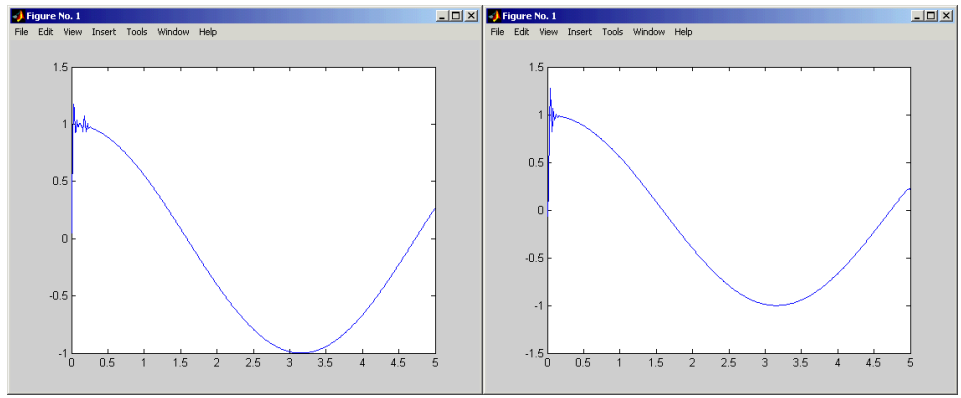
The next example uses a Hilbert filter in the time domain.

```
Fc = 25; % Carrier signal frequency
Fs = 100; % Sampling rate of signal
[numh,denh] = hilbiir(1/Fs,15/Fs,15); % Design Hilbert filter.
t = [0:1/Fs:5]'; % Times to sample the signal
x = cos(t); % Signal is a cosine wave.
y = amod(x,Fc,[Fs pi/4],'amssb',numh,denh); % Modulate,
% using a Hilbert filter in the time domain.
z = ademod(y,Fc,[Fs pi/4],'amssb'); % Demodulate.
plot(t,z) % Plot recovered signal.
```

The resulting plot is on the left below. If you replace the sixth line above with

```
y = amod(x,Fc,[Fs pi/4],'amssb'); % Modulate,
```

then modulation uses a Hilbert transform in the frequency domain. The result is the plot on the right below. The two plots differ slightly in their initial errors.



See Also

`ademod`, `dmod`, `ddemod`, `amodce`, `ademodce`

amodce

Purpose

Analog baseband modulator

Syntax

```
y = amodce(x,Fs,'amdsb-tc',offset);
y = amodce(x,Fs,'amdsb-sc');
y = amodce(x,Fs,'amssb');
y = amodce(x,Fs,'amssb/time',num,den);
y = amodce(x,Fs,'amssb/time');
y = amodce(x,Fs,'qam');
y = amodce(x,Fs,'fm',deviation);
y = amodce(x,Fs,'pm',deviation);
y = amodce(x,[Fs initphase],...);
```

Optional Inputs

Input	Default Value, or Default Behavior If Input Is Omitted
`offset`	`-min(min(x))`
`deviation`	1

Description

The function `amodce` performs analog baseband modulation. The corresponding demodulation function is `ademodce`. The table below lists the modulation schemes that `amodce` supports.

Modulation Scheme	Third Input Argument
Amplitude modulation, double sideband	`'amdsb-tc'`
Amplitude modulation, double sideband suppressed carrier	`'amdsb-sc'`
Amplitude modulation, single sideband suppressed carrier	`'amssb'` or `'amssb/time'`
Quadrature amplitude modulation	`'qam'`
Frequency modulation	`'fm'`
Phase modulation	`'pm'`

For All Syntaxes

The generic syntax `y = amodce(x,Fs,...)` modulates the message signal that `x` represents, and returns the modulated signal's complex envelope. The input and output signals share the same sampling rate `Fs`, measured in hertz. (Thus `1/Fs` represents the time interval between two consecutive samples in `x`.) The initial phase of the carrier signal is zero. `x` is a real matrix and `y` is a complex matrix. Their sizes depend on the modulation method:

- **(QAM method)** `x` must have an even number of columns. The odd-numbered columns in `x` represent in-phase components and the even-numbered columns represent quadrature components. If `x` is n-by-2m, then `y` is n-by-m and each *pair* of columns of `x` is processed separately.
- **(Other methods)** `x` and `y` have the same dimensions. If `x` is a two-dimensional matrix, then each column of `x` is processed separately.

The generic syntax `y = amodce(x,[Fs initphase],...)` is the same, except that the second input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate as described in the paragraph above. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

For Specific Syntaxes

`y = amodce(x,Fs,'amdsb-tc',offset)` implements double-sideband amplitude modulation. `offset` is the value added to `x` prior to the modulation. If you omit `offset`, then its default value is `-min(min(x))`. This default value produces 100% modulation.

`y = amodce(x,Fs,'amdsb-sc')` implements double-sideband suppressed-carrier amplitude modulation.

`y = amodce(x,Fs,'amssb')` implements single-sideband suppressed-carrier amplitude modulation. By default, it produces the lower sideband. It does a Hilbert transform in the frequency domain.

`y = amodce(x,Fs,'amssb/time',num,den)` is the same as the syntax above, except that it specifies a time-domain Hilbert filter. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. You can use the function `hilbiir` to design the Hilbert filter.

`y = amodce(x,Fs,'amssb/time')` is the same as the syntax above, except that it uses a default time-domain Hilbert filter. The filter's transfer function is defined by `[num,den] = hilbiir(1/Fs)`, where `num` and `den` are as in the paragraph above.

`y = amodce(x,Fs,'qam')` implements quadrature amplitude modulation. `x` is a two-column matrix whose first column represents the in-phase signal and whose second column represents the quadrature signal. `y` is a column vector.

`y = amodce(x,Fs,'fm',deviation)` implements frequency modulation. The bandwidth of the modulated signal is `max(x)-min(x)`. The optional argument `deviation` is a scalar that represents the frequency deviation constant of the modulation.

`y = amodce(x,Fs,'pm',deviation)` implements phase modulation. The optional argument `deviation` is a scalar that represents the phase deviation constant of the modulation.

Examples

This example is similar to the one under the heading "Hilbert Filter Example" on the `amod` reference page, except that it uses baseband simulation. The plots in the passband (`amod`) example show far more obvious errors in the recovered signal.

```
Fs = 100; % Sampling rate of signal
[numh,denh] = hilbiir(1/Fs,15/Fs,15); % Design Hilbert filter.
t = [0:1/Fs:5]'; % Times to sample the signal
x = cos(t); % Signal is a cosine wave.
y = amodce(x,[Fs pi/4],'amssb/time',numh,denh); % Modulate,
% using a Hilbert filter in the time domain.
z = ademodce(y,[Fs pi/4],'amssb'); % Demodulate.
% Find order of magnitude of average difference between x and z.
d = ceil(log10(sum(abs(x-z))/length(x)))
```

The output shows that the average difference between the original and recovered signals is smaller than 10^{-16}.

```
d =

   -16
```

Other examples using `amodce` are in the sections "Representing Analog Signals" on page 2-62 and "Simple Analog Modulation Example" on page 2-64.

See Also

`ademodce`, `dmodce`, `ddemodce`, `amod`, `ademod`

apkconst

Purpose

Plot a combined circular ASK-PSK signal constellation

Syntax

```
apkconst(numsig);
apkconst(numsig,amp);
apkconst(numsig,amp,phs);
apkconst(numsig,amp,'n');
apkconst(numsig,amp,phs,plotspec);
y = apkconst(...);
```

Description

APK refers to a hybrid of amplitude- and phase-keying modulation. See the reference listed below for more details.

`apkconst(numsig)` plots a circular signal constellation. `numsig` is a vector of positive integers. The plot contains `length(numsig)` circles. The kth circle has radius k and contains `numsig`(k) evenly spaced constellation points. One point on each circle has zero phase.

`apkconst(numsig,amp)` is the same as the previous syntax, except that `amp`(k) is the radius of the kth circle. `amp` is a vector of positive real numbers. The lengths of `amp` and `numsig` must be the same.

`apkconst(numsig,amp,phs)` is the same as the previous syntax, except that it is not necessarily true that one point on each circle has zero phase. However, one point on the kth circle has phase `phs`(k). The lengths of `phs`, `amp`, and `numsig` must all be the same.

`apkconst(numsig,amp,phs,'n')` is the same as the previous syntax, except that the plot includes a number next to each constellation point. The number indicates how symbols would be mapped to constellation points if you were using `numsig`, `amp`, and `phs` in modulation and demodulation functions such as `dmodce/ddemodce` or `modmap/demodmap`.

`apkconst(numsig,amp,phs,plotspec)` is the same as `apkconst(numsig,amp,phs)`, except that `plotspec` influences the appearance of the constellation points via the `plot` function. `plotspec` is a two-character

string made up of one character from each odd-numbered column in the table below.

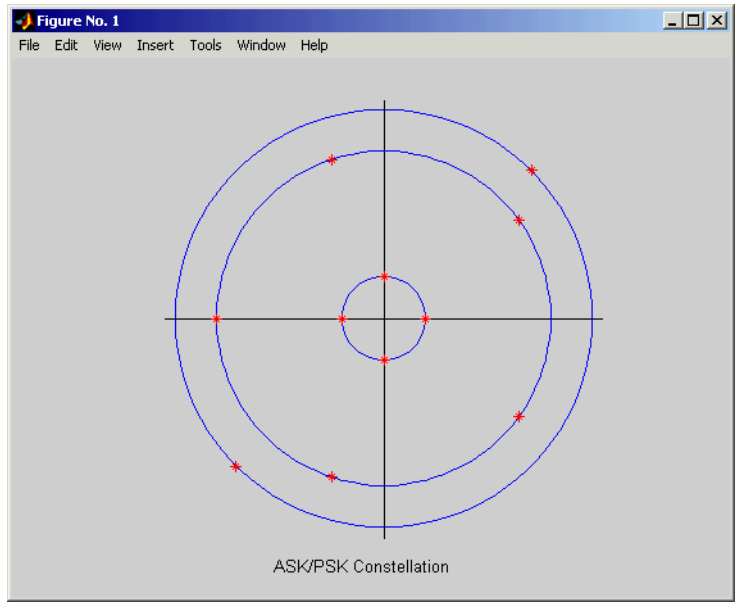
Color Character	Meaning	Marker-Type Character	Meaning
`y`	Yellow	`.`	Point
`m`	Magenta	`o`	Circle
`c`	Cyan	`x`	Cross
`r`	Red	`+`	Plus sign
`g`	Green	`*`	Asterisk
`b`	Blue	`s`	Square
`w`	White	`d`	Diamond
`k`	Black	`v`	Triangle (down)
		`^`	Triangle (up)
		`<`	Triangle (left)
		`>`	Triangle (right)
		`p`	Five-pointed star
		`h`	Six-pointed star

`y = apkconst(...)` does *not* produce a plot, but instead returns a complex vector `y` that represents the coordinates of the points in the constellation. The real part of `y` gives the in-phase component of each point and the imaginary part of `y` gives the quadrature component of each point.

Examples

The command below produces a plot having three circles. One circle has radius 1 and four points, one of which has zero phase. Another circle has radius 4 and five points, one of which has phase π. The outermost circle has radius 5 and two points, one of which has phase π/4. The plot follows.

```
apkconst([4 5 2],[1 4 5],[0 pi pi/4])
```



The command below produces a vector containing the coordinates in the complex plane of the points in the figure above.

```
y = apkconst([4 5 2],[1 4 5],[0 pi pi/4])

y =

  Columns 1 through 4

   1.0000            0.0000 + 1.0000i  -1.0000 + 0.0000i  -0.0000 - 1.0000i

  Columns 5 through 8

  -4.0000 + 0.0000i  -1.2361 - 3.8042i   3.2361 - 2.3511i   3.2361 + 2.3511i

  Columns 9 through 11

  -1.2361 + 3.8042i   3.5355 + 3.5355i  -3.5355 - 3.5355i
```

See Also

dmod, modmap, ddemod, demodmap

References

Thomas, C. Melvil, Michaeil Y. Weidner, and S. H. Durrani, "Digital Amplitude-Phase Keying with M-ary Alphabets," *IEEE Transactions on Communications,* Vol. Com-22, No. 2, Feb., 1974, pp. 168-180.

arithdeco

Purpose

Decode binary code using arithmetic decoding

Syntax

```
dseq = arithdeco(code,counts,len)
```

Description

`dseq = arithdeco(code,counts,len)` decodes the binary arithmetic code in the vector `code` to recover the corresponding sequence of `len` symbols. The vector `counts` represents the source's statistics by listing the number of times each symbol of the source's alphabet occurs in a test data set. This function assumes that the data in `code` was produced by the `arithenco` function.

Examples

This example is similar to the example on the `arithenco` reference page, except that it uses `arithdeco` to recover the original sequence.

```
counts = [99 1]; % A one occurs 99% of the time.
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01],19069); % Random sequence
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq)); % Decode.
isequal(seq,dseq) % Check that dseq matches the original seq.
```

The output is

```
ans =

     1
```

Algorithm

This function uses the algorithm described in [1].

See Also

`arithenco`

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

Purpose

Encode a sequence of symbols using arithmetic coding

Syntax

```
code = arithenco(seq,counts)
```

Description

`code = arithenco(seq,counts)` generates the binary arithmetic code corresponding to the sequence of symbols specified in the vector `seq`. The vector `counts` represents the source's statistics by listing the number of times each symbol of the source's alphabet occurs in a test data set.

Examples

This example illustrates the compression that arithmetic coding can accomplish in some situations. A source has a two-symbol alphabet and produces a test data set in which 99% of the symbols are 1s. Encoding 1000 symbols from this source produces a code vector having many fewer than 1000 elements. The actual number of elements in `code` varies, depending on the particular random sequence contained in `seq`.

```
counts = [99 1]; % A one occurs 99% of the time.
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01],19069); % Random sequence
code = arithenco(seq,counts);
s = size(code) % length of code is only 8.3% of length of seq.
```

The output is

```
s =

     1    83
```

Algorithm

This function uses the algorithm described in [1].

See Also

`arithdeco`

References

[1] Sayood, Khalid, *Introduction to Data Compression,* San Francisco, Morgan Kaufmann, 2000.

Purpose

Add white Gaussian noise to a signal

Syntax

```
y = awgn(x,snr);
y = awgn(x,snr,sigpower);
y = awgn(x,snr,'measured');
y = awgn(x,snr,sigpower,state);
y = awgn(x,snr,'measured',state);
y = awgn(...,powertype);
```

Description

`y = awgn(x,snr)` adds white Gaussian noise to the vector signal `x`. The scalar `snr` specifies the signal-to-noise ratio in decibels. If `x` is complex, then `awgn` adds complex noise. This syntax assumes that the power of `x` is 0 dBW.

`y = awgn(x,snr,sigpower)` is the same as the syntax above, except that `sigpower` is the power of `x` in dBW.

`y = awgn(x,snr,'measured')` is the same as `y = awgn(x,snr)`, except that `awgn` measures the power of `x` before adding noise.

`y = awgn(x,snr,sigpower,state)` is the same as `y = awgn(x,snr,sigpower)`, except that `awgn` first resets the state of the normal random number generator `randn` to the integer `state`.

`y = awgn(x,snr,'measured',state)` is the same as `y = awgn(x,snr,'measured')`, except that `awgn` first resets the state of normal random number generator `randn` to the integer `state`.

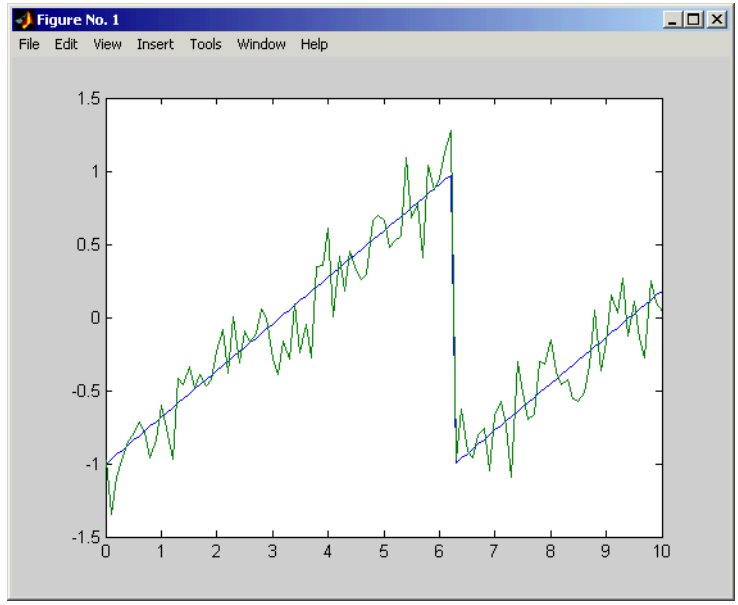
`y = awgn(...,`*`powertype`*`)` is the same as the previous syntaxes, except that the string *`powertype`* specifies the units of `snr` and `sigpower`. Choices for *`powertype`* are '**`db`**' and '**`linear`**'. If *`powertype`* is '**`db`**', then `snr` is measured in dB and `sigpower` is measured in dBW. If *`powertype`* is '**`linear`**', then `snr` is measured as a ratio and `sigpower` is measured in watts.

Examples

The commands below add white Gaussian noise to a sawtooth signal. It then plots the original and noisy signals.

```
t = 0:.1:10;
x = sawtooth(t); % Create sawtooth signal.
y = awgn(x,10,'measured'); % Add white Gaussian noise.
```

```
plot(t,x,t,y) % Plot both signals.
```



See Also

wgn, randn

bchdeco

Purpose

BCH decoder

Syntax

```
msg = bchdeco(code,k,t);
msg = bchdeco(code,k,t,prim_poly);
[msg,err] = bchdeco(...);
[msg,err,ccode] = bchdeco(...);
```

Description

`msg = bchdeco(code,k,t)` decodes `code` using the BCH method. `k` is the message length. The codeword length n must have the form 2^m-1 for some integer m greater than or equal to 3. `code` is a binary matrix with n columns, each row of which represents one codeword. `msg` is a binary matrix with `k` columns, each row of which represents one message. `t` is the error-correction capability. BCH decoding requires a primitive polynomial for GF(2^m); this syntax uses the default primitive polynomial, `gfprimdf(m)`.

`msg = bchdeco(code,k,t,prim_poly)` is the same as the first syntax, except that `prim_poly` is a row vector that gives the coefficients, in order of ascending powers, of the primitive polynomial for GF(2^m) that will be used during processing.

`[msg,err] = bchdeco(...)` returns a column vector `err` that gives information about error correction. A nonnegative integer in `err(r)` indicates the number of errors corrected in the `r`th codeword; a negative integer indicates that there are more errors in the `r`th codeword than can be corrected.

`[msg,err,ccode] = bchdeco(...)` returns the corrected code in `ccode`.

Examples

The script below encodes a (random) message, simulates the addition of noise to the code, and then decodes the message.

```
m = 4; n = 2^m-1; % Codeword length
params = bchpoly(n);
% Arbitrarily focus on 3rd row of params.
k = params(3,2); % Codeword length
t = params(3,3); % Error-correction capability
msg = randint(100,k);
code = bchenco(msg,n,k); % Encode the message.
% Corrupt up to t bits in each codeword.
noisycode = rem(code + randerr(100,n,1:t),2);
```

```
% Decode the noisy code.
[newmsg,err,ccode] = bchdeco(noisycode,k,t);
if ccode==code
   disp('All errors were corrected.')
end
if newmsg==msg
   disp('The message was recovered perfectly.')
end
```

In this case, all errors are corrected and the message is recovered perfectly. However, if the ninth line is changed to

```
noisycode = rem(code + randerr(100,n,1:(t+1)),2);
```

then some codewords will contain more than `t` errors. This is too many errors, and some will go uncorrected.

See Also

`bchenco`, `bchpoly`

bchenco

Purpose

BCH encoder

Syntax

```
code = bchenco(msg,n,k);
code = bchenco(msg,n,k,genpoly);
```

Description

`code = bchenco(msg,n,k)` encodes `msg` using the BCH technique and the generator polynomial `genpoly = bchpoly(n,k)`. `n` is the codeword length and `k` is the message length. `msg` is a binary matrix with `k` columns. Each row of `msg` represents a message. `code` is a binary matrix with `n` columns. Each row of `code` represents a codeword.

`code = bchenco(msg,n,k,genpoly)` is the same as the first syntax, except that `genpoly` is a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples

See the example on the reference page for the function `bchdeco`.

See Also

`bchdeco`, `encode`, `decode`, `bchpoly`, `cyclgen`

Purpose

Produce parameters or generator polynomial for binary BCH code

Syntax

```
bchpoly
params = bchpoly
params = bchpoly(n);
genpoly = bchpoly(n,k);
genpoly = bchpoly(prim_poly,k);
[genpoly,factors] = bchpoly(...,k);
[genpoly,factors,cst] = bchpoly(...,k);
[genpoly,factors,cst,h] = bchpoly(...,k);
[genpoly,factors,cst,h,t] = bchpoly(...,k);
```

Description

`bchpoly` produces a figure window containing a table that lists valid codeword and message lengths of binary BCH codes, as well as the corresponding error-correction capabilities. The codeword lengths listed are 7, 15, 31, 63, 127, 255, and 511. The codeword lengths, message length, and error-correction capabilities are denoted by N, K, and T, respectively.

`params = bchpoly` produces a three-column matrix containing the same information that is in the table mentioned in the syntax above. The first column of `params` gives the codeword length, the second column gives the message length, and the third column gives the error-correction capability.

`params = bchpoly(n)` produces a matrix `params` containing valid codeword and message lengths of binary BCH codes in its first and second columns, respectively. If `n` < 1024, then `params` has a third column that lists the corresponding error-correction capabilities. The codeword lengths listed in the first column of `params` are all equal to `max(7,2^ceil(log2(n+1))-1)`. This expression gives the smallest number of the form 2^m-1 that is at least as big as `n`, where m is an integer greater than or equal to 3.

`genpoly = bchpoly(n,k)` produces a generator polynomial for a binary BCH code having codeword length `n` and message length `k`. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the generator polynomial. `n` must have the form 2^m-1 for some integer m greater than or equal to 3. `k` must be a valid message length, as reported in the second column of the output of the command `genpoly = bchpoly(n)`. The primitive

polynomial used for the GF(2^m) calculations is the default primitive polynomial, `gfprimdf(m)`.

`genpoly = bchpoly(prim_poly,k)` produces a generator polynomial for a binary BCH code having codeword length n and message length `k`. `prim_poly` represents a degree-m primitive polynomial for the field GF(2^m). Both `prim_poly` and `genpoly` are row vectors that represent polynomials by giving the coefficients in order of ascending powers. Given the degree m of the primitive polynomial, the message length n is 2^m-1. `k` must be a valid message length, as reported in the second column of the output of the command `genpoly = bchpoly(n)`.

The remaining syntaxes, of the form

```
[genpoly,...] = bchpoly(...,k)
```

return some or all of the output variables listed in the table below.

Additional Output Variables for bchpoly(...,k)

Output Variable	Significance	Format
`factors`	Irreducible factors of the generator polynomial	Binary matrix, each row of which gives the coefficients of a factor polynomial in order of ascending powers
`cst`	Cyclotomic cosets of the field GF(2^m)	Same as `gfcosets(m)`
`h`	Parity-check matrix of the code	`(n-k)`-by-`n` binary matrix
`t`	Error-correction capability of the code	Positive integer

Examples

The script below uses `bchpoly` to find out what message lengths are valid for a BCH code with codeword length 2^4-1. It then chooses one of the possible message lengths and uses `bchpoly` to find the generator polynomial and parity-check matrix for such a code.

```
m = 4;
n = 2^m-1; % Codeword length is 15.
% Want to find out possible valid message lengths.
params = bchpoly(n);
disp(['Possible message lengths are ',num2str(params(:,2)')])
disp(' ')

ii = 1; % Arbitrarily choose first row.
k = params(ii,2); % Message lengths are in 2nd column.
% Get generator polynomial and other facts.
[genpoly,factors,cst,parmat,errorcorr] = bchpoly(n,k);
disp(['For k = ',num2str(k),' the generator polynomial is'])
gfpretty(genpoly)
disp('and the parity-check matrix is')
parmat
```

The full output is below.

```
Possible message lengths are 11   7   5

For k = 11 the generator polynomial is

                                        4
                                  1 + X + X
and the parity-check matrix is

parmat =

  Columns 1 through 12

     1     0     0     0     1     0     0     1     1     0     1     0
     0     1     0     0     1     1     0     1     0     1     1     1
     0     0     1     0     0     1     1     0     1     0     1     1
     0     0     0     1     0     0     1     1     0     1     0     1

  Columns 13 through 15

     1     1     1
     1     0     0
     1     1     0
     1     1     1
```

See Also

`cyclpoly`, `encode`, `decode`

References

Peterson, W. Wesley, and E. J. Weldon, Jr., *Error-correcting Codes*, 2nd ed., Cambridge, Mass., MIT Press, 1972.

Purpose

Convert binary vectors to decimal numbers

Syntax

```
d = bi2de(b);
d = bi2de(b,flg)
d = bi2de(b,p);
d = bi2de(b,p,flg);
```

Description

`d = bi2de(b)` converts a binary row vector `b` to a nonnegative decimal integer. If `b` is a matrix, then each row is interpreted separately as a binary number. In this case, the output `d` is a column vector, each element of which is the decimal representation of the corresponding row of `b`.

Note By default, `bi2de` interprets the first column of `b` as the *lowest*-order digit.

`d = bi2de(b,flg)` is the same as the syntax above, except that *`flg`* is a string that determines whether the first column of `b` contains the lowest-order or highest-order digits. Possible values for *`flg`* are '**`right-msb`**' and '**`left-msb`**'. The value '**`right-msb`**' produces the default behavior.

`d = bi2de(b,p)` converts a base-`p` row vector `b` to a nonnegative decimal integer , where `p` is an integer greater than or equal to 2. The first column of `b` is the *lowest* base-`p` digit. If `b` is a matrix, then the output `d` is a nonnegative decimal vector, each row of which is the decimal form of the corresponding row of `b`.

`d = bi2de(b,p,flg)` is the same as the syntax above, except that *`flg`* is a string that determines whether the first column of `b` contains the lowest-order or highest-order digits. Possible values for *`flg`* are '**`right-msb`**' and '**`left-msb`**'. The value '**`right-msb`**' produces the default behavior.

Examples

The code below generates a matrix that contains binary representations of five random numbers between 0 and 15. It then converts all five numbers to decimal integers.

```
b = randint(5,4);   % Generate a 5-by-4 random binary matrix.
de = bi2de(b);
```

```
disp('    Dec          Binary')
disp('   -----   ------------------')
disp([de, b])
```

Sample output is below. Your results might vary because the numbers are random.

```
    Dec          Binary
   -----   ------------------
    13     1     0     1     1
     7     1     1     1     0
    15     1     1     1     1
     4     0     0     1     0
     9     1     0     0     1
```

The command below converts a base-five number into its decimal counterpart, using the leftmost base-five digit (4 in this case) as the most significant digit. The example reflects the fact that $4(5^3) + 2(5^2) + 5^0 = 551$.

```
d = bi2de([4 2 0 1],5,'left-msb')

d =

   551
```

See Also

de2bi

Purpose

Compute number of bit errors and bit error rate

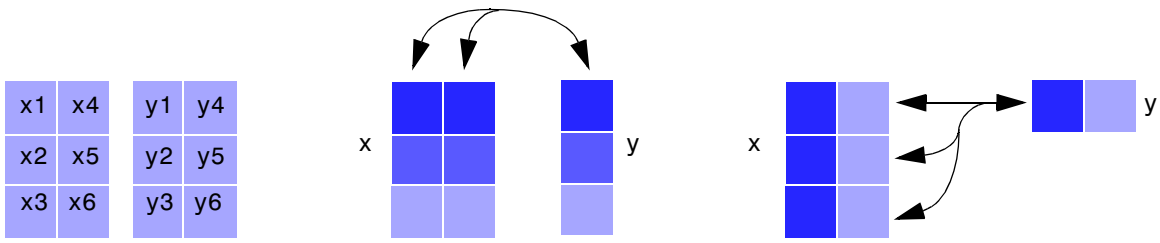
Syntax

```
[number,ratio] = biterr(x,y);
[number,ratio] = biterr(x,y,k);
[number,ratio] = biterr(...,flg);
[number,ratio,individual] = biterr(...)
```

Description

For All Syntaxes

The `biterr` function compares unsigned binary representations of elements in `x` with those in `y`. The schematics below illustrate how the shapes of `x` and `y` determine which elements `biterr` compares.



(a) Compares x1 with y1, x2 with y2, and so on.

(b) Compares column vector y with each column of matrix x

(c) Compares row vector y with each row of matrix x

Each element of `x` and `y` must be a nonnegative decimal integer; `biterr` converts each element into its natural unsigned binary representation. `number` is a scalar or vector that indicates the number of bits that differ. `ratio` is `number` divided by the *total number of bits*. The total number of bits, the size of `number`, and the elements that `biterr` compares are determined by the dimensions of `x` and `y` and by the optional parameters.

For Specific Syntaxes

`[number,ratio] = biterr(x,y)` compares the elements in `x` and `y`. If the largest among all elements of `x` and `y` has exactly `k` bits in its simplest binary representation, then the total number of bits is `k` times the number of entries in the *smaller* input. The sizes of `x` and `y` determine which elements are compared:

- If `x` and `y` are matrices of the same dimensions, then `biterr` compares `x` and `y` element-by-element. `number` is a scalar. See schematic (a) in the figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then `biterr` compares the vector element-by-element with *each row (resp., column)* of the matrix. The length of the vector must equal the number of columns (resp., rows) in the matrix. `number` is a column (resp., row) vector whose mth entry indicates the number of bits that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

`[number,ratio] = biterr(x,y,k)` is the same as the first syntax, except that it considers each entry in `x` and `y` to have `k` bits. The total number of bits is `k` times the number of entries of the smaller of `x` and `y`. An error occurs if the binary representation of an element of `x` or `y` would require more than `k` digits.

`[number,ratio] = biterr(x,y,k,flg)` is similar to the previous syntaxes, except that *`flg`* can override the defaults that govern which elements `biterr` compares and how `biterr` computes the outputs. The possible values of *`flg`* are '**`row-wise`**', '**`column-wise`**', and '**`overall`**'. The table below describes the differences that result from various combinations of inputs. As always, `ratio` is `number` divided by the total number of bits. If you do not provide `k` as an input argument, then the function defines it internally as the number of bits in the simplest binary representation of the largest among all elements of `x` and `y`.

Comparing a Two-Dimensional Matrix x with Another Input y

Shape of y	flg	Type of Comparison	number	Total Number of Bits
Two-dimensional matrix	'**overall**' (default)	Element-by-element	Total number of bit errors	k times number of entries of y
	'**row-wise**'	mth row of x vs. mth row of y	Column vector whose entries count bit errors in each row	k times number of entries of y
	'**column-wise**'	mth column of x vs. mth column of y	Row vector whose entries count bit errors in each column	k times number of entries of y
Row vector	'**overall**'	y vs. each row of x	Total number of bit errors	k times number of entries of x
	'**row-wise**' (default)	y vs. each row of x	Column vector whose entries count bit errors in each row of x	k times size of y
Column vector	'**overall**'	y vs. each column of x	Total number of bit errors	k times number of entries of x
	'**column-wise**' (default)	y vs. each column of x	Row vector whose entries count bit errors in each column of x	k times size of y

`[number,ratio,individual] = biterr(...)` returns a matrix `individual` whose dimensions are those of the larger of `x` and `y`. Each entry of `individual` corresponds to a comparison between a pair of elements of `x` and `y`, and specifies the number of bits by which the elements in the pair differ.

Examples

Example 1

The commands below compare the column vector [0; 0; 0] to each column of a random binary matrix. The output is the number, proportion, and locations of 1s in the matrix. In this case, `individual` is the same as the random matrix.

```
format rat;
[number,ratio,individual] = biterr([0;0;0],randint(3,5))

number =

      2           0           0           3           1


ratio =

     2/3          0           0           1          1/3


individual =

      1           0           0           1           0
      1           0           0           1           0
      0           0           0           1           1
```

Example 2

The commands below illustrate the use of *`flg`* to override the default row-by-row comparison. Notice that `number` and `ratio` are scalars, while `individual` has the same dimensions as the larger of the first two arguments of `biterr`.

```
format rat;
[number,ratio,individual] = biterr([1 2; 3 4],[1 3],3,'overall')

number =

      5
```

```
ratio =

     5/12


individual =

      0           1
      1           3
```

Example 3

The script below adds errors to 10% of the elements in a matrix. Each entry in the matrix is a two-bit number in decimal form. The script computes the bit error rate using `biterr` and the symbol error rate using `symerr`.

```
x = randint(100,100,4); % Original signal
% Create errors to add to ten percent of the elements of x.
% Errors can be either 1, 2, or 3 (not zero).
errorplace = (rand(100,100) > .9); % Where to put errors
errorvalue = randint(100,100,[1,3]); % Value of the errors
errors = errorplace.*errorvalue;
y = rem(x+errors,4); % Signal with errors added, mod 4
format short
[num_bit,ratio_bit] = biterr(x,y,2)
[num_sym,ratio_sym] = symerr(x,y)
```

Sample output is below. Notice that `ratio_sym` is close to the target value of `0.10`. Your results might vary because the example uses random numbers.

```
num_bit =

        1304


ratio_bit =

    0.0652
```

```
num_sym =

   981


ratio_sym =

    0.0981
```

See Also

symerr

Purpose

Source code mu-law or A-law compressor or expander

Syntax

```
out = compand(in,Mu,v);
out = compand(in,Mu,v,'mu/compressor');
out = compand(in,Mu,v,'mu/expander');
out = compand(in,A,v,'A/compressor');
out = compand(in,A,v,'A/expander');
```

Description

`out = compand(in,param,v)` implements a μ-law compressor for the input vector `in`. `Mu` specifies μ and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,Mu,v,'mu/compressor')` is the same as the syntax above.

`out = compand(in,Mu,v,'mu/expander')` implements a μ-law expander for the input vector `in`. `Mu` specifies μ and `v` is the input signal's maximum magnitude. `out` has the same dimensions and maximum magnitude as `in`.

`out = compand(in,A,v,'A/compressor')` implements an A-law compressor for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

`out = compand(in,A,v,'A/expander')` implements an A-law expander for the input vector `in`. The scalar `A` is the A-law parameter, and `v` is the input signal's maximum magnitude. `out` is a vector of the same length and maximum magnitude as `in`.

Note The prevailing parameters used in practice are μ = 255 and A = 87.6.

Examples

The examples below illustrate the fact that compressors and expanders perform inverse operations.

```
compressed = compand(1:5,87.6,5,'a/compressor')
```

```
compressed =

    3.5296    4.1629    4.5333    4.7961    5.0000

expanded = compand(compressed,87.6,5,'a/expander')

expanded =

    1.0000    2.0000    3.0000    4.0000    5.0000
```

Algorithm

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

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

where V is the maximum value of the signal x, μ is the μ-law parameter of the compander, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB).

The output of the A-law compressor is

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

where A is the A-law parameter of the compander and the other elements are as in the μ-law case.

See Also

`quantiz`, `dpcmenco`, `dpcmdeco`

References

Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

Purpose

Convolutionally encode binary data

Syntax

```
code = convenc(msg,trellis);
code = convenc(msg,trellis,init_state);
[code,final_state] = convenc(...);
```

Description

`code = convenc(msg,trellis)` encodes the binary vector `msg` using the convolutional encoder whose MATLAB trellis structure is `trellis`. For details about MATLAB trellis structures, see "Trellis Description of a Convolutional Encoder" on page 2-50. Each symbol in `msg` consists of `log2(trellis.numInputSymbols)` bits. The vector `msg` contains one or more symbols. The output vector `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits.

`code = convenc(msg,trellis,init_state)` is the same as the syntax above, except that `init_state` specifies the starting state of the encoder registers. The scalar `init_state` is an integer between 0 and `trellis.numStates-1`. If the encoder schematic has more than one input stream, then the shift register that receives the first input stream provides the least significant bits in `init_state`, while the shift register that receives the last input stream provides the most significant bits in `init_state`. To use the default value for `init_state`, specify `init_state` as `0` or `[]`.

`[code,final_state] = convenc(...)` encodes the input message and also returns in `final_state` the encoder's state. `final_state` has the same format as `init_state`.

Examples

The command below encodes five two-bit symbols using a rate 2/3 convolutional code. A schematic of this encoder is on the reference page for the `poly2trellis` function.

```
code1 = convenc(randint(10,1,2,123),...
poly2trellis([5 4],[23 35 0;0 5 13]));
```

The commands below define the encoder's trellis structure explicitly and then use `convenc` to encode ten one-bit symbols. A schematic of this encoder is in "Trellis Description of a Convolutional Encoder" on page 2-50.

```
trel = struct('numInputSymbols',2,'numOutputSymbols',4,...
'numStates',4,'nextStates',[0 2;0 2;1 3;1 3],...
```

```
'outputs',[0 3;1 2;3 0;2 1]);
code2 = convenc(randint(10,1),trel);
```

The commands below illustrate how to use the final state and initial state arguments when invoking `convenc` repeatedly. Notice that `[code3; code4]` is the same as the earlier example's output, `code1`.

```
trel = poly2trellis([5 4],[23 35 0;0 5 13]);
msg = randint(10,1,2,123);
% Encode part of msg, recording final state for later use.
[code3,fstate] = convenc(msg(1:6),trel);
% Encode the rest of msg, using state as an input argument.
code4 = convenc(msg(7:10),trel,fstate);
```

See Also

`vitdec`, `poly2trellis`, `istrellis`, `vitsimdemo`

References

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

Purpose

Convolution matrix of Galois field vector

Syntax

```
A = convmtx(c,n);
```

Description

A *convolution matrix* is a matrix, formed from a vector, whose inner product with another vector is the convolution of the two vectors.

`A = convmtx(c,n)` returns a convolution matrix for the Galois vector `c`. The output `A` is a Galois array that represents convolution with `c` in the sense that `conv(c,x)` equals

- `A*x`, if `c` is a column vector and `x` is any Galois column vector of length `n`. In this case, `A` has `n` columns and `m+n-1` rows.
- `x*A`, if `c` is a row vector and `x` is any Galois row vector of length `n`. In this case, `A` has `n` rows and `m+n-1` columns.

Examples

The code below illustrates the equivalence between using the `conv` function and multiplying by the output of `convmtx`.

```
m = 4;
c = gf([1; 9; 3],m); % Column vector
n = 6;
x = gf(randint(n,1,2^m),m);
ck1 = isequal(conv(c,x), convmtx(c,n)*x) % True
ck2 = isequal(conv(c',x'),x'*convmtx(c',n)) % True
```

The output is

```
ck1 =

     1


ck2 =

     1
```

See Also

`conv`

cosets

Purpose

Produce cyclotomic cosets for a Galois field

Syntax

```
cst = cosets(m);
```

Description

`cst = cosets(m)` produces cyclotomic cosets mod `2^m-1`. Each element of the cell array `cst` is a Galois array that represents one cyclotomic coset.

A cyclotomic coset is a set of elements that share the same minimal polynomial. Together, the cyclotomic cosets mod `2^m-1` form a partition of the group of nonzero elements of GF(`2^m`). For more details on cyclotomic cosets, see the works listed in "References" below.

Examples

The commands below find and display the cyclotomic cosets for GF(8). As an example of interpreting the results, `c{2}` indicates that A, A^2, and A^2 + A share the same minimal polynomial, where A is a primitive element for GF(8).

```
c = cosets(3);
c{1}'

ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     1

c{2}'

ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     2     4     6

c{3}'

ans = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     3     5     7
```

See Also

`minpol`

References

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, Mass., Addison-Wesley, 1983, p. 105.

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

cyclgen

Purpose

Produce parity-check and generator matrices for cyclic code

Syntax

```
h = cyclgen(n,pol);
h = cyclgen(n,pol,opt);
[h,g] = cyclgen(...);
[h,g,k] = cyclgen(...);
```

Description

For all syntaxes, the codeword length is `n` and the message length is `k`. A polynomial can generate a cyclic code with codeword length `n` and message length `k` if and only if the polynomial is a degree-`(n-k)` divisor of `x^n-1`. (Over the binary field GF(2), `x^n-1` is the same as `x^n+1`.) This implies that `k` equals `n` minus the degree of the generator polynomial.

`h = cyclgen(n,pol)` produces an `(n-k)`-by-`n` parity-check matrix for a systematic binary cyclic code having codeword length `n`. The row vector `pol` gives the binary coefficients, in order of ascending powers, of the degree-`(n-k)` generator polynomial.

`h = cyclgen(n,pol,opt)` is the same as the syntax above, except that the argument *`opt`* determines whether the matrix should be associated with a systematic or nonsystematic code. The values for *`opt`* are '**`system`**' and '**`nonsys`**'.

`[h,g] = cyclgen(...)` is the same as `h = cyclgen(...)` except that it also produces the `k`-by-`n` generator matrix `g` that corresponds to the parity-check matrix `h`.

`[h,g,k] = cyclgen(...)` is the same as `[h,g] = cyclgen(...)` except that it also returns the message length `k`.

Examples

The code below produces parity-check and generator matrices for a binary cyclic code with codeword length 7 and message length 4.

```
pol = cyclpoly(7,4);
[parmat,genmat,k] = cyclgen(7,pol)
```

The output is

```
parmat =

     1     0     0     1     1     1     0
     0     1     0     0     1     1     1
     0     0     1     1     1     0     1


genmat =

     1     0     1     1     0     0     0
     1     1     1     0     1     0     0
     1     1     0     0     0     1     0
     0     1     1     0     0     0     1


k =

     4
```

In the output below, notice that the parity-check matrix is different from `parmat` above, because it corresponds to a nonsystematic cyclic code. In particular, `parmatn` does not have a 3-by-3 identity matrix in its leftmost three columns, as `parmat` does.

```
parmatn = cyclgen(7,cyclpoly(7,4),'nonsys')

parmatn =

     1     1     1     0     1     0     0
     0     1     1     1     0     1     0
     0     0     1     1     1     0     1
```

See Also

`encode`, `decode`, `bchpoly`, `cyclpoly`

cyclpoly

Purpose

Produce generator polynomials for a cyclic code

Syntax

```
pol = cyclpoly(n,k);
pol = cyclpoly(n,k,opt);
```

Description

For all syntaxes, a polynomial is represented as a row containing the coefficients in order of ascending powers.

`pol = cyclpoly(n,k)` returns the row vector representing one nontrivial generator polynomial for a cyclic code having codeword length `n` and message length `k`.

`pol = cyclpoly(n,k,opt)` searches for one or more nontrivial generator polynomials for cyclic codes having codeword length `n` and message length `k`. The output `pol` depends on the argument `opt` as shown in the table below.

opt	Significance of pol	Format of pol
`'min'`	One generator polynomial having the smallest possible weight	The row vector representing the polynomial
`'max'`	One generator polynomial having the greatest possible weight	The row vector representing the polynomial
`'all'`	All generator polynomials	A matrix, each row of which represents one such polynomial
a positive integer, L	All generator polynomials having weight L	A matrix, each row of which represents one such polynomial

The weight of a binary polynomial is the number of nonzero terms it has. If no generator polynomial satisfies the given conditions, then the output `pol` is empty and an error message is displayed.

Examples

The first command below produces representations of three generator polynomials for a [15,4] cyclic code. The second command shows that $1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^{11}$ is one such polynomial having the largest number of nonzero terms.

```
c1 = cyclpoly(15,4,'all')
```

```
c2 = cyclpoly(15,4,'max')
```

The output is

```
c1 =

     1     1     0     0     0     1     1     0     0     0     1     1
     1     0     0     1     1     0     1     0     1     1     1     1
     1     1     1     1     0     1     0     1     1     0     0     1

c2 =

     1     1     1     1     0     1     0     1     1     0     0     1
```

This command shows that no generator polynomial for a [15,4] cyclic code has exactly three nonzero terms.

```
c3 = cyclpoly(15,4,3)

No generator polynomial satisfies the given constraints.

c3 =

     []
```

Algorithm

If *opt* is `'min'`, `'max'`, or omitted, then polynomials are constructed by converting decimal integers to base `p`. Based on the decimal ordering, `gfprimfd` returns the first polynomial it finds that satisfies the appropriate conditions. This algorithm is similar to the one used in `gfprimfd`.

See Also

`cyclgen`, `encode`

ddemod

Purpose

Digital passband demodulator

Syntax

```
z = ddemod(y,Fc,Fd,Fs,'ask/opt',M,num,den);
z = ddemod(y,Fc,Fd,Fs,'fsk/opt',M);
z = ddemod(y,Fc,Fd,Fs,'msk');
z = ddemod(y,Fc,Fd,Fs,'psk/opt',M,num,den);
z = ddemod(y,Fc,Fd,Fs,'qask/opt',M,num,den);
z = ddemod(y,Fc,Fd,Fs,'qask/arb/opt',inphase,quadr,num,den);
z = ddemod(y,Fc,Fd,Fs,'qask/cir/opt',numsig,amp,phs,num,den);
z = ddemod(y,Fc,Fd,[Fs initphase],...);
```

Optional Inputs

Input	Default Value, or Default Behavior If Input Is Omitted
opt	ddemod demaps after demodulating. If the method is ASK, then the algorithm does not use a Costas loop. If the method is FSK, then demodulation is coherent.
num, den	Omitting these arguments prevents ddemod from using a filter.
amp	[1:length(numsig)]
phs	numsig*0

Description

The function ddemod performs digital passband demodulation. The corresponding modulation function is dmod. The table below lists the demodulation schemes that ddemod supports.

Demodulation Scheme	Fifth Input Argument	Where */opt* Can Contain
M-ary amplitude shift keying	'**ask**/*opt*'	/**nomap**; /**costas**
M-ary frequency shift keying	'**fsk**/*opt*'	/**noncoherence**
Minimum shift keying	'**msk**'	
M-ary phase shift keying	'**psk**/*opt*'	/**nomap**
Quadrature amplitude shift keying	'**qask**/*opt*', '**qask**/**arb**/*opt*', or '**qask**/**cir**/*opt*'	/**nomap**

The second column of the table indicates in bold type the required portion of the fifth input argument for `ddemod`. The third column indicates optional flags that you can append to the fifth argument. The order of optional flags does not matter.

To Demodulate Without Demapping (ASK, PSK, QASK only)

Ordinarily, the `ddemod` function first demodulates the analog signal it receives and then demaps the demodulated signal in order to recover the digital message signal. The optional **`/nomap`** flag, appended to the fifth input argument, prevents `ddemod` from demapping. The output is then an analog signal `x` whose sampling rate is `Fs`. You can use the `demodmap` function to perform the demapping step. The **`/nomap`** option is not available for FSK or MSK demodulation.

To Demodulate a Digital Signal (General Information)

The generic syntax `z = ddemod(y,Fc,Fd,Fs,...)` demodulates the digital message signal `z` from a received analog signal `y`. After measuring the distance from the received signal to all possible digits in the coding scheme, `ddemod` returns the nearest digit.

`y` and `z` are real matrices whose sizes depend on the demodulation method:

- **(ASK, FSK, MSK methods)** If `y` is a vector of length `n*Fs/Fd`, then `z` is a column vector of length n. Otherwise, if `y` is (`n*Fs/Fd`)-by-m, then `z` is n-by-m and each column of `y` is processed separately.
- **(PSK, QASK methods)** If `y` is (`n*Fs/Fd`)-by-m, then `z` is n-by-2m. The odd-numbered columns in `z` represent in-phase components and the even-numbered columns represent quadrature components. Each column of `y` is processed separately.

The carrier frequency in hertz is `Fc`. The sampling rates in hertz of `y` and `z`, respectively, are `Fs` and `Fd`. (Thus 1/`Fs` represents the time interval between two consecutive samples in `y`, and similarly for `z`.) The ratio `Fs/Fd` must be a positive integer. The time interval between two decision points is 1/`Fd`.

The generic syntax `z = ddemod(y,Fc,Fd,[Fs initphase],...)` is the same, except that the fourth input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate as described in the paragraph above. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

ddemod can use a lowpass filter with sample time 1/Fs while demodulating, in order to filter out the carrier signal. To specify the lowpass filter, include num and den in the list of input arguments. num and den are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. If num is empty, zero, or absent, then the function does not use a filter.

To Demodulate a Digital Signal (Specific Syntax Information)

`z = ddemod(y,Fc,Fd,Fs,'ask',M)` implements M-ary amplitude shift keying demodulation. Each entry of z is in the range [0, M-1].

`z = ddemod(y,Fc,Fd,Fs,'ask/costas',M)` is the same as the syntax above, except that the algorithm includes a Costas loop.

`z = ddemod(y,Fc,Fd,Fs,'fsk',M,tone)` implements coherent M-ary frequency shift keying demodulation. The optional argument tone is the separation between successive frequencies in the modulated signal z. The default value of tone is Fd. Each entry of z is in the range [0, M-1].

`z = ddemod(y,Fc,Fd,Fs,'fsk/noncoherence',M,tone)` is the same as the syntax above, except that it uses noncoherent demodulation.

`z = ddemod(y,Fc,Fd,Fs,'msk')` implements minimum shift keying demodulation. Each entry of z is either 0 or 1. The separation between the two frequencies is Fd/2.

`z = ddemod(y,Fc,Fd,Fs,'psk',M)` implements M-ary correlation phase shift keying demodulation. Each entry of z is in the range [0, M-1].

`z = ddemod(y,Fc,Fd,Fs,'qask',M)` implements M-ary quadrature amplitude shift keying demodulation with a square signal constellation. The table below

shows the maximum among in-phase and quadrature coordinates of constellation points, for several small values of M.

M	Maximum of Coordinates of Constellation Points	M	Maximum of Coordinates of Constellation Points
2	1	32	5
4	1	64	7
8	3 (quadrature maximum is 1)	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)`.

`z = ddemod(y,Fc,Fd,Fs,'qask/arb',inphase,quadr)` implements quadrature amplitude shift keying demodulation, with a signal constellation that you define using the vectors `inphase` and `quadr`. The signal constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`z = ddemod(y,Fc,Fd,Fs,'qask/cir',numsig,amp,phs)` implements quadrature amplitude shift keying demodulation with a circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of the kth circle, `numsig`(k) is the number of constellation points on the kth circle, and `phs`(k) is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

Examples

This example mimics the one in "Simple Digital Modulation Example" on page 2-77 but uses passband simulation. It generates a random digital signal, modulates it using `dmod`, and adds noise. Then it demodulates the noisy signal and computes the symbol error rate. The `ddemod` function demodulates the analog signal `y` and then demaps to produce the digital signal `z`.

Important differences between this example and the original baseband example are the explicit reference to the carrier signal frequency `Fc` and the fact that `y` and `ynoisy` are real, not complex. For variety, this example uses ASK instead of PSK, as well as a different sampling rate `Fd`.

```
M = 16; % Use 16-ary modulation.
Fc = 10; % Carrier signal frequency is 10 Hz.
Fd = 1; % Sampling rates of original and modulated signals
Fs = 50; % are 1 and 50, respectively (samples per second).
x = randint(100,1,M); % Random digital message
% Use M-ary PSK modulation to produce y.
y = dmod(x,Fc,Fd,Fs,'ask',M);
% Add some Gaussian noise.
ynoisy = y + .01*randn(Fs/Fd*100,1);
% Demodulate y to recover the message.
z = ddemod(ynoisy,Fc,Fd,Fs,'ask',M);
s = symerr(x,z) % Check symbol error rate.

s =

     0
```

See Also

`dmod`, `amod`, `ademod`, `dmodce`, `ddemodce`, `demodmap`, `modmap`, `eyediagram`, `scatterplot`

Purpose

Digital baseband demodulator

Syntax

```
z = ddemodce(y,Fd,Fs,'ask/opt',M,num,den);
z = ddemodce(y,Fd,Fs,'fsk/opt',M);
z = ddemodce(y,Fd,Fs,'msk');
z = ddemodce(y,Fd,Fs,'psk/opt',M,num,den);
z = ddemodce(y,Fd,Fs,'qask/opt',M,num,den);
z = ddemodce(y,Fd,Fs,'qask/arb/opt',inphase,quadr,num,den);
z = ddemodce(y,Fd,Fs,'qask/cir/opt',numsig,amp,phs,num,den);
z = ddemodce(y,Fd,[Fs initphase],...);
```

Optional Inputs

Input	Default Value, or Default Behavior If Input Is Omitted
opt	ddemodce demaps after demodulating. If the method is ASK, then the algorithm does not use a Costas loop. If the method is FSK, then demodulation is coherent.
num, den	Omitting these arguments prevents ddemodce from using a filter.
amp	[1:length(numsig)]
phs	numsig*0

Description

The function ddemodce performs digital baseband demodulation. The corresponding modulation function is dmodce. The table below lists the demodulation schemes that ddemodce supports.

Demodulation Scheme	Fourth Input Argument	Where */opt* Can Contain
M-ary amplitude shift keying	'**ask**/*opt*'	**/nomap**; **/costas**
M-ary frequency shift keying	'**fsk**/*opt*'	**/noncoherence**
Minimum shift keying	'**msk**'	
M-ary phase shift keying	'**psk**/*opt*'	**/nomap**
Quadrature amplitude shift keying	'**qask**/*opt*', '**qask/arb**/*opt*', or '**qask/cir**/*opt*'	**/nomap**

The second column of the table indicates in bold type the required portion of the fourth input argument for `ddemodce`. The third column indicates optional flags that you can append to the fourth argument. The order of optional flags does not matter.

To Demodulate Without Demapping (ASK, PSK, QASK Only)

Ordinarily, the `ddemodce` function first demodulates the analog signal it receives and then demaps the demodulated signal in order to recover the digital message signal. The optional **`/nomap`** flag, appended to the fourth input argument, prevents `ddemodce` from demapping. The output is then an analog signal `z` whose sampling rate is `Fs`. The size of `z` depends on the size of `y` and the demodulation method:

- **(ASK method)** `z` has the same size as `y`.
- **(PSK and QASK methods)** If `y` is a vector of length n, then `z` is an n-by-2 matrix. Otherwise, if `y` is n-by-m, then `z` is n-by-2m and each column of `y` is processed separately. In either case, the odd-numbered columns in `z` represent in-phase components and the even-numbered columns represent quadrature components.

You can use the `demodmap` function to perform the demapping step. The **`/nomap`** option is not available for FSK or MSK demodulation.

To Demodulate a Digital Signal (General Information)

The generic syntax `z = ddemodce(y,Fd,Fs,...)` demodulates the digital message signal `z` from a received analog signal `y`. After measuring the distance from the received signal to all possible digits in the coding scheme, `ddemodce` returns the nearest digit.

`y` is a complex matrix and `z` is a real matrix. The sizes of `y` and `z` depend on the demodulation method:

- **(ASK, FSK, MSK methods)** If `y` is a vector of length `n*Fs/Fd`, then `z` is a column vector of length n. Otherwise, if `y` is (`n*Fs/Fd`)-by-m, then `z` is n-by-m and each column of `y` is processed separately.
- **(PSK, QASK methods)** If `y` is (`n*Fs/Fd`)-by-m, then `z` is n-by-2m. The odd-numbered columns in `z` represent in-phase components and the even-numbered columns represent quadrature components. Each column of `y` is processed separately.

The sampling rates in hertz of `y` and `z`, respectively, are `Fs` and `Fd`. (Thus `1/Fs` represents the time interval between two consecutive samples in `y`, and similarly for `z`.) The ratio `Fs/Fd` must be a positive integer. The time interval between two decision points is `1/Fd`.

The generic syntax `z = ddemodce(y,Fd,[Fs initphase],...)` is the same, except that the third input argument is a two-element vector instead of a scalar. The first entry, `Fs`, is the sampling rate as described in the paragraph above. The second entry, `initphase`, is the initial phase of the carrier signal, measured in radians.

To use a lowpass filter in conjunction with ASK, PSK, or QASK demodulation, include `num` and `den` in the list of input arguments. `num` and `den` are row vectors that give the coefficients, in *descending* order, of the numerator and denominator of the filter's transfer function. If `num` is empty, zero, or absent, then `ddemodce` does not use a filter.

To Demodulate a Digital Signal (Specific Syntax Information)

`z = ddemodce(y,Fd,Fs,'ask',M)` implements `M`-ary amplitude shift keying demodulation. Each entry of `z` is in the range [0, `M`-1].

`z = ddemodce(y,Fd,Fs,'ask/costas',M)` is the same as the syntax above, except that the algorithm includes a Costas loop.

`z = ddemodce(y,Fd,Fs,'fsk',M,tone)` implements coherent `M`-ary frequency shift keying demodulation. The optional argument `tone` is the separation between successive frequencies in the modulated signal `z`. The default value of `tone` is `Fd`. Each entry of `z` is in the range [0, `M`-1].

`z = ddemodce(y,Fd,Fs,'fsk/noncoherence',M,tone)` is the same as the syntax above, except that it uses noncoherent demodulation.

`z = ddemodce(y,Fd,Fs,'msk')` implements minimum shift keying demodulation. Each entry of `z` is either 0 or 1. The separation between the two frequencies is `Fd/2`.

`z = ddemodce(y,Fd,Fs,'psk',M)` implements `M`-ary correlation phase shift keying demodulation. Each entry of `z` is in the range [0, `M`-1].

`z = ddemodce(y,Fd,Fs,'qask',M)` implements `M`-ary quadrature amplitude shift keying demodulation with a square signal constellation. The table below shows the maximum among in-phase and quadrature coordinates of constellation points, for several small values of M.

M	Maximum of Coordinates of Constellation Points	M	Maximum of Coordinates of Constellation Points
2	1	32	5
4	1	64	7
8	3 (quadrature maximum is 1)	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)`.

`z = ddemodce(y,Fd,Fs,'qask/arb',inphase,quadr)` implements quadrature amplitude shift keying demodulation, with a signal constellation that you define using the vectors `inphase` and `quadr`. The signal constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`z = ddemodce(y,Fd,Fs,'qask/cir',numsig,amp,phs)` implements quadrature amplitude shift keying demodulation with a circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of the kth circle, `numsig`(k) is the number of constellation points on the kth circle, and `phs`(k) is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

See Also

`dmodce`, `amodce`, `ademodce`, `dmod`, `ddemod`, `demodmap`, `modmap`, `eyediagram`, `scatterplot`

de2bi

Purpose

Convert decimal numbers to binary vectors

Syntax

```
b = de2bi(d);
b = de2bi(d,n);
b = de2bi(d,n,p);
b = de2bi(d,[],p);
b = de2bi(d,...,flg)
```

Description

`b = de2bi(d)` converts a nonnegative decimal integer `d` to a binary row vector. If `d` is a vector, then the output `b` is a matrix, each row of which is the binary form of the corresponding element in `d`. If `d` is a matrix, then `de2bi` treats it like the vector `d(:)`.

Note By default, `de2bi` uses the first column of `b` as the *lowest*-order digit.

`b = de2bi(d,n)` is the same as `b = de2bi(d)`, except that its output has `n` columns, where `n` is a positive integer. An error occurs if the binary representations would require more than `n` digits. If necessary, the binary representation of `d` is padded with extra zeros.

`b = de2bi(d,n,p)` converts a nonnegative decimal integer `d` to a base-`p` row vector, where `p` is an integer greater than or equal to 2. The first column of `b` is the *lowest* base-`p` digit. `b` is padded with extra zeros if necessary, so that it has `n` columns, where `n` is a positive integer. An error occurs if the base-`p` representations would require more than `n` digits. If `d` is a nonnegative decimal vector, then the output `b` is a matrix, each row of which is the (possibly zero-padded) base-p form of the corresponding element in `d`. If `d` is a matrix, then `de2bi` treats it like the vector `d(:)`.

`b = de2bi(d,[],p)` specifies the base `p` but not the number of columns.

`b = de2bi(d,...,flg)` uses the string *`flg`* to determine whether the first column of `b` contains the lowest-order or highest-order digits. Values for *`flg`* are '**`right-msb`**' and '**`left-msb`**'. The value '**`right-msb`**' produces the default behavior.

Examples

The code below counts to ten in decimal and binary.

```
d = (1:10)';
b = de2bi(d);
disp('    Dec          Binary       ')
disp('   -----   -------------------')
disp([d, b])
```

The output is below.

```
     Dec          Binary
    -----   -------------------
      1     1     0     0     0
      2     0     1     0     0
      3     1     1     0     0
      4     0     0     1     0
      5     1     0     1     0
      6     0     1     1     0
      7     1     1     1     0
      8     0     0     0     1
      9     1     0     0     1
     10     0     1     0     1
```

The command below shows how `de2bi` pads its output with zeros.

```
bb = de2bi([3 9],5) % Zero-padding the output

bb =

     1     1     0     0     0
     1     0     0     1     0
```

The commands below show how to convert a decimal integer to base three without specifying the number of columns in the output matrix. They also show how to place the most significant digit on the left instead of on the right.

```
t = de2bi(12,[],3) % Convert 12 to base 3.

t =

     0     1     1
```

```
tleft = de2bi(12,[],3,'left-msb') % Significant digit on left

tleft =

     1     1     0
```

See Also

bi2de

Purpose

Block decoder

Syntax

```
msg = decode(code,n,k,'hamming/fmt',prim_poly);
msg = decode(code,n,k,'linear/fmt',genmat,trt);
msg = decode(code,n,k,'cyclic/fmt',genpoly,trt);
msg = decode(code,n,k,'bch/fmt',t,prim_poly);
msg = decode(code,n,k);
[msg,err] = decode(...);
[msg,err,ccode] = decode(...);
[msg,err,ccode,cerr] = decode(...);
```

Optional Inputs

Input	Default Value
fmt	**binary**
prim_poly	gfprimdf(m) where n = 2^m-1
genpoly	cyclpoly(n,k)
trt	Uses syndtable to create the syndrome decoding table associated with the method's parity-check matrix

Description

For All Syntaxes

The decode function aims to recover messages that were encoded using an error-correction coding technique. The technique and the defining parameters must match those that were used to encode the original signal.

The "For All Syntaxes" section on the reference page for the encode function explains the meanings of n and k, the possible values of *fmt*, and the possible formats for code and msg. You should be familiar with the conventions described there before reading the rest of this section. Using the decode function with an input argument code that was *not* created by the encode function might cause errors.

For Specific Syntaxes

msg = decode(code,n,k,'**hamming**/*fmt*',prim_poly) decodes code using the Hamming method. For this syntax, n must have the form 2^m-1 for some integer m greater than or equal to 3, and k must equal n-m. prim_poly is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive

polynomial for GF(2^m) that is used in the encoding process. The default value of `prim_poly` is `gfprimdf(m)`. The decoding table that the function uses to correct a single error in each codeword is `syndtable(hammgen(m))`.

`msg = decode(code,n,k,'linear/fmt',genmat,trt)` decodes `code`, which is a linear block code determined by the `k`-by-`n` generator matrix `genmat`. `genmat` is required as input. `decode` tries to correct errors using the decoding table `trt`, where `trt` is a `2^(n-k)`-by-`n` matrix.

`msg = decode(code,n,k,'cyclic/fmt',genpoly,trt)` decodes the cyclic code `code` and tries to correct errors using the decoding table `trt`, where `trt` is a `2^(n-k)`-by-`n` matrix. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial of the code. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an [n,k] cyclic code must have degree n-k and must divide x^n-1.

`msg = decode(code,n,k,'bch/fmt',t,prim_poly)` decodes `code` using the BCH method. `prim_poly` is a row vector that gives the coefficients, in order of ascending powers, of the primitive polynomial for GF(2^m) that will be used during processing. The default value of `prim_poly` is `gfprimdf(m)`. For this syntax, `n` must have the form 2^m-1 for some integer m greater than or equal to 3. `k` and `t` must be a valid message length and error-correction capability, respectively, as reported in the second and third columns of a row of `params` in the command

```
params = bchpoly(n)
```

`msg = decode(code,n,k)` is the same as
`msg = decode(code,n,k,'hamming/binary')`.

`[msg,err] = decode(...)` returns a column vector `err` that gives information about error correction. If the code is a convolutional code, then `err` contains the metric calculations used in the decoding decision process. For other types of codes, a nonnegative integer in the rth row of `err` (or the rth row of `vec2mat(err,k)` if `code` is a column vector) indicates the number of errors corrected in the rth *message* word; a negative integer indicates that there are more errors in the rth word than can be corrected.

`[msg,err,ccode] = decode(...)` returns the corrected code in `ccode`.

`[msg,err,ccode,cerr] = decode(...)` returns a column vector `cerr` whose meaning depends on the format of `code`:

- If `code` is a binary vector, then a nonnegative integer in the rth row of `vec2mat(cerr,n)` indicates the number of errors corrected in the rth *codeword*; a negative integer indicates that there are more errors in the rth codeword than can be corrected.
- If `code` is not a binary vector, then `cerr = err`.

Examples

On the reference page for `encode`, some of the example code illustrates the use of the `decode` function.

The example below illustrates the use of `err` and `cerr` when the coding method is not convolutional code and the code is a binary vector. The script encodes two five-bit messages using BCH code. Each codeword has fifteen bits. Errors are added to the first two bits of the first codeword and the first bit of the second codeword. Then `decode` is used to recover the original message. As a result, the errors are corrected. `err` is the same size as `msg` and `cerr` is the same size as `code`. `err` reflects the fact that the first *message* was recovered after correcting two errors, while the second message was recovered after correcting one error. `cerr` reflects the fact that the first *codeword* was decoded after correcting two errors, while the second codeword was decoded after correcting one error.

```
m = 4; n = 2^m-1; % Codeword length is 15.
k = 5; % Valid message length for BCH code when n = 15
t = 3; % Corresponding error-correction capability
msg = ones(10,1); % Two messages, five bits each
code = encode(msg,n,k,'bch');  % Encode the message.
% Now place two errors in first word and one error
% in the second word.  Create errors by reversing bits.
noisycode = code;
noisycode(1:2) = bitxor(noisycode(1:2),[1 1]');
noisycode(16) = bitxor(noisycode(16),1);
% Decode and try to correct the errors.
[newmsg,err,ccode,cerr] = decode(noisycode,n,k,'bch',t);
disp('Transpose of err is'); disp(err')
disp('Transpose of cerr is'); disp(cerr')
```

The output is below.

```
Transpose of err is
     2     2     2     2     2     1     1     1     1     1

Transpose of cerr is
  Columns 1 through 12

     2     2     2     2     2     2     2     2     2     2     2     2

  Columns 13 through 24

     2     2     2     1     1     1     1     1     1     1     1     1

  Columns 25 through 30

     1     1     1     1     1     1
```

Algorithm

Depending on the decoding method, `decode` relies on such lower-level functions as `hammgen`, `syndtable`, `cyclgen`, and `bchdeco`.

See Also

`encode`, `bchpoly`, `cyclpoly`, `syndtable`, `gen2par`, `bchdeco`

Purpose

Demap a digital message from a demodulated signal

Syntax

```
z = demodmap(y,Fd,Fs,'ask',M);
z = demodmap(y,Fd,Fs,'fsk',M,tone);
z = demodmap(y,Fd,Fs,'msk');
z = demodmap(y,Fd,Fs,'psk',M);
z = demodmap(y,Fd,Fs,'qask',M);
z = demodmap(y,Fd,Fs,'qask/arb',inphase,quadr);
z = demodmap(y,Fd,Fs,'qask/cir',numsig,amp,phs);
z = demodmap(y,[Fd offset],Fs,...)
```

Optional Inputs

Input	Default Value
`tone`	`Fd`
`amp`	`[1:length(numsig)]`
`phs`	`numsig*0`

Description

The digital demodulation process consists of two steps: demodulating an analog signal and demapping the demodulated signal to a digital signal. You can perform the first step using `ademod`, `ademodce`, or your own custom demodulator. The function `demodmap` performs the second step. The table below lists the demodulation schemes that `demodmap` supports.

Demodulation Scheme	Fourth Input Argument
M-ary amplitude shift keying	`'ask'`
M-ary frequency shift keying	`'fsk'`
Minimum shift keying	`'msk'`
M-ary phase shift keying	`'psk'`
Quadrature amplitude shift keying	`'qask'`, `'qask/arb'`, or `'qask/cir'`

To Demap a Digital Signal (General Information)

The generic syntax `z = demodmap(y,Fd,Fs,...)` demaps the digital message signal `z` from a received analog signal `y`. After measuring the distance from the

received signal to all possible digits in the coding scheme, the demapper returns the nearest digit.

`y` is a matrix. The sizes of `y` and `z` depend on the demodulation method:

- **(ASK, FSK, MSK methods)** If `y` is a vector of length `n*Fs/Fd`, then `z` is a column vector of length n. Otherwise, if `y` is (`n*Fs/Fd`)-by-m, then `z` is n-by-m and each column of `y` is processed separately.
- **(PSK, QASK methods)** `y` must have an even number of columns. The odd-numbered columns in `y` represent in-phase components and the even-numbered columns represent quadrature components. Each *pair* of columns of `y` is processed separately. If `y` is (`n*Fs/Fd`)-by-2m, then `z` is n-by-m.

The sampling rates in hertz of `y` and `z`, respectively, are `Fs` and `Fd`. (Thus `1/Fs` represents the time interval between two consecutive samples in `y`, and similarly for `z`.) The ratio `Fs/Fd` must be a positive integer. The time interval between two decision points is `1/Fd`.

To shift the decision times ahead by the integer `offset`, use the alternative syntax

```
z = demodmap(y,[Fd offset],...)
```

instead of the demapping syntaxes listed in this section and the next. The default decision offset is 0.

To Demap a Digital Signal (Specific Syntax Information)

`z = demodmap(y,Fd,Fs,'ask',M)` demaps from an `M`-ary amplitude shift keying signal constellation. Each entry of `z` is in the range [0, `M`-1].

`z = demodmap(y,Fd,Fs,'fsk',M,tone)` demaps using the coherent `M`-ary frequency shift keying method. The optional argument `tone` is the separation between successive frequencies in the modulated signal `y`. The default value of `tone` is `Fd`. Each entry of `z` is in the range [0, `M`-1].

`z = demodmap(y,Fd,Fs,'msk')` demaps using the minimum shift keying method. Each entry of `z` is either 0 or 1. The separation between the two frequencies is `Fd/2`.

`z = demodmap(y,Fd,Fs,'psk',M)` demaps from an `M`-ary phase shift keying signal constellation. Each entry of `z` is in the range [0, `M`-1].

`z = demodmap(y,Fd,Fs,'qask',M)` demaps from an `M`-ary quadrature amplitude shift keying square signal constellation. The table below shows the maximum among in-phase and quadrature coordinates of constellation points, for several small values of M.

M	Maximum of Coordinates of Constellation Points	M	Maximum of Coordinates of Constellation Points
2	1	32	5
4	1	64	7
8	3 (quadrature maximum = 1)	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)`.

`z = demodmap(y,Fd,Fs,'qask/arb',inphase,quadr)` demaps from a quadrature amplitude shift keying signal constellation that you define using the vectors `inphase` and `quadr`. The signal constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`z = demodmap(y,Fd,Fs,'qask/cir',numsig,amp,phs)` demaps from a quadrature amplitude shift keying circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of the kth circle, `numsig`(k) is the number of constellation points on the kth circle, and `phs`(k) is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then

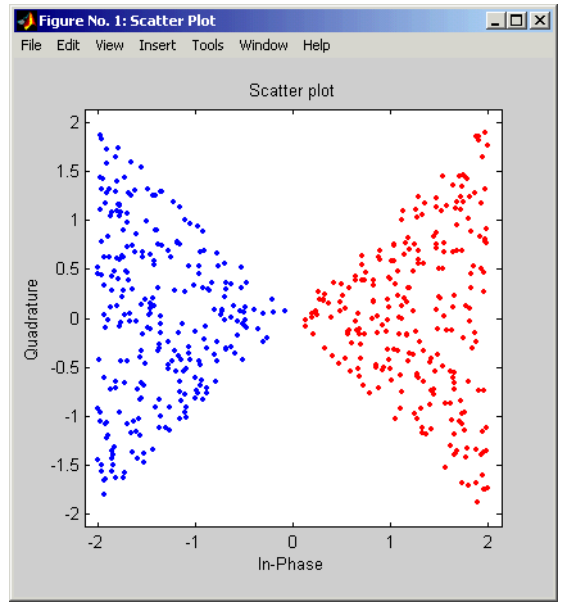
its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

Examples

The script below suggests which regions in the in-phase/quadrature plane are associated with different digits. It demaps random points, looks for points that were demapped to the digits 0 and 2, and plots those points in red and blue, respectively. The horizontal axis shows in-phase components and the vertical axis shows quadrature components.

```
% Construct [in-phase, quadrature] for random points.
y = 4*(rand(1000,2)-1/2);
% Demap to a digital signal, using 4-PSK method.
z = demodmap(y,1,1,'psk',4);
red = find(z==0); % Indices of points that mapped to the digit 0
h = scatterplot(y(red,:),1,0,'r.'); hold on  % Plot in red.
blue = find(z==2); % Indices of points that mapped to the digit 2
scatterplot(y(blue,:),1,0,'b.',h); hold off % Plot in blue.
```



See Also

`modmap`, `ddemod`, `ddemodce`, `ademod`, `ademodce`, `eyediagram`, `scatterplot`

Purpose

Discrete Fourier transform matrix in a Galois field

Syntax

```
dm = dftmtx(alph);
```

Description

`dm = dftmtx(alph)` returns a Galois array that represents the discrete Fourier transform operation on a Galois vector, with respect to the Galois scalar `alph`. The element `alph` is a primitive nth root of unity in the Galois field $GF(2^m) = GF(n+1)$; that is, n must be the smallest positive value of `k` for which `alph^k` equals 1. The discrete Fourier transform has size n and `dm` is an n-by-n array. The array `dm` represents the transform in the sense that `dm` times any length-n Galois column vector yields the transform of that vector.

Note The inverse discrete Fourier transform matrix is `dftmtx(1/alph)`.

Examples

The example below illustrates the discrete Fourier transform and its inverse, with respect to the element `gf(3,4)`. The example examines the first n powers of that element to make sure that only the nth power equals one. Afterward, the example transforms a random Galois vector, undoes the transform, and checks the result.

```
m = 4;
n = 2^m-1;
a = 3;
alph = gf(a,m);
mp = minpol(alph);
if (mp(1)==1 && isprimitive(mp)) % Check that alph has order n.
    disp('alph is a primitive nth root of unity.')
    dm = dftmtx(alph);
    idm = dftmtx(1/alph);
    x = gf(randint(n,1,2^m),m);
    y = dm*x; % Transform x.
    z = idm*y; % Recover x.
    ck = isequal(x,z)
end
```

The output is

```
alph is a primitive nth root of unity.
```

```
ck =

     1
```

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `alph` must be a primitive nth root of unity in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithm

The element `dm(a,b)` equals `alph^((a-1)*(b-1))`.

See Also

`fft`, `ifft`

dmod

Purpose

Digital passband modulator

Syntax

```
y = dmod(x,Fc,Fd,Fs,'method/nomap'...);
y = dmod(x,Fc,Fd,Fs,'ask',M);
y = dmod(x,Fc,Fd,Fs,'fsk',M,tone);
y = dmod(x,Fc,Fd,Fs,'msk');
y = dmod(x,Fc,Fd,Fs,'psk',M);
y = dmod(x,Fc,Fd,Fs,'qask',M);
y = dmod(x,Fc,Fd,Fs,'qask/arb',inphase,quadr);
y = dmod(x,Fc,Fd,Fs,'qask/cir',numsig,amp,phs);
y = dmod(x,Fc,Fd,[Fs initphase],...);
[y,t] = dmod(...);
```

Optional Inputs

Input	Default Value
`tone`	`Fd`
`amp`	`[1:length(numsig)]`
`phs`	`numsig*0`

Description

The function `dmod` performs digital passband modulation and some related tasks. The corresponding demodulation function is `ddemod`. The table below lists the modulation schemes that `dmod` supports.

Modulation Scheme	Fifth Input Argument
M-ary amplitude shift keying	`'ask'`
M-ary frequency shift keying	`'fsk'`
Minimum shift keying	`'msk'`
M-ary phase shift keying	`'psk'`
Quadrature amplitude shift keying	`'qask'`, `'qask/arb'`, or `'qask/cir'`

To Avoid the Mapping Process

Ordinarily, the `dmod` function first maps the digital message signal to an analog signal and then modulates the analog signal. The generic syntax

```
y = dmod(x,Fc,Fd,Fs,'method/nomap'...)
```

uses the **nomap** flag to tell dmod that the digital message has already been mapped to an analog signal x whose sampling rate is Fs. As a result, dmod skips its usual mapping step. You can use the modmap function to perform the mapping step. In this generic syntax, *method* is one of the seven values listed in the table above and the other variables are as in the next section.

To Modulate a Digital Signal (General Information)

The generic syntax `y = dmod(x,Fc,Fd,Fs,...)` modulates the digital message signal that x represents. x is a matrix of nonnegative integers. If x is a vector of length n, then y is a vector of length `n*Fs/Fd`. Otherwise, if x is n-by-m, then y is (`n*Fs/Fd`)-by-m and each column of x is processed separately.

Fc is the carrier frequency in hertz. The sampling rates in hertz of x and y, respectively, are Fd and Fs. (Thus 1/Fd represents the time interval between two consecutive samples in x, and similarly for y.) The ratio Fs/Fd must be a positive integer. For best results, use values such that Fs > Fc > Fd. The initial phase of the carrier signal is zero.

The generic syntax `y = dmod(x,Fc,Fd,[Fs initphase],...)` is the same, except that the fourth input argument is a two-element vector instead of a scalar. The first entry, Fs, is the sampling rate as described in the paragraph above. The second entry, initphase, is the initial phase of the carrier signal, measured in radians.

To Modulate a Digital Signal (Specific Syntax Information)

`y = dmod(x,Fc,Fd,Fs,'ask',M)` performs M-ary amplitude shift keying modulation. Each entry of x must be in the range [0, M-1]. The maximum value of the modulated signal is 1.

`y = dmod(x,Fc,Fd,Fs,'fsk',M,tone)` performs M-ary frequency shift keying modulation. Each entry of x must be in the range [0, M-1]. The optional argument tone is the separation between successive frequencies in the modulated signal y. The default value of tone is Fd. The maximum value of y is 1.

`y = dmod(x,Fc,Fd,Fs,'msk')` performs minimum shift keying modulation. Each entry of x is either 0 or 1. The maximum value of y is 1.

`y = dmod(x,Fc,Fd,Fs,'psk',M)` performs `M`-ary phase shift keying modulation. Each entry of `x` must be in the range [0, `M`-1]. The maximum value of `y` is 1.

`y = dmod(x,Fc,Fd,Fs,'qask',M)` performs `M`-ary quadrature amplitude shift keying modulation with a square signal constellation. The table below shows the maximum value of `y` for several small values of M.

M	Maximum Value of y	M	Maximum Value of y
2	1	32	5
4	1	64	7
8	3	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)`.

`y = dmod(x,Fc,Fd,Fs,'qask/arb',inphase,quadr)` performs quadrature amplitude shift keying modulation, with a signal constellation that you define using the vectors `inphase` and `quadr`. The constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`y = dmod(x,Fc,Fd,Fs,'qask/cir',numsig,amp,phs)` performs quadrature amplitude shift keying modulation with a circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of the kth circle, `numsig`(k) is the number of constellation points on the kth circle, and `phs`(k) is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

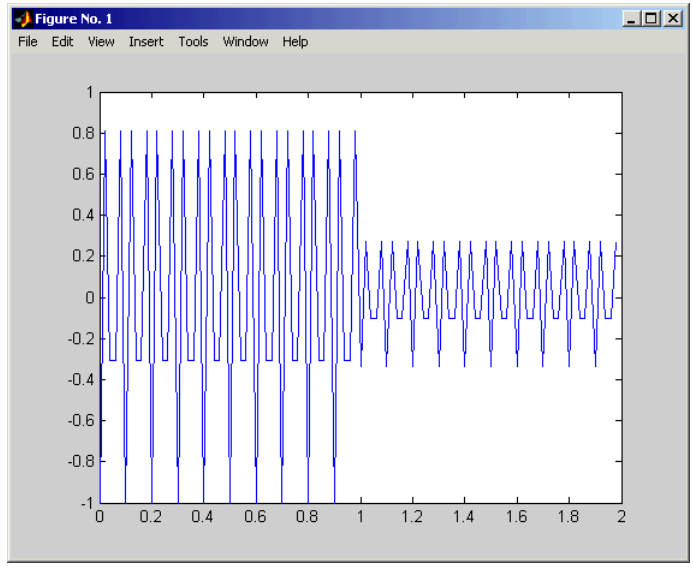
Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

`[y,t] = dmod(...)` returns the computation time in `t`. `t` is a vector whose length is the number of rows of `y`.

Examples

An example on the reference page for `ddemod` uses `dmod`. Also, the code below shows the waveforms used to communicate the digits 0 and 1 using 4-ASK modulation. Notice that the `dmod` command has two output arguments. The second output, `t`, is used to scale the horizontal axis in the plot.

```
Fc = 20; Fd = 10; Fs = 50;
M = 4; % Use 4-ASK modulation.
x = ones(Fd,1)*[0 1]; x=x(:);
% Modulate, keeping track of time.
[y,t] = dmod(x,Fc,Fd,Fs,'ask',M);
plot(t,y) % Plot signal versus time.
```



See Also

`ddemod`, `dmodce`, `ddemodce`, `amod`, `amodce`

dmodce

Purpose

Digital baseband modulator

Syntax

```
y = dmodce(x,Fd,Fs,'method/nomap'...);
y = dmodce(x,Fd,Fs,'ask',M);
y = dmodce(x,Fd,Fs,'fsk',M,tone);
y = dmodce(x,Fd,Fs,'msk');
y = dmodce(x,Fd,Fs,'psk',M);
y = dmodce(x,Fd,Fs,'qask',M);
y = dmodce(x,Fd,Fs,'qask/arb',inphase,quadr);
y = dmodce(x,Fd,Fs,'qask/cir',numsig,amp,phs);
y = dmodce(x,Fd,[Fs initphase],...);
```

Optional Inputs

Input	Default Value
`tone`	`Fd`
`amp`	`[1:length(numsig)]`
`phs`	`numsig*0`

Description

The function `dmodce` performs digital baseband modulation and some related tasks. The corresponding demodulation function is `ddemodce`. The table below lists the modulation schemes that `dmodce` supports.

Modulation Scheme	Fourth Input Argument
M-ary amplitude shift keying	'**ask**'
M-ary frequency shift keying	'**fsk**'
Minimum shift keying	'**msk**'
M-ary phase shift keying	'**psk**'
Quadrature amplitude shift keying	'**qask**', '**qask/arb**', or '**qask/cir**'

To Modulate Without Mapping

Ordinarily, the `dmodce` function first maps the digital message signal to an analog signal and then modulates the analog signal. The generic syntax

```
y = dmodce(x,Fd,Fs,'method/nomap'...)
```

uses the **/nomap** flag to tell dmodce that the digital message has already been mapped to an analog signal x whose sampling rate is Fs. As a result, dmodce skips its usual mapping step. You can use the modmap function to perform the mapping step. In this generic syntax, *method* is one of the seven values listed in the table above, and the other variables are as in the next section.

To Modulate a Digital Signal (General Information)

The generic syntax `y = dmodce(x,Fd,Fs,...)` modulates the digital message signal that x represents. x is a matrix of nonnegative integers. If x is a vector of length n, then y is a vector of length `n*Fs/Fd`. Otherwise, if x is n-by-m, then y is (`n*Fs/Fd`)-by-m and each column of x is processed separately. Because dmodce implements baseband simulation, the entries of y are *complex*.

The sampling rates in hertz of x and y, respectively, are Fd and Fs. (Thus 1/Fd represents the time interval between two consecutive samples in x, and similarly for y.) The ratio Fs/Fd must be a positive integer. The initial phase in the modulation is zero.

The generic syntax `y = dmodce(x,Fd,[Fs initphase],...)` is the same, except that the third input argument is a two-element vector instead of a scalar. The first entry, Fs, is the sampling rate as described in the paragraph above. The second entry, initphase, is the initial phase in the modulation, measured in radians.

To Modulate a Digital Signal (Specific Syntax Information)

`y = dmodce(x,Fd,Fs,'ask',M)` performs M-ary amplitude shift keying modulation. Each entry of x must be in the range [0, M-1]. The maximum value of the modulated signal is 1.

`y = dmodce(x,Fd,Fs,'fsk',M,tone)` performs M-ary frequency shift keying modulation. Each entry of x must be in the range [0, M-1]. The optional argument tone is the separation between successive frequencies in the modulated signal y. The default value of tone is Fd. The maximum value of y is 1.

`y = dmodce(x,Fd,Fs,'msk')` performs minimum shift keying modulation. Each entry of x is either 0 or 1. The maximum value of y is 1. The separation between the two frequencies is Fd/2.

`y = dmodce(x,Fd,Fs,'psk',M)` performs `M`-ary phase shift keying modulation. Each entry of `x` must be in the range [0, `M`-1]. The maximum value of `y` is 1.

`y = dmodce(x,Fd,Fs,'qask',M)` performs `M`-ary quadrature amplitude shift keying modulation with a square signal constellation. The table below shows the maximum value of `y` for several small values of M.

M	Maximum Value of y	M	Maximum Value of y
2	1	32	5
4	1	64	7
8	3	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)`.

`y = dmodce(x,Fd,Fs,'qask/arb',inphase,quadr)` performs quadrature amplitude shift keying modulation, with a signal constellation that you define using the vectors `inphase` and `quadr`. The constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`y = dmodce(x,Fd,Fs,'qask/cir',numsig,amp,phs)` performs quadrature amplitude shift keying modulation with a circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of the kth circle, `numsig`(k) is the number of constellation points on the kth circle, and `phs`(k) is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

Examples

This example uses FSK modulation and demodulation with different values of the frequency separation, `tone`. The output indicates that the symbol error rate varies depending on the value of `tone`. Your results might be different from those shown below, because the example uses random numbers.

```
M = 4; Fd = 1; Fs = 32;
SNRperBit = 5;
adjSNR = SNRperBit-10*log10(Fs/Fd)+10*log10(log2(M));
x = randint(5000,1,M); % Original signal
% Modulate using FSK with orthogonal tone spacing.
tone = .5;
randn('state',1945724); % Seed the Gaussian generator.
w1 = dmodce(x,Fd,Fs,'fsk',M,tone);
y1  = awgn(w1, adjSNR, 'measured', [], 'dB');
z1 = ddemodce(y1,Fd,Fs,'fsk',M,tone);
ser1 = symerr(x,z1)

% Modulate using FSK with nonorthogonal tone spacing.
tone = .25;
randn('state',1945724); % Reseed the Gaussian generator.
w2 = dmodce(x,Fd,Fs,'fsk',M,tone);
y2  = awgn(w2, adjSNR, 'measured', [], 'dB');
z2 = ddemodce(y2,Fd,Fs,'fsk',M,tone);
ser2 = symerr(x,z2)
```

The output is

```
ser1 =

    67


ser2 =

   258
```

See Also ddemodce, dmod, ddemod, amod, amodce, modmap, apkconst

Purpose

Decode using differential pulse code modulation

Syntax

```
sig = dpcmdeco(indx,codebook,predictor);
[sig,quanterror] = dpcmdeco(indx,codebook,predictor);
```

Description

`sig = dpcmdeco(indx,codebook,predictor)` implements differential pulse code demodulation to decode the vector `indx`. The vector `codebook` represents the predictive-error quantization codebook. The vector `predictor` specifies the predictive transfer function. If the transfer function has predictive order M, then `predictor` has length M+1 and an initial entry of 0. To decode correctly, use the same codebook and predictor in `dpcmenco` and `dpcmdeco`.

See either "Representing Quantization Parameters" on page 2-13 or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[sig,quanterror] = dpcmdeco(indx,codebook,predictor)` is the same as the syntax above, except that the vector `quanterror` is the quantization of the predictive error based on the quantization parameters. `quanterror` is the same size as `sig`.

Note You can estimate the input parameters `codebook`, `partition`, and `predictor` using the function `dpcmopt`.

Examples

See "Example: DPCM Encoding and Decoding" on page 2-19 and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 2-20 for examples that use `dpcmdeco`.

See Also

`quantiz`, `dpcmopt`, `dpcmenco`

References

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

dpcmenco

Purpose

Encode using differential pulse code modulation

Syntax

```
indx = dpcmenco(sig,codebook,partition,predictor)
[indx,quants] = dpcmenco(sig,codebook,partition,predictor)
```

Description

`indx = dpcmenco(sig,codebook,partition,predictor)` implements differential pulse code modulation to encode the vector `sig`. `partition` is a vector whose entries give the endpoints of the partition intervals. `codebook`, a vector whose length exceeds the length of `partition` by one, prescribes a value for each partition in the quantization. `predictor` specifies the predictive transfer function. If the transfer function has predictive order M, then `predictor` has length M+1 and an initial entry of 0. The output vector `indx` is the quantization index.

See "Implementing Differential Pulse Code Modulation" on page 2-18 for more about the format of `predictor`. See either "Representing Quantization Parameters" on page 2-13 or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[indx,quants] = dpcmenco(sig,codebook,partition,predictor)` is the same as the syntax above, except that `quants` contains the quantization of `sig` based on the quantization parameters. `quants` is a vector of the same size as `sig`.

Note If `predictor` is an order-one transfer function, then the modulation is called a delta modulation.

Examples

See "Example: DPCM Encoding and Decoding" on page 2-19 and "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 2-20 for examples that use `dpcmenco`.

See Also

`quantiz`, `dpcmopt`,`dpcmdeco`

References

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

Purpose

Optimize differential pulse code modulation parameters

Syntax

```
predictor = dpcmopt(training_set,ord);
[predictor,codebook,partition] = dpcmopt(training_set,ord,len);
[predictor,codebook,partition] = ...
   dpcmopt(training_set,ord,initcodebook);
```

Description

`predictor = dpcmopt(training_set,ord)` returns a vector representing a predictive transfer function of order `ord` that is appropriate for the training data in the vector `training_set`. `predictor` is a row vector of length `ord+1`. See "Representing Quantization Parameters" on page 2-13 for more about its format.

Note `dpcmopt` optimizes for the data in `training_set`. For best results, `training_set` should be similar to the data that you plan to quantize.

`[predictor,codebook,partition] = dpcmopt(training_set,ord,len)` is the same as the syntax above, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `len` is an integer that prescribes the length of `codebook`. `partition` is a vector of length `len-1`. See either "Representing Quantization Parameters" on page 2-13 or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

`[predictor,codebook,partition] = dpcmopt(training_set,ord,initcodebook)` is the same as the first syntax, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `initcodebook`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `initcodebook`. The output `partition` is a vector whose length is one less than the length of `codebook`.

Examples

See "Example: Comparing Optimized and Nonoptimized DPCM Parameters" on page 2-20 for an example that uses `dpcmopt`.

See Also

`dpcmenco`, `dpcmdeco`, `quantiz`, `lloyds`

encode

Purpose

Block encoder

Syntax

```
code = encode(msg,n,k,'linear/fmt',genmat);
code = encode(msg,n,k,'cyclic/fmt',genpoly);
code = encode(msg,n,k,'bch/fmt',genpoly);
code = encode(msg,n,k,'hamming/fmt',prim_poly);
code = encode(msg,n,k);
[code,added] = encode(...);
```

Optional Inputs

Input	Default Value
fmt	**binary**
genpoly	cyclpoly(n,k) for cyclic codes; bchpoly(n,k) for BCH codes
prim_poly	gfprimdf(n-k)

Description

For All Syntaxes

The encode function encodes messages using one of the following error-correction coding methods:

- Linear block
- Cyclic
- BCH (Bose, Ray-Chaudhuri, Hocquenghem)
- Hamming

For all of these methods, the codeword length is n and the message length is k.

msg, which represents the messages, can have one of several formats. The table "Information Formats" below shows which formats are allowed for msg, how the argument *fmt* should reflect the format of msg, and how the format of the output code depends on these choices. The examples in the table are for k = 4. If *fmt* is not specified as input, then its default value is **binary**.

Note If `2^n` or `2^k` is large, then you should use the default **`binary`** format instead of the **`decimal`** format. This is because the function uses a binary format internally, while the roundoff error associated with converting many bits to large decimal numbers and back might be substantial.

Information Formats

Format of msg	Value of "fmt" Argument	Format of code
Binary column vector	**`binary`**	Binary column vector
Example: `msg = [0 1 1 0, 0 1 0 1, 1 0 0 1]'`		
Binary matrix with `k` columns	**`binary`**	Binary matrix with `n` columns
Example: `msg = [0 1 1 0; 0 1 0 1; 1 0 0 1]`		
Column vector of integers in the range [0, `2^k-1`]	**`decimal`**	Column vector of integers in the range [0, `2^n-1`]
Example: `msg = [6, 10, 9]'`		

For Specific Syntaxes

`code = encode(msg,n,k,'`**`linear`**`/`*`fmt`*`',genmat)` encodes `msg` using `genmat` as the generator matrix for the linear block encoding method. `genmat`, a `k`-by-`n` matrix, is required as input.

`code = encode(msg,n,k,'`**`cyclic`**`/`*`fmt`*`',genpoly)` encodes `msg` and creates a systematic cyclic code. `genpoly` is a row vector that gives the coefficients, in order of ascending powers, of the binary generator polynomial. The default value of `genpoly` is `cyclpoly(n,k)`. By definition, the generator polynomial for an [n,k] cyclic code must have degree n-k and must divide x^n-1.

`code = encode(msg,n,k,'`**`bch`**`/`*`fmt`*`',genpoly)` encodes `msg` using the BCH encoding method. `genpoly` is a row vector that gives the coefficients, in order of

ascending powers, of the degree-(n-k) binary BCH generator polynomial. The default value of genpoly is bchpoly(n,k). For this syntax, n must have the form 2^m-1 for some integer m greater than or equal to 3. k must be a valid message length as reported in the second column of params in the command

```
params = bchpoly(n)
```

code = encode(msg,n,k,'**hamming**/*fmt*',prim_poly) encodes msg using the Hamming encoding method. For this syntax, n must have the form 2^m-1 for some integer m greater than or equal to 3, and k must equal n-m. prim_poly is a row vector that gives the binary coefficients, in order of ascending powers, of the primitive polynomial for GF(2^m) that is used in the encoding process. The default value of prim_poly is the default primitive polynomial gfprimdf(m).

code = encode(msg,n,k) is the same as code = encode(msg,n,k,'**hamming**/**binary**').

[code,added] = encode(...) returns the additional variable added. added is the number of zeros that were placed at the end of the message matrix before encoding, in order for the matrix to have the appropriate shape. "Appropriate" depends on n, k, the shape of msg, and the encoding method.

Examples

The example below illustrates the three different information formats (binary vector, binary matrix, and decimal vector) for Hamming code. The three messages have identical content in different formats; as a result, the three codes that encode creates have identical content in correspondingly different formats.

```
m = 4; n = 2^m-1; % Codeword length = 15
k = 11; % Message length

% Create 100 messages, k bits each.
msg1 = randint(100*k,1,[0,1]); % As a column vector
msg2 = vec2mat(msg1,k); % As a k-column matrix
msg3 = bi2de(msg2); % As a column of decimal integers

% Create 100 codewords, n bits each.
code1 = encode(msg1,n,k,'hamming/binary');
code2 = encode(msg2,n,k,'hamming/binary');
code3 = encode(msg3,n,k,'hamming/decimal');
```

```
if ( vec2mat(code1,n)==code2 & de2bi(code3,n)==code2 )
   disp('All three formats produced the same content.')
end
```

The output is

```
All three formats produced the same content.
```

The next example creates a cyclic code, adds noise, and then decodes the noisy code. It uses the `decode` function.

```
n = 3; k = 2; % A (3,2) cyclic code
msg = randint(100,k,[0,1]); % 100 messages, k bits each
code = encode(msg,n,k,'cyclic/binary');
% Add noise.
noisycode = rem(code + randerr(100,n,[0 1;.7 .3]), 2);
newmsg = decode(noisycode,n,k,'cyclic'); % Try to decode.
% Compute error rate for decoding the noisy code.
[number,ratio] = biterr(newmsg,msg);
disp(['The bit error rate is ',num2str(ratio)])
```

The output is below. Your error rate results might vary because the noise is random.

```
The bit error rate is 0.08
```

The next example encodes the same message using Hamming, BCH, and cyclic methods. Before creating BCH code, it uses the `bchpoly` command to find out what codeword and message lengths are valid. This example also creates Hamming code with the '**`linear`**' option of the `encode` command. It then decodes each code and recovers the original message.

```
n = 6; % Try codeword length = 6.
% Find any valid message length for BCH code.
params = bchpoly(n);
n = params(1,1); % Redefine codeword length in case earlier one
% was invalid.
k = params(1,2); % Message length
m = log2(n+1); % Express n as 2^m-1.
msg = randint(100,1,[0,2^k-1]); % Column of decimal integers

% Create various codes.
codehamming = encode(msg,n,k,'hamming/decimal');
```

```
[parmat,genmat] = hammgen(m);
codehamming2 = encode(msg,n,k,'linear/decimal',genmat);
if codehamming==codehamming2
   disp('The ''linear'' method can create Hamming code.')
end
codebch = encode(msg,n,k,'bch/decimal');
codecyclic = encode(msg,n,k,'cyclic/decimal');

% Decode to recover the original message.
decodedhamming = decode(codehamming,n,k,'hamming/decimal');
decodedbch = decode(codebch,n,k,'bch/decimal');
decodedcyclic = decode(codecyclic,n,k,'cyclic/decimal');
if (decodedhamming==msg & decodedbch==msg & decodedcyclic==msg)
  disp('All decoding worked flawlessly in this noiseless world.')
end
```

The output is

```
The 'linear' method can create Hamming code.
All decoding worked flawlessly in this noiseless world.
```

Algorithm

Depending on the encoding method, `encode` relies on such lower-level functions as `hammgen`, `cyclgen`, and `bchenco`.

See Also

`decode`, `bchpoly`, `cyclpoly`, `cyclgen`, `hammgen`, `bchenco`

Purpose

Generate an eye diagram

Syntax

```
eyediagram(x,n);
eyediagram(x,n,period);
eyediagram(x,n,period,offset);
eyediagram(x,n,period,offset,plotstring);
eyediagram(x,n,period,offset,plotstring,h);
h = eyediagram(...);
```

Description

`eyediagram(x,n)` creates an eye diagram for the signal `x`, plotting `n` samples in each trace. `n` must be an integer greater than 1. The labels on the horizontal axis of the diagram range between -1/2 and 1/2. The function assumes that the first value of the signal, and every `n`th value thereafter, occur at integer times. The interpretation of `x` and the number of plots depend on the shape and complexity of `x`:

- If `x` is a real two-column matrix, then `eyediagram` interprets the first column as in-phase components and the second column as quadrature components. The two components appear in different subplots of a single figure window.
- If `x` is a complex vector, then `eyediagram` interprets the real part as in-phase components and the imaginary part as quadrature components. The two components appear in different subplots of a single figure window.
- If `x` is a real vector, then `eyediagram` interprets it as a real signal. The figure window contains a single plot.

`eyediagram(x,n,period)` is the same as the syntax above, except that the labels on the horizontal axis range between `-period/2` and `period/2`.

`eyediagram(x,n,period,offset)` is the same as the syntax above, except that the function assumes that the `(offset+1)`st value of the signal, and every `n`th value thereafter, occur at times that are integer multiples of `period`. The variable `offset` must be a nonnegative integer between 0 and `n-1`.

`eyediagram(x,n,period,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function. The default string is `'b-'`, which produces a blue solid line.

`eyediagram(x,n,period,offset,plotstring,h)` is the same as the syntax above, except that the eye diagram is in the figure whose handle is `h`, rather than a new figure. `h` must be a handle to a figure that `eyediagram` previously generated.

Note You cannot use `hold on` to plot multiple signals in the same figure.

`h = eyediagram(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the eye diagram.

Examples

See “Example: Eye Diagrams” on page 2-8 for an example. For an online demonstration, type `playshow scattereyedemo`.

See Also

`scatterplot`, `plot`, `scattereyedemo`

Purpose

Discrete Fourier transform

Syntax

```
fft(x)
```

Description

`fft(x)` is the discrete Fourier transform (DFT) of the Galois vector `x`. If `x` is in the Galois field GF(2^m), then the length of `x` must be 2^m-1.

Examples

```
m = 4;
n = 2^m-1;
x = gf(randint(n,1,2^m),m); % Random vector
y = fft(x); % Transform of x
z = ifft(y); % Inverse transform of y
ck = isequal(z,x) % Check that ifft(fft(x)) recovers x.

ck =

     1
```

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `x` must be in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithm

If `x` is a column vector, then `fft` applies `dftmtx` to the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`ifft`, `dftmtx`

filter

Purpose

One-dimensional digital filter over a Galois field

Syntax

```
y = filter(b,a,x);
[y,zf] = filter(b,a,x);
```

Description

`y = filter(b,a,x)` filters the data in the vector `x` with the filter described by numerator coefficient vector `b` and denominator coefficient vector `a`. The vectors `b`, `a`, and `x` must be Galois vectors in the same field. If `a(1)` is not equal to `1`, then `filter` normalizes the filter coefficients by `a(1)`. As a result, `a(1)` must be nonzero.

The filter is a "Direct Form II Transposed" implementation of the standard difference equation below.

```
a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb) ...
                      - a(2)*y(n-1) - ... - a(na+1)*y(n-na)
```

`[y,zf] = filter(b,a,x)` returns the final conditions of the filter delays in the Galois vector `zf`. The length of the vector `zf` is `max(size(a),size(b))-1`.

Examples

An example is in "Filtering" on page 2-114.

Purpose

Convert between parity-check and generator matrices

Syntax

```
parmat = gen2par(genmat);
genmat = gen2par(parmat);
```

Description

`parmat = gen2par(genmat)` converts the standard-form binary generator matrix `genmat` into the corresponding parity-check matrix `parmat`.

`genmat = gen2par(parmat)` converts the standard-form binary parity-check matrix `parmat` into the corresponding generator matrix `genmat`.

The standard forms of the generator and parity-check matrices for an [n,k] binary linear block code are shown in the table below.

Type of Matrix	Standard Form	Dimensions
Generator	$[I_k\ P]$ or $[P\ I_k]$	k-by-n
Parity-check	$[-P'\ I_{n-k}]$ or $[I_{n-k}\ -P']$	(n-k)-by-n

where I_k is the identity matrix of size k and the `'` symbol indicates matrix transpose. Two standard forms are listed for each type, because different authors use different conventions. For *binary* codes, the minus signs in the parity-check form listed above are irrelevant; that is, -1 = 1 in the binary field.

Examples

The commands below convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

```
parmat = hammgen(3)
genmat = gen2par(parmat)
parmat2 = gen2par(genmat) % Ans should be the same as parmat above
```

The output is

```
parmat =

     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1
```

```
genmat =

     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

parmat2 =

     1     0     0     1     0     1     1
     0     1     0     1     1     1     0
     0     0     1     0     1     1     1
```

See Also

cyclgen, hammgen

Purpose

Create a Galois field array

Syntax

```
x_gf = gf(x,m);
x_gf = gf(x,m,prim_poly);
x_gf = gf(x);
```

Description

`x_gf = gf(x,m)` creates a Galois field array from the matrix `x`. The Galois field has `2^m` elements, where `m` is an integer between 1 and 16. The elements of `x` must be integers between 0 and `2^m-1`. The output `x_gf` is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. As a result, when you manipulate `x_gf` using operators or functions such as `+` or `det`, MATLAB works within the Galois field you have specified.

Note To learn how to manipulate `x_gf` using familiar MATLAB operators and functions, see "Galois Field Computations" on page 2-93. To learn how the integers in `x` represent elements of GF(`2^m`), see "How Integers Correspond to Galois Field Elements" on page 2-97.

`x_gf = gf(x,m,prim_poly)` is the same as the previous syntax, except that it uses the primitive polynomial `prim_poly` to define the field. `prim_poly` is the integer representation of a primitive polynomial. For example, the number 41 represents the polynomial D^5+D^2+1 because the binary form of 41 is 1 0 0 1 0 1. For more information about the primitive polynomial, see "Specifying the Primitive Polynomial" on page 2-99.

`x_gf = gf(x)` creates a GF(2) array from the matrix `x`. Each element of `x` must be 0 or 1.

Default Primitive Polynomials

The table below lists the primitive polynomial that `gf` uses by default for each Galois field GF(`2^m`). To use a different primitive polynomial, specify `prim_poly` as an input argument when you invoke `gf`.

m	Default Primitive Polynomial	Integer Representation
1	D + 1	3
2	D^2 + D + 1	7
3	D^3 + D + 1	11
4	D^4 + D + 1	19
5	D^5 + D^2 + 1	37
6	D^6 + D + 1	67
7	D^7 + D^3 + 1	137
8	D^8 + D^4 + D^3 + D^2 + 1	285
9	D^9 + D^4 + 1	529
10	D^10 + D^3 + 1	1033
11	D^11 + D^2 + 1	2053
12	D^12 + D^6 + D^4 + D + 1	4179
13	D^13 + D^4 + D^3 + D + 1	8219
14	D^14 + D^10 + D^6 + D + 1	17475
15	D^15 + D + 1	32771
16	D^16 + D^12 + D^3 + D + 1	69643

Examples

For examples that use `gf`, see

- “Example: Creating Galois Field Variables” on page 2-95
- “Example: Representing a Primitive Element” on page 2-98

- Other sample code within “Galois Field Computations” on page 2-93
- The Galois field demonstration: type `playshow gfdemo`.

See Also

`gftable`, list of functions and operators for Galois field computations, `gfdemo`

gfadd

Purpose

Add polynomials over a Galois field

Syntax

```
c = gfadd(a,b,p);
c = gfadd(a,b,p,len);
c = gfadd(a,b,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the + operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction" on page 2-103.

`c = gfadd(a,b,p)` adds two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1. If `a` and `b` are matrices of the same size, then the function treats each row independently.

`c = gfadd(a,b,p,len)` adds row vectors `a` and `b` as in the previous syntax, except that it returns a row vector of length `len`. The output `c` is a truncated or extended representation of the sum. If the row vector corresponding to the sum has fewer than `len` entries (including zeros), then extra zeros are added at the end; if it has more than `len` entries, then entries from the end are removed.

`c = gfadd(a,b,field)` adds two GF(p^m) elements, where m is a positive integer. `a` and `b` are the exponential format of the two elements, relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the sum, relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

In the code below, `sum5` is the sum of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and `linpart` is the degree-one part of `sum5`.

```
sum5 = gfadd([2 3 1],[4 2 3],5)
```

```
sum5 =

     1     0     4

linpart = gfadd([2 3 1],[4 2 3],5,2)

linpart =

     1     0
```

The code below shows that $A^2 + A^4 = A^1$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
g = gfadd(2,4,field)

g =

     1
```

Other examples are in "Arithmetic in Galois Fields" on page A-12.

See Also

`gfsub`, `gfconv`, `gfmul`, `gfdeconv`, `gfdiv`, `gftuple`

gfconv

Purpose

Multiply polynomials over a Galois field

Syntax

```
c = gfconv(a,b,p);
c = gfconv(a,b,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `conv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials" on page 2-117.

The `gfconv` function multiplies polynomials over a Galois field. (To multiply elements of a Galois field, use `gfmul` instead.) Algebraically, multiplying polynomials over a Galois field is equivalent to convolving vectors containing the polynomials' coefficients, where the convolution operation uses arithmetic over the same Galois field.

`c = gfconv(a,b,p)` multiplies two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1.

`c = gfconv(a,b,field)` multiplies two GF(p^m) polynomials, where p is a prime number and m is a positive integer. `a`, `b`, and `c` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats.

Examples

The command below shows that $(1 + x + x^4)(x + x^2) = x + 2x^2 + x^3 + x^5 + x^6$ over GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)

gfc =


     0     1     2     1     0     1     1
```

The code below illustrates the identity

$$(x^r + x^s)^p = x^{rp} + x^{sp} \text{ in GF}(p)$$

for the case in which $p = 7$, $r = 5$, and $s = 3$. (The identity holds when p is any prime number, and r and s are positive integers.)

```
p = 7; r = 5; s = 3;
a = gfrepcov([r s]); % x^r + x^s

% Compute a^p over GF(p).
c = 1;
for ii = 1:p
   c = gfconv(c,a,p);
end;

% Check whether c = x^(rp) + x^(sp).
powers = [];
for ii = 1:length(c)
   if c(ii)~=0
      powers = [powers, ii];
   end;
end;
if (powers==[r*p+1 s*p+1] | powers==[s*p+1 r*p+1])
   disp('The identity is proved for this case of r, s, and p.')
end
```

See Also

gfdeconv, gfadd, gfsub, gfmul, gftuple

gfcosets

Purpose

Produce cyclotomic cosets for a Galois field

Syntax

```
c = gfcosets(m,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `cosets` function.

`c = gfcosets(m,p)` produces the cyclotomic cosets for GF(`p^m`), where `m` is a positive integer and `p` is a prime number.

The output matrix `c` is structured so that each row represents one coset. The row represents the coset by giving the exponential format of the elements of the coset, relative to the default primitive polynomial for the field. For a description of exponential formats, see "Representing Elements of Galois Fields" on page A-3.

The first column contains the coset leaders. Because the lengths of cosets might vary, entries of `NaN` are used to fill the extra spaces when necessary to make `c` rectangular.

A cyclotomic coset is a set of elements that all satisfy the same minimal polynomial. For more details on cyclotomic cosets, see the works listed in "References" below.

Examples

The command below finds the cyclotomic cosets for GF(9).

```
c = gfcosets(2,3)

c =

     0   NaN
     1     3
     2     6
     4   NaN
     5     7
```

The `gfminpol` function can check that the elements of, for example, the third row of `c` indeed belong in the same coset.

```
m = [gfminpol(2,2,3); gfminpol(6,2,3)] % Rows are identical.

m =

     2     0     1
     2     0     1
```

See Also

`gfminpol`, `gfprimdf`, `gfroots`

References

Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, Mass., Addison-Wesley, 1983, p. 105.

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

gfdeconv

Purpose

Divide polynomials over a Galois field

Syntax

```
[quot,remd] = gfdeconv(b,a,p);
[quot,remd] = gfdeconv(b,a,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `deconv` function with Galois arrays. For details, see "Multiplication and Division of Polynomials" on page 2-117.

The `gfdeconv` function divides polynomials over a Galois field. (To divide elements of a Galois field, use `gfdiv` instead.) Algebraically, dividing polynomials over a Galois field is equivalent to deconvolving vectors containing the polynomials' coefficients, where the deconvolution operation uses arithmetic over the same Galois field.

`[quot,remd] = gfdeconv(b,a,p)` divides the polynomial `b` by the polynomial `a` over GF(`p`) and returns the quotient in `quot` and the remainder in `remd`. `p` is a prime number. `b`, `a`, `quot`, and `remd` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1.

`[quot,remd] = gfdeconv(b,a,field)` divides the polynomial `b` by the polynomial `a` over GF(p^m) and returns the quotient in `quot` and the remainder in `remd`. Here p is a prime number and m is a positive integer. `b`, `a`, `quot`, and `remd` are row vectors that list the exponential formats of the coefficients of the corresponding polynomials, in order of ascending powers. The exponential format is relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats.

Examples

The code below shows that

$$(x + x^3 + x^4) \div (1 + x) = 1 + x^3 \text{ Remainder } 2$$

in GF(3). It also checks the results of the division.

```
p = 3;
b = [0 1 0 1 1]; a = [1 1];
```

```
[quot, remd] = gfdeconv(b,a,p)
% Check the result.
bnew = gfadd(gfconv(quot,a,p),remd,p);
if isequal(bnew,b)
   disp('Correct.')
end;
```

The output is below.

```
quot =

     1     0     0     1


remd =

     2

Correct.
```

Working over GF(3), the code below outputs those polynomials of the form $x^k - 1$ (k = 2, 3, 4,..., 8) that $1 + x^2$ divides evenly.

```
p = 3; m = 2;
a = [1 0 1]; % 1+x^2
for ii = 2:p^m-1
   b = gfrepcov(ii); % x^ii
   b(1) = p-1; % -1+x^ii
   [quot, remd] = gfdeconv(b,a,p);
   % Display -1+x^ii if a divides it evenly.
   if remd==0
      gfpretty(b)
   end
end
```

The output is below.

```
     4
2 + X

     8
2 + X
```

In light of the discussion in "Algorithm" on the reference page for `gfprimck` along with the irreducibility of $1 + x^2$ over GF(3), this output indicates that $1 + x^2$ is not primitive for GF(9).

Algorithm

The algorithm of `gfdeconv` is similar to that of the MATLAB function `deconv`.

See Also

`gfconv`, `gfadd`, `gfsub`, `gfdiv`, `gftuple`

Purpose

Divide elements of a Galois field

Syntax

```
quot = gfdiv(b,a,p);
quot = gfdiv(b,a,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the `./` operator to Galois arrays. For details, see "Example: Division" on page 2-105.

The `gfdiv` function divides elements of a Galois field. (To divide polynomials over a Galois field, use `gfdeconv` instead.)

`quot = gfdiv(b,a,p)` divides b by a in GF(`p`) and returns the quotient. `p` is a prime number. If `a` and `b` are matrices of the same size, then the function treats each element independently. All entries of `b`, `a`, and `quot` are between 0 and `p`-1.

`quot = gfdiv(b,a,field)` divides b by a in GF(p^m) and returns the quotient. p is a prime number and m is a positive integer. If `a` and `b` are matrices of the same size, then the function treats each element independently. All entries of `b`, `a`, and `quot` are the exponential formats of elements of GF(p^m) relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats.

In all cases, an attempt to divide by the zero element of the field results in a "quotient" of `NaN`.

Examples

The code below displays lists of multiplicative inverses in GF(5) and GF(25). It uses column vectors as inputs to `gfdiv`.

```
% Find inverses of nonzero elements of GF(5).
p = 5;
b = ones(p-1,1);
a = [1:p-1]';
quot1 = gfdiv(b,a,p);
disp('Inverses in GF(5):')
disp('element  inverse')
disp([a, quot1])
```

```
% Find inverses of nonzero elements of GF(25).
m = 2;
field = gftuple([-1:p^m-2]',m,p);
b = zeros(p^m-1,1); % Numerator is zero since 1 = alpha^0.
a = [0:p^m-2]';
quot2 = gfdiv(b,a,field);
disp('Inverses in GF(25), expressed in EXPONENTIAL FORMAT with')
disp('respect to a root of the default primitive polynomial:')
disp('element  inverse')
disp([a, quot2])
```

See Also

gfmul, gfdeconv, gfconv, gftuple

Purpose

Filter data using polynomials over a prime Galois field

Syntax

```
y = gffilter(b,a,x,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `filter` function with Galois arrays. For details, see "Filtering" on page 2-114.

`y = gffilter(b,a,x,p)` filters the data `x` using the filter described by vectors `a` and `b`. `y` is the filtered data in GF(`p`). `p` is a prime number, and all entries of `a` and `b` are between 0 and `p`-1.

By definition of the filter, `y` solves the difference equation

```
a(1)y(n) = b(1)x(n)+b(2)x(n-1)+b(3)x(n-2)+...+b(B+1)x(n-B)
                      -a(2)y(n-1)-a(3)y(n-2)-...-a(A+1)y(n-A)
```

where

- `A`+1 is the length of the vector `a`
- `B`+1 is the length of the vector `b`
- `n` varies between 1 and the length of the vector `x`.

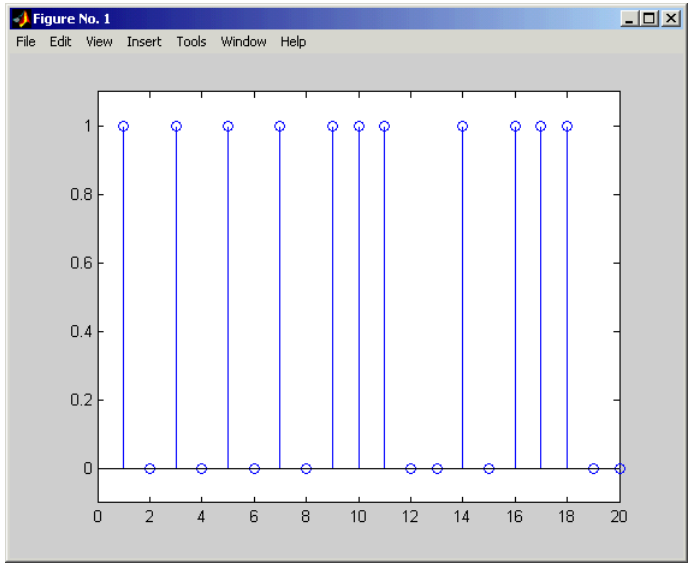
The vector `a` represents the degree-n_a polynomial

```
a(1)+a(2)x+a(3)x^2+...+a(A+1)x^A
```

Examples

The impulse response of a particular filter is given in the code and diagram below.

```
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])
```



See Also

`gfconv`, `gfadd`, `filter`

Purpose

Find a particular solution of `Ax = b` over a prime Galois field

Syntax

```
x = gflineq(A,b,p);
[x,vld] = gflineq(A,b,p);
```

Description

Note This function performs computations in GF(p) where p is odd. To work in GF(2^m), apply the `\` or `/` operator to Galois arrays. For details, see "Solving Linear Equations" on page 2-112.

`x = gflineq(A,b,p)` returns a particular solution of the linear equation `A x = b` over GF(`p`), where `p` is a prime number. If `A` is a k-by-n matrix and `b` is a vector of length k, then `x` is a vector of length n. Each entry of `A`, `x`, and `b` is an integer between 0 and `p-1`. If no solution exists, then `x` is empty.

`[x,vld] = gflineq(...)` returns a flag `vld` that indicates the existence of a solution. If `vld = 1`, then the solution `x` exists and is valid; if `vld = 0`, then no solution exists.

Examples

The code below produces some valid solutions of a linear equation over GF(3).

```
A = [2 0 1;
     1 1 0;
     1 1 2];
% An example in which the solutions are valid
[x,vld] = gflineq(A,[1;0;0],3)

x =

     2
     1
     0


vld =

     1
```

By contrast, the command below finds that the linear equation has *no* solutions.

```
[x2,vld2] = gflineq(zeros(3,3),[2;0;0],3)
This linear equation has no solution.

x2 =

     []


vld2 =

     0
```

Algorithm

`gflineq` uses Gaussian elimination.

See Also

`gfadd`, `gfdiv`, `gfroots`, `gfrank`, `gfconv`, `conv`

Purpose

Find the minimal polynomial of an element of a Galois field

Syntax

```
pol = gfminpol(k,m,p);
pol = gfminpol(k,prim_poly,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `minpol` function with Galois arrays. For details, see "Minimal Polynomials" on page 2-120.

`pol = gfminpol(k,m,p)` finds the minimal polynomial of α^k over GF(`p`), where `p` is a prime number, `m` is an integer greater than 1, and α is a root of the default primitive polynomial for GF(`p^m`). The format of the output is as follows:

- If `k` is a nonnegative integer, then `pol` is a row vector that gives the coefficients of the minimal polynomial in order of ascending powers.
- If `k` is a vector of length *len* all of whose entries are nonnegative integers, then `pol` is a matrix having *len* rows; the rth row of `pol` gives the coefficients of the minimal polynomial of $\alpha^{k(r)}$ in order of ascending powers.

`pol = gfminpol(k,prim_poly,p)` is the same as the first syntax listed, except that α is a root of the primitive polynomial for GF(p^m) specified by `prim_poly`. `prim_poly` is a row vector that gives the coefficients of the degree-m primitive polynomial in order of ascending powers.

Examples

The syntax `gfminpol(k,m,p)` is used in the sample code in "Characterization of Polynomials" on page A-16.

See Also

`gfprimdf`, `gfcosets`, `gfroots`

gfmul

Purpose

Multiply elements of a Galois field

Syntax

```
c = gfmul(a,b,p);
c = gfmul(a,b,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the `.*` operator to Galois arrays. For details, see "Example: Multiplication" on page 2-104.

The `gfmul` function multiplies elements of a Galois field. (To multiply polynomials over a Galois field, use `gfconv` instead.)

`c = gfmul(a,b,p)` multiplies `a` and `b` in GF(`p`). Each entry of `a` and `b` is between 0 and `p`-1. `p` is a prime number. If `a` and `b` are matrices of the same size, then the function treats each element independently.

`c = gfmul(a,b,field)` multiplies `a` and `b` in GF(p^m), where p is a prime number and m is a positive integer. `a` and `b` represent elements of GF(p^m) in exponential format relative to some primitive element of GF(p^m). `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the product, relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

"Arithmetic in Galois Fields" on page A-12 contains examples. Also, the code below shows that $A^2 \cdot A^4 = A^6$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
a = gfmul(2,4,field)

a =

     6
```

See Also gfdiv, gfdeconv, gfadd, gfsub, gftuple

gfpretty

Purpose

Display a polynomial in traditional format

Syntax

```
gfpretty(a)
gfpretty(a,st)
gfpretty(a,st,n)
```

Description

`gfpretty(a)` displays a polynomial in a traditional format, using `X` as the variable and the entries of the row vector `a` as the coefficients in order of ascending powers. The polynomial is displayed in order of ascending powers. Terms having a zero coefficient are not displayed.

`gfpretty(a,st)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of `X`.

`gfpretty(a,st,n)` is the same as the first syntax listed, except that the content of the string `st` is used as the variable instead of `X`, and each line of the display has width `n` instead of the default value of 79.

Note For all syntaxes: If you do not use a fixed-width font, then the spacing in the display might not look correct.

Examples

The code below displays statements about the elements of GF(81).

```
p = 3; m = 4;
ii = randint(1,1,[1,p^m-2]); % Random exponent for prim element
primpolys = gfprimfd(m,'all',p);
[rows, cols] = size(primpolys);
jj = randint(1,1,[1,rows]); % Random primitive polynomial

disp('If A is a root of the primitive polynomial')
gfpretty(primpolys(jj,:)) % Polynomial in X
disp('then the element')
gfpretty([zeros(1,ii),1],'A') % The polynomial A^ii
disp('can also be expressed as')
gfpretty(gftuple(ii,m,p),'A') % Polynomial in A
```

Below is a sample of the output.

If A is a root of the primitive polynomial

$$2 + 2X^3 + X^4$$

then the element

$$A^{22}$$

can also be expressed as

$$2 + A^2 + A^3$$

See Also

gftuple, gfprimdf

gfprimck

Purpose

Check whether a polynomial over a Galois field is primitive

Syntax

```
ck = gfprimck(a,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `isprimitive` function. For details, see "Finding Primitive Polynomials" on page 2-100.

`ck = gfprimck(a,p)` returns a flag `ck` that indicates whether a polynomial over GF(`p`) is irreducible or primitive. `a` is a row vector that gives the coefficients of the polynomial in order of ascending powers. Each coefficient is between 0 and `p`-1. If m is the degree of the polynomial, then the output `ck` is

- -1 if `a` is not an irreducible polynomial
- 0 if `a` is irreducible but not a primitive polynomial for GF(p^m)
- 1 if `a` is a primitive polynomial for GF(p^m)

This function considers the zero polynomial to be "not irreducible" and considers all polynomials of degree zero or one to be primitive.

Examples

"Characterization of Polynomials" on page A-16 contains examples.

Algorithm

An irreducible polynomial over GF(p) of degree at least 2 is primitive if and only if it does not divide $-1 + x^k$ for any positive integer k smaller than p^m-1.

See Also

`gfprimfd`, `gfprimdf`, `gftuple`, `gfminpol`, `gfadd`

References

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

Purpose

Provide default primitive polynomials for a Galois field

Syntax

```
pol = gfprimdf(m,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials" on page 2-100.

`pol = gfprimdf(m,p)` returns the row vector that gives the coefficients, in order of ascending powers, of the default primitive polynomial for GF(`p^m`). `m` is a positive integer and `p` is a prime number.

Examples

The command below shows that $2 + x + x^2$ is the default primitive polynomial for GF(5^2).

```
pol = gfprimdf(2,5)

pol =

     2     1     1
```

The code below displays the default primitive polynomial for each of the fields GF(3^m), where m ranges between 3 and 5.

```
for m = 3:5
   gfpretty(gfprimdf(m,3))
end
```

```
                                        3
                              1 + 2 X + X

                                      4
                               2 + X + X

                                        5
                              1 + 2 X + X
```

See Also

`gfprimck`, `gfprimfd`, `gftuple`, `gfminpol`

gfprimfd

Purpose

Find primitive polynomials for a Galois field

Syntax

```
pol = gfprimfd(m,opt,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `primpoly` function. For details, see "Finding Primitive Polynomials" on page 2-100.

- If `m` = 1, then `pol` = [1 1].
- A polynomial is represented as a row containing the coefficients in order of ascending powers.

`pol = gfprimfd(m,opt,p)` searches for one or more primitive polynomials for GF(`p^m`), where `p` is a prime number and `m` is a positive integer. If `m` = 1, then `pol` = [1 1]. If `m` > 1, then the output `pol` depends on the argument *`opt`* as shown in the table below. Each polynomial is represented in `pol` as a row containing the coefficients in order of ascending powers.

opt	Significance of pol	Format of pol
'**min**'	One primitive polynomial for GF(`p^m`) having the smallest possible number of nonzero terms	The row vector representing the polynomial
'**max**'	One primitive polynomial for GF(`p^m`) having the greatest possible number of nonzero terms	The row vector representing the polynomial
'**all**'	All primitive polynomials for GF(`p^m`)	A matrix, each row of which represents one such polynomial
A positive integer	All primitive polynomials for GF(`p^m`) that have *`opt`* nonzero terms	A matrix, each row of which represents one such polynomial

Examples

The code below seeks primitive polynomials for GF(81) having various other properties. Notice that `fourterms` is empty because no primitive polynomial for GF(81) has exactly four nonzero terms. Also notice that `fewterms` represents a

single polynomial having three terms, while `threeterms` represents *all* of the three-term primitive polynomials for GF(81).

```
p = 3; m = 4; % Work in GF(81).
fewterms = gfprimfd(m,'min',p)
threeterms = gfprimfd(m,3,p)
fourterms = gfprimfd(m,4,p)
```

The output is below.

```
fewterms =

     2     1     0     0     1


threeterms =

     2     1     0     0     1
     2     2     0     0     1
     2     0     0     1     1
     2     0     0     2     1

No primitive polynomial satisfies the given constraints.

fourterms =

     []
```

Algorithm

`gfprimfd` tests for primitivity using `gfprimck`. If *`opt`* is '**`min`**', '**`max`**', or omitted, then polynomials are constructed by converting decimal integers to base `p`. Based on the decimal ordering, `gfprimfd` returns the first polynomial it finds that satisfies the appropriate conditions.

See Also

`gfprimck`, `gfprimdf`, `gftuple`, `gfminpol`

gfrank

Purpose

Compute the rank of a matrix over a Galois field

Syntax

```
rk = gfrank(A,p);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `rank` function with Galois arrays. For details, see "Computing Ranks" on page 2-112.

`rk = gfrank(A,p)` calculates the rank of the matrix `A` in GF(`p`), where `p` is a prime number.

Algorithm

`gfrank` uses an algorithm similar to Gaussian elimination.

Examples

In the code below, `gfrank` says that the matrix `A` has less than full rank. This conclusion makes sense because the determinant of `A` is zero mod `p`.

```
A = [1 0 1;
   2 1 0;
   0 1 1];
p = 3;
det_a = det(A); % Ordinary determinant of A
detmodp = rem(det(A),p); % Determinant mod p
rankp = gfrank(A,p);
disp(['Determinant = ',num2str(det_a)])
disp(['Determinant mod p is ',num2str(detmodp)])
disp(['Rank over GF(p) is ',num2str(rankp)])
```

The output is below.

```
Determinant = 3
Determinant mod p is 0
Rank over GF(p) is 2
```

Purpose

Convert one binary polynomial representation to another

Syntax

```
polystandard = gfrepcov(poly2)
```

Description

Two logical ways to represent polynomials over GF(2) are listed below:

1 `[A_0 A_1 A_2 ... A_(m-1)]` represents the polynomial

$A\_0 + A\_1x + A\_2x^2 + \ldots + A\_(m\text{-}1)x^{m-1}$.

Each entry `A_k` is either one or zero.

2 `[A_0 A_1 A_2 ... A_(m-1)]` represents the polynomial

$x^{A\_0} + x^{A\_1} + x^{A\_2} + \ldots + x^{A\_(m\text{-}1)}$.

Each entry `A_k` is a nonnegative integer. All entries must be distinct.

Format **1** is the standard form used by the Galois field functions in this toolbox, but there are some cases in which format **2** is more convenient.

`polystandard = gfrepcov(poly2)` converts from the second format to the first, for polynomials of degree *at least* 2. `poly2` and `polystandard` are row vectors. The entries of `poly2` are distinct integers, and at least one entry must exceed 1. Each entry of `polystandard` is either 0 or 1.

Note If `poly2` is a *binary* row vector, then `gfrepcov` assumes that it is already in Format 1 above and returns it unaltered.

Examples

The command below converts the representation format of the polynomial $1 + x^2 + x^5$.

```
polystandard = gfrepcov([0 2 5])

polystandard =

     1     0     1     0     0     1
```

See Also

`gfpretty`

gfroots

Purpose

Find the roots of a polynomial over a prime Galois field

Syntax

```
rt = gfroots(f,m,p);
rt = gfroots(f,prim_poly,p);
[rt,rt_tuple] = gfroots(...);
[rt,rt_tuple,field] = gfroots(...);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), use the `roots` function with Galois arrays. For details, see "Roots of Polynomials" on page 2-119.

For all syntaxes, `f` is a row vector that gives the coefficients, in order of ascending powers, of a degree-d polynomial.

Note `gfroots` lists each root exactly once, ignoring multiplicities of roots.

`rt = gfroots(f,m,p)` finds roots in GF(`p^m`) of the polynomial that `f` represents. `m` is an integer greater than or equal to d. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the default primitive polynomial for GF(`p^m`).

`rt = gfroots(f,prim_poly,p)` finds roots in GF(p^m) of the polynomial that `f` represents. `rt` is a column vector each of whose entries is the exponential format of a root. The exponential format is relative to a root of the degree-m primitive polynomial for GF(p^m) that `prim_poly` represents. m is an integer greater than or equal to d.

`[rt,rt_tuple] = gfroots(...)` returns an additional matrix `rt_tuple`, whose kth row is the polynomial format of the root `rt(k)`. The polynomial and exponential formats are both relative to the same primitive element.

`[rt,rt_tuple,field] = gfroots(...)` returns additional matrices `rt_tuple` and `field`. `rt_tuple` is described in the paragraph above. `field` gives the list of elements of the extension field. The list of elements, the polynomial format, and the exponential format are all relative to the same primitive element.

Note For a description of the various formats that `gfroots` uses, see "Representing Elements of Galois Fields" on page A-3.

Examples

"Roots of Polynomials" on page A-17 contains a description and example of the use of `gfroots`.

As another example, the code below finds the polynomial format of the roots of the primitive polynomial $2 + x^3 + x^4$ for GF(81). It then displays the roots in traditional form as polynomials in `alpha`. (The output is omitted here.) Because `prim_poly` is both the primitive polynomial and the polynomial whose roots are sought, `alpha` itself is a root.

```
p = 3; m = 4;
prim_poly = [2 0 0 1 1]; % A primitive polynomial for GF(81)
f = prim_poly; % Find roots of the primitive polynomial.
[rt,rt_tuple] = gfroots(f,prim_poly,p);
% Display roots as polynomials in alpha.
for ii = 1:length(rt_tuple)
   gfpretty(rt_tuple(ii,:),'alpha')
end
```

See Also

`gfprimdf`

gfsub

Purpose

Subtract polynomials over a Galois field

Syntax

```
c = gfsub(a,b,p);
c = gfsub(a,b,p,len);
c = gfsub(a,b,field);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To work in GF(2^m), apply the - operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction" on page 2-103.

`c = gfsub(a,b,p)` calculates `a` minus `b`, where `a` and `b` represent polynomials over GF(`p`) and `p` is a prime number. `a`, `b`, and `c` are row vectors that give the coefficients of the corresponding polynomials in order of ascending powers. Each coefficient is between 0 and `p`-1. If `a` and `b` are matrices of the same size, then the function treats each row independently.

`c = gfsub(a,b,p,len)` subtracts row vectors as in the syntax above, except that it returns a row vector of length `len`. The output `c` is a truncated or extended representation of the answer. If the row vector corresponding to the answer has fewer than `len` entries (including zeros), then extra zeros are added at the end; if it has more than `len` entries, then entries from the end are removed.

`c = gfsub(a,b,field)` calculates `a` minus `b`, where `a` and `b` are the exponential format of two elements of GF(p^m), relative to some primitive element of GF(p^m). p is a prime number and m is a positive integer. `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. `c` is the exponential format of the answer, relative to the same primitive element. See "Representing Elements of Galois Fields" on page A-3 for an explanation of these formats. If `a` and `b` are matrices of the same size, then the function treats each element independently.

Examples

In the code below, `differ` is the difference of $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5), and `linpart` is the degree-one part of `differ`.

```
differ = gfsub([2 3 1],[4 2 3],5)
```

```
differ =

     3     1     3

linpart = gfsub([2 3 1],[4 2 3],5,2)

linpart =

     3     1
```

The code below shows that $A^2 - A^4 = A^7$, where A is a root of the primitive polynomial $2 + 2x + x^2$ for GF(9).

```
p = 3; m = 2;
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
d = gfsub(2,4,field)

d =

     7
```

See Also

gfadd, gfconv, gfmul, gfdeconv, gfdiv, gftuple

gftable

Purpose

Generate a file to accelerate Galois field computations

Syntax

```
gftable(m,prim_poly);
```

Description

`gftable(m,prim_poly)` generates a file that can help accelerate computations in the field GF(`2^m`) as described by the *nondefault* primitive polynomial `prim_poly`. The integer `m` is between 1 and 16. The integer `prim_poly` represents a primitive polynomial for GF(`2^m`) using the format described in "Specifying the Primitive Polynomial" on page 2-99. The function places the file, called `userGftable.mat`, in your current working directory. If necessary, the function overwrites any writable existing version of the file.

Note If `prim_poly` is the default primitive polynomial for GF(`2^m`) listed in the table on the `gf` reference page, then this function has no effect. A MAT-file in your MATLAB installation already includes information that facilitates computations with respect to the default primitive polynomial.

Examples

In the example below, you would expect `t3` to be similar to `t1` and significantly smaller than `t2`, assuming that you do not already have a `userGftable.mat` file that includes the (`m`, `prim_poly`) pair (`8`, `501`).

```
% Sample code to check how much gftable improves speed.
tic; a = gf(repmat([0:2^8-1],1000,1),8); b = a.^100; t1 = toc;
tic; a = gf(repmat([0:2^8-1],1000,1),8,501); b = a.^100; t2 = toc;
gftable(8,501); % Include this primitive polynomial in the file.
tic; a = gf(repmat([0:2^8-1],1000,1),8,501); b = a.^100; t3 = toc;
```

See Also

`gf`, "Speed and Nondefault Primitive Polynomials" on page 2-123

Purpose

Minimize the length of a polynomial representation

Syntax

```
c = gftrunc(a);
```

Description

`c = gftrunc(a)` truncates a row vector, `a`, that gives the coefficients of a GF(p) polynomial in order of ascending powers. If a(k) = 0 whenever k > d + 1, then the polynomial has degree d. The row vector `c` omits these high-order zeros and thus has length d + 1.

Examples

In the code below, zeros are removed from the end, but *not* from the beginning or middle, of the row-vector representation of $x^2 + 2x^3 + 3x^4 + 4x^7 + 5x^8$.

```
c = gftrunc([0 0 1 2 3 0 0 4 5 0 0])

c =

     0     0     1     2     3     0     0     4     5
```

See Also

`gfadd`, `gfsub`, `gfconv`, `gfdeconv`, `gftuple`

gftuple

Purpose

Simplify or convert the format of elements of a Galois field

Syntax

```
tp = gftuple(a,m,p);
tp = gftuple(a,prim_poly,p);
tp = gftuple(a,prim_poly,p,prim_ck);
[tp,expform] = gftuple(...);
```

Description

Note This function performs computations in GF(p^m) where p is odd. To perform equivalent computations in GF(2^m), apply the `.^` operator and the `log` function to Galois arrays. For more information, see "Example: Exponentiation" on page 2-106 and "Example: Elementwise Logarithm" on page 2-106, respectively.

For All Syntaxes

`gftuple` serves to simplify the polynomial or exponential format of Galois field elements, or to convert from one format to another. For an explanation of the formats that `gftuple` uses, see "Representing Elements of Galois Fields" on page A-3.

In this discussion, the format of an element of GF(p^m) is called "simplest" if all exponents of the primitive element are

- Between 0 and m-1 for the polynomial format
- Either `-Inf`, or between 0 and p^{m-2}, for the exponential format

For all syntaxes, `a` is a matrix, each row of which represents an element of a Galois field. The format of `a` determines how MATLAB interprets it:

- If `a` is a column of integers, then MATLAB interprets each row as an *exponential* format of an element. Negative integers are equivalent to `-Inf` in that they all represent the zero element of the field.
- If `a` has more than one column, then MATLAB interprets each row as a *polynomial* format of an element. (Each entry of `a` must be an integer between 0 and `p`-1.)

The exponential or polynomial formats mentioned above are all relative to a primitive element specified by the *second* input argument. The second argument is described below.

For Specific Syntaxes

`tp = gftuple(a,m,p)` returns the simplest polynomial format of the elements that `a` represents, where the kth row of `tp` corresponds to the kth row of `a`. The formats are relative to a root of the default primitive polynomial for GF(`p^m`). `m` is a positive integer and `p` is a prime number. If possible, the default primitive polynomial is used to simplify the polynomial formats.

`tp = gftuple(a,prim_poly,p)` returns the simplest polynomial format of the element that `a` represents, where the kth row of `tp` corresponds to the kth row of `a`. `p` is a prime number. The formats are relative to a root of the primitive polynomial whose coefficients are given, in order of ascending powers, by the row vector `prim_poly`. If possible, the function uses this primitive polynomial to simplify the polynomial formats.

`tp = gftuple(a,prim_poly,p,prim_ck)` is the same as `tp = gftuple(a,prim_poly,p)` except that `gftuple` checks whether `prim_poly` represents a polynomial that is indeed primitive. If not, then `gftuple` generates an error and `tp` is not returned. The input argument `prim_ck` can be any number or string; only its existence matters.

`[tp,expform] = gftuple(...)` returns the additional matrix `expform`. The kth row of `expform` is the simplest exponential format of the element that the kth row of `a` represents. All other features are as described in earlier parts of this "Description" section, depending on the input arguments.

Examples

Some examples are in these subsections of "Appendix: Galois Fields of Odd Characteristic":

- "List of All Elements of a Galois Field" on page A-4 (end of section)
- "Converting to Simplest Polynomial Format" on page A-8
- "Converting to Simplest Exponential Format" on page A-10

As another example, the `gftuple` command below generates a list of elements of GF(`p^m`), arranged relative to a root of the default primitive polynomial. Some functions in this toolbox use such a list as an input argument.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field = gftuple([-1:p^m-2]',m,p);
```

Finally, the two commands below illustrate the influence of the *shape* of the input matrix. In the first command, a column vector is treated as a sequence of elements expressed in exponential format. In the second command, a row vector is treated as a single element expressed in polynomial format.

```
tp1 = gftuple([0; 1],3,3)

tp1 =

     1     0     0
     0     1     0

tp2 = gftuple([0, 0, 0, 1],3,3)

tp2 =


     2     1     0
```

The outputs reflect that, according to the default primitive polynomial for GF(3^3), the relations below are true.

$$\alpha^0 = 1 + 0\alpha + 0\alpha^2$$

$$\alpha^1 = 0 + 1\alpha + 0\alpha^2$$

$$0 + 0\alpha + 0\alpha^2 + \alpha^3 = 2 + \alpha + 0\alpha^2$$

Algorithm

`gftuple` uses recursive callbacks to determine the exponential format.

See Also

`gfadd`, `gfmul`, `gfconv`, `gfdiv`, `gfdeconv`, `gfprimdf`

Purpose

Calculate the minimum distance of a linear block code

Syntax

```
wt = gfweight(genmat);
wt = gfweight(genmat,'gen');
wt = gfweight(parmat,'par');
wt = gfweight(genpoly,n);
```

Description

The minimum distance, or minimum weight, of a linear block code is defined as the smallest positive number of nonzero entries in any n-tuple that is a codeword.

`wt = gfweight(genmat)` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(genmat,'gen')` returns the minimum distance of the linear block code whose generator matrix is `genmat`.

`wt = gfweight(parmat,'par')` returns the minimum distance of the linear block code whose parity-check matrix is `parmat`.

`wt = gfweight(genpoly,n)` returns the minimum distance of the *cyclic* code whose codeword length is `n` and whose generator polynomial is represented by `genpoly`. `genpoly` is a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples

The commands below illustrate three different ways to compute the minimum distance of a (7,4) cyclic code.

```
n = 7;
% Generator polynomial of (7,4) cyclic code
genpoly = cyclpoly(n,4);
[parmat, genmat] = cyclgen(n,genpoly);
wts = [gfweight(genmat,'gen'),gfweight(parmat,'par'),...
    gfweight(genpoly,n)]

wts =

     3     3     3
```

See Also

`hammgen`, `cyclpoly`, `bchpoly`

hammgen

Purpose

Produce parity-check and generator matrices for Hamming code

Syntax

```
h = hammgen(m);
h = hammgen(m,pol);
[h,g] = hammgen(...);
[h,g,n,k] = hammgen(...);
```

Description

For all syntaxes, the codeword length is `n`. `n` has the form 2^m-1 for some positive integer m greater than or equal to 3. The message length, `k`, has the form `n`-`m`.

`h = hammgen(m)` produces an `m`-by-`n` parity-check matrix for a Hamming code having codeword length `n = 2^m-1`. The input `m` is a positive integer greater than or equal to 3. The message length of the code is `n`-`m`. The binary primitive polynomial used to produce the Hamming code is the default primitive polynomial for GF(`2^m`), represented by `gfprimdf(m)`.

`h = hammgen(m,pol)` produces an `m`-by-`n` parity-check matrix for a Hamming code having codeword length `n = 2^m-1`. The input `m` is a positive integer greater than or equal to 3. The message length of the code is `n`-`m`. `pol` is a row vector that gives the coefficients, in order of ascending powers, of the binary primitive polynomial for GF(`2^m`) that is used to produce the Hamming code. `hammgen` produces an error if `pol` represents a polynomial that is not, in fact, primitive.

`[h,g] = hammgen(...)` is the same as `h = hammgen(...)` except that it also produces the `k`-by-`n` generator matrix `g` that corresponds to the parity-check matrix `h`. `k`, the message length, equals `n-m`, or `2^m-1-m`.

`[h,g,n,k] = hammgen(...)` is the same as `[h,g] = hammgen(...)` except that it also returns the codeword length `n` and the message length `k`.

Note If your value of `m` is less than 25 and if your primitive polynomial is the default primitive polynomial for GF(`2^m`), then the syntax `hammgen(m)` is likely to be faster than the syntax `hammgen(m,pol)`.

Examples

The command below exhibits the parity-check and generator matrices for a Hamming code with codeword length $7 = 2^3-1$ and message length $4 = 7-3$.

```
[h,g,n,k] = 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


n =

     7


k =

     4
```

The command below, which uses $1 + x^2 + x^3$ as the primitive polynomial for $GF(2^3)$, shows that the parity-check matrix depends on the choice of primitive polynomial. Notice that `h1` below is different from `h` in the example above.

```
h1 = hammgen(3,[1 0 1 1])

h1 =

     1     0     0     1     1     1     0
     0     1     0     0     1     1     1
     0     0     1     1     1     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`, `gfprimfd`, `gfprimdf`

Purpose

Convert a Hankel matrix to a linear system model

Syntax

```
[num,den] = hank2sys(h,ini,tol)
[num,den,sv] = hank2sys(h,ini,tol)
[a,b,c,d] = hank2sys(h,ini,tol)
[a,b,c,d,sv] = hank2sys(h,ini,tol)
```

Description

`[num,den] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a linear system transfer function with numerator `num` and denominator `den`. The vectors `num` and `den` list the coefficients of their respective polynomials in ascending order of powers of z^{-1}. The argument `ini` is the system impulse at time zero. If `tol` > 1, then `tol` is the order of the conversion. If `tol` < 1, then `tol` is the tolerance in selecting the conversion order based on the singular values. If you omit `tol`, then its default value is 0.01. This conversion uses the singular value decomposition method.

`[num,den,sv] = hank2sys(h,ini,tol)` returns a vector `sv` that lists the singular values of `h`.

`[a,b,c,d] = hank2sys(h,ini,tol)` converts a Hankel matrix `h` to a corresponding linear system state-space model. `a`, `b`, `c`, and `d` are matrices. The input parameters are the same as in the first syntax above.

`[a,b,c,d,sv] = hank2sys(h,ini,tol)` is the same as the syntax above, except that `sv` is a vector that lists the singular values of `h`.

Examples

```
h = hankel([1 0 1]);
[num,den,sv] = hank2sys(h,0,.01)

num =

         0    1.0000    0.0000    1.0000


den =

    1.0000    0.0000    0.0000    0.0000
```

```
sv =

    1.6180
    1.0000
    0.6180
```

See Also hilbiir, hankel, rcosflt

Purpose

Design a Hilbert transform IIR filter

Syntax

```
hilbiir;
hilbiir(ts);
hilbiir(ts,dly);
hilbiir(ts,dly,bandwidth);
hilbiir(ts,dly,bandwidth,tol);
[num,den] = hilbiir(...);
[num,den,sv] = hilbiir(...);
[a,b,c,d] = hilbiir(...);
[a,b,c,d,sv] = hilbiir(...);
```

Description

The function `hilbiir` designs a Hilbert transform filter. The output is either

- A plot of the filter's impulse response, or
- A quantitative characterization of the filter, using either a transfer function model or a state-space model

Background Information

An ideal Hilbert transform filter has the transfer function H(s) = -j sgn(s), where sgn(.) is the signum function (`sign` in MATLAB). The impulse response of the Hilbert transform filter is

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

Because the Hilbert transform filter is a noncausal filter, the `hilbiir` function introduces a group delay, `dly`. A Hilbert transform filter with this delay has the impulse response

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

Choosing a Group Delay Parameter

The filter design is an approximation. If you provide the filter's group delay as an input argument, then these two suggestions can help improve the accuracy of the results:

- Choose the sample time `ts` and the filter's group delay `dly` so that `dly` is at least a few times larger than `ts` and `rem(dly,ts) = ts/2`. For example, you can set `ts` to `2*dly/N`, where `N` is a positive integer.
- At the point `t = dly`, the impulse response of the Hilbert transform filter can be interpreted as `0`, `-Inf`, or `Inf`. If `hilbiir` encounters this point, then it sets the impulse response there to zero. To improve accuracy, avoid the point `t = dly`.

Syntaxes for Plots

Each of these syntaxes produces a plot of the impulse response of the filter that the `hilbiir` function designs, as well as the impulse response of a corresponding ideal Hilbert transform filter.

`hilbiir` plots the impulse response of a fourth-order digital Hilbert transform filter with a 1-second group delay. The sample time is 2/7 seconds. In this particular design, the tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter with a 1-second group delay.

`hilbiir(ts)` plots the impulse response of a fourth-order Hilbert transform filter with a sample time of `ts` seconds and a group delay of `ts*7/2` seconds. The tolerance index is 0.05. The plot also displays the impulse response of the ideal Hilbert transform filter having a sample time of `ts` seconds and a group delay of `ts*7/2` seconds.

`hilbiir(ts,dly)` is the same as the syntax above, except that the filter's group delay is `dly` for both the ideal filter and the filter that `hilbiir` designs. See "Choosing a Group Delay Parameter" above for guidelines on choosing `dly`.

`hilbiir(ts,dly,bandwidth)` is the same as the syntax above, except that `bandwidth` specifies the assumed bandwidth of the input signal and that the filter design might use a compensator for the input signal. If `bandwidth` = 0 or `bandwidth` > 1/(2*`ts`), then `hilbiir` does not use a compensator.

`hilbiir(ts,dly,bandwidth,tol)` is the same as the syntax above, except that `tol` is the tolerance index. If `tol` < 1, then the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < \texttt{tol}$$

If `tol` > 1, then the order of the filter is `tol`.

Syntaxes for Transfer Function and State-Space Quantities

Each of these syntaxes produces quantitative information about the filter that `hilbiir` designs, but does *not* produce a plot. The input arguments for these syntaxes (if you provide any) are the same as those described in the "Syntaxes for Plots" section above.

`[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 that `hilbiir` uses in the computation.

`[a,b,c,d] = hilbiir(...)` outputs the discrete-time state-space model of the designed Hilbert transform filter. `a`, `b`, `c`, and `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 that `hilbiir` uses in the computation.

Algorithm

The `hilbiir` function calculates the impulse response of the ideal Hilbert transform filter response with a group delay. It fits the response curve using a singular-value decomposition method. See the book by Kailath listed below.

Examples

At the MATLAB prompt, type `hilbiir` or `[num,den] = hilbiir` for an example using the function's default values.

See Also

`grpdelay`

References

Kailath, Thomas, *Linear Systems*, Englewood Cliffs, N.J., Prentice-Hall, 1980.

ifft

Purpose

Inverse discrete Fourier transform

Syntax

```
ifft(x)
```

Description

`ifft(x)` is the inverse discrete Fourier transform (DFT) of the Galois vector `x`. If `x` is in the Galois field GF(2^m), then the length of `x` must be 2^m-1.

Examples

For an example using `ifft`, see the reference page for `fft`.

Limitations

The Galois field over which this function works must have 256 or fewer elements. In other words, `x` must be in the Galois field GF(2^m), where m is an integer between 1 and 8.

Algorithm

If `x` is a column vector, then `ifft` applies `dftmtx` to the multiplicative inverse of the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`fft`, `dftmtx`

Purpose

True for a primitive polynomial for a Galois field

Syntax

```
isprimitive(a)
```

Description

`isprimitive(a)` returns `1` if the polynomial that `a` represents is primitive for the Galois field GF(2^m), and `0` otherwise. The input `a` can represent the polynomial using one of these formats:

- A nonnegative integer less than 2^{17}. The binary representation of this integer indicates the coefficients of the polynomial. In this case, m is `floor(log2(a))`.
- A Galois row vector in GF(2), listing the coefficients of the polynomial in order of descending powers. In this case, m is the order of the polynomial represented by `a`.

Examples

```
a = primpoly(3,'all','nodisplay'); % All primitive polys for GF(8)

a =

    11
    13

isp1 = isprimitive(13) % 13 represents a primitive polynomial.

isp1 =

     1

isp2 = isprimitive(14) % 14 represents a nonprimitive polynomial.

isp2 =

     0
```

See Also

`primpoly`

istrellis

Purpose

Check if the input is a valid trellis structure

Syntax

```
[isok,status] = istrellis(s);
```

Description

`[isok,status] = istrellis(s)` checks if the input `s` is a valid trellis structure. If the input is a valid trellis structure, then `isok` is 1 and `status` is an empty string. Otherwise, `isok` is 0 and `status` is a string that indicates why `s` is not a valid trellis structure.

A valid trellis structure is a MATLAB structure whose fields are as in the table below.

Fields of a Valid Trellis Structure for a Rate k/n Code

Field in Trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

In the `nextStates` matrix, each entry is an integer between 0 and `numStates`-1. The element in the sth row and uth column denotes the next state when the starting state is s-1 and the input bits have decimal representation u-1. To convert the input bits to a decimal value, use the first input bit as the most significant bit (MSB). For example, the second column of the `nextStates` matrix stores the next states when the current set of input values is {0,...,0,1}.

To convert the state to a decimal value, use this rule: If k exceeds 1, then the shift register that receives the first input stream in the encoder provides the least significant bits in the state number, while the shift register that receives the last input stream in the encoder provides the most significant bits in the state number.

In the `outputs` matrix, the element in the sth row and uth column denotes the encoder's output when the starting state is s-1 and the input bits have decimal representation u-1. To convert to decimal value, use the first output bit as the MSB.

Examples

These commands assemble the fields into a very simple trellis structure, and then verify the validity of the trellis structure.

```
trellis.numInputSymbols = 2;
trellis.numOutputSymbols = 2;
trellis.numStates = 2;
trellis.nextStates = [0 1;0 1];
trellis.outputs = [0 0;1 1];
[isok,status] = istrellis(trellis)

isok =

     1


status =

     ''
```

Another example of a trellis is in "Trellis Description of a Convolutional Encoder" on page 2-50.

See Also

`poly2trellis`, `struct`, `convenc`, `vitdec`

lloyds

Purpose

Optimize quantization parameters using the Lloyd algorithm

Syntax

```
[partition,codebook] = lloyds(training_set,initcodebook);
[partition,codebook] = lloyds(training_set,len);
[partition,codebook] = lloyds(training_set,...,tol);
[partition,codebook,distor] = lloyds(...);
[partition,codebook,distor,reldistor] = lloyds(...);
```

Description

`[partition,codebook] = lloyds(training_set,initcodebook)` optimizes the scalar quantization parameters `partition` and `codebook` for the training data in the vector `training_set`. `initcodebook`, a vector of length at least 2, is the initial guess of the codebook values. The output `codebook` is a vector of the same length as `initcodebook`. The output `partition` is a vector whose length is one less than the length of `codebook`.

See either "Representing Quantization Parameters" on page 2-13 or the reference page for `quantiz` in this chapter, for a description of the formats of `partition` and `codebook`.

Note `lloyds` optimizes for the data in `training_set`. For best results, `training_set` should be similar to the data that you plan to quantize.

`[partition,codebook] = lloyds(training_set,len)` is the same as the first syntax, except that the scalar argument `len` indicates the size of the vector `codebook`. This syntax does not include an initial codebook guess.

`[partition,codebook] = lloyds(training_set,...,tol)` is the same as the two syntaxes above, except that `tol` replaces 10^{-7} in condition **1** of the algorithm description below.

`[partition,codebook,distor] = lloyds(...)` returns the final mean square distortion in the variable `distor`.

`[partition,codebook,distor,reldistor] = lloyds(...)` returns a value `reldistor` that is related to the algorithm's termination. In case **1** of "Algorithm" below, `reldistor` is the relative change in distortion between the last two iterations. In case **2** , `reldistor` is the same as `distor`.

Examples

The code below optimizes the quantization parameters for a sinusoidal transmission via a 3-bit channel. Because the typical data is sinusoidal, `training_set` is a sampled sine wave. Because the channel can transmit 3 bits at a time, `lloyds` prepares a codebook of length 2^3.

```
% Generate a complete period of a sinusoidal signal.
x = sin([0:1000]*pi/500);
[partition,codebook] = lloyds(x,2^3)

partition =

   -0.8540   -0.5973   -0.3017    0.0031    0.3077    0.6023    0.8572


codebook =

  Columns 1 through 7

   -0.9504   -0.7330   -0.4519   -0.1481    0.1558    0.4575    0.7372

  Column 8

    0.9515
```

Algorithm

`lloyds` uses an iterative process to try to minimize the mean square distortion. The optimization processing ends when either

1. The relative change in distortion between iterations is less than 10^{-7}.
2. The distortion is less than `eps*max(training_set)`, where `eps` is the MATLAB floating-point relative accuracy.

See Also

`quantiz`, `dpcmopt`

References

Lloyd, S. P., "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, Vol IT-28, March, 1982, pp. 129-137.

Max, J., "Quantizing for Minimum Distortion," *IRE Transactions on Information Theory*, Vol. IT-6, March, 1960, pp. 7-12.

log

Purpose

Logarithm in a Galois field

Syntax

```
y = log(x)
```

Description

`y = log(x)` computes the logarithm of each element in the Galois array `x`. That is, `y` is an integer array that solves the equation `A.^y = x`, where `A` is the primitive element used to represent elements in `x`. More explicitly, the base `A` of the logarithm is `gf(2,x.m)` or `gf(2,x.m,x.prim_poly)`. All elements in `x` must be nonzero because the logarithm of zero is undefined.

Examples

The code below illustrates how the logarithm operation inverts exponentiation.

```
m = 4; x = gf([8 1 6; 3 5 7; 4 9 2],m);
y = log(x);
primel = gf(2,m); % Primitive element in the field
z = primel .^ y; % This is now the same as x.
ck = isequal(x,z)

ck =

     1
```

The code below shows that the logarithm of `1` is `0` and that the logarithm of the base (`primel`) is `1`.

```
m = 4; primel = gf(2,m);
yy = log([1, primel])

yy =

     0     1
```

Purpose

Generalized Marcum Q function

Syntax

```
Q = marcumq(a,b);
Q = marcumq(a,b,m);
```

Description

`Q = marcumq(a,b)` computes the Marcum Q function of `a` and `b`, defined by

$$Q(a,b) = \int_b^\infty x \exp\left(-\frac{x^2+a^2}{2}\right) I_0(ax)\,dx$$

where `a` and `b` are nonnegative real numbers. In this expression, I_0 is the modified Bessel function of the first kind of zero order.

`Q = marcumq(a,b,m)` computes the generalized Marcum Q, defined by

$$Q_m(a,b) = \frac{1}{a^{m-1}} \int_b^\infty x^m \exp\left(-\frac{x^2+a^2}{2}\right) I_{m-1}(ax)\,dx$$

where `a` and `b` are nonnegative real numbers, and `m` is a nonnegative integer. In this expression, $I_{m\text{-}1}$ is the modified Bessel function of the first kind of order m-1.

See Also

`besseli`; `ncx2cdf` (Statistics Toolbox)

References

Cantrell, P. E., and A. K. Ojha, "Comparison of Generalized Q-Function Algorithms," *IEEE Transactions on Information Theory*, Vol. IT-33, July, 1987, pp. 591-596.

Marcum, J. I., "A Statistical Theory of Target Detection by Pulsed Radar: Mathematical Appendix," RAND Corporation, Santa Monica, CA, Research Memorandum RM-753, July 1, 1948. Reprinted in *IRE Transactions on Information Theory*, Vol. IT-6, April, 1960, pp. 59-267.

McGee, W. F., "Another Recursive Method of Computing the Q Function," *IEEE Transactions on Information Theory*, vol. IT-16, July, 1970, pp. 500-501.

mask2shift

Purpose

Convert mask vector to shift for a shift register configuration

Syntax

```
shift = mask2shift(prpoly,mask)
```

Description

`shift = mask2shift(prpoly,mask)` returns the shift that is equivalent to a mask, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `mask` input is a binary vector whose length is the degree of the primitive polynomial.

Note To save time, `mask2shift` does not check that `prpoly` is primitive. If it is not primitive, then the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

For more information about how masks and shifts are related to pseudonoise sequence generators, see `shift2mask`.

Definition of Equivalent Shift

If A is a root of the primitive polynomial and m(A) is the mask polynomial evaluated at A, then the equivalent shift s solves the equation $A^s = m(A)$. To interpret the vector `mask` as a polynomial, treat `mask` as a list of coefficients in order of descending powers.

Examples

The first command below converts a mask of $x^3 + 1$ into an equivalent shift, for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$. The second command shows that a mask of 1 is equivalent to a shift of 0. In both cases, notice that the length of the `mask` vector is one less than the length of the `prpoly` vector.

```
s = mask2shift([1 1 0 0 1],[1 0 0 1])
```

```
s =

     4

s2 = mask2shift([1 1 0 0 1],[0 0 0 1])

s2 =

     0
```

See Also

`shift2mask`, `log`, `isprimitive`, `primpoly`

References

[1] Lee, J. S., and L. E. Miller, *CDMA Systems Engineering Handbook*, Boston, Artech House, 1998.

[2] Simon, Marvin K., Jim K. Omura, et al., *Spread Spectrum Communications Handbook*, New York, McGraw-Hill, 1994.

minpol

Purpose

Find the minimal polynomial of an element of a Galois field

Syntax

```
pl = minpol(x);
```

Description

`pl = minpol(x)` finds the minimal polynomial of each element in the Galois column vector `x`. The output `pl` is an array in GF(2). The kth row of `pl` lists the coefficients, in order of descending powers, of the minimal polynomial of the kth element of `x`.

Note The output is in GF(2) even if the input is in a different Galois field.

Examples

The code below uses `m = 4` and finds that the minimal polynomial of `gf(2,m)` is just the primitive polynomial used for the field GF(`2^m`). This is true for any value of `m`, not just the value used in the example.

```
m = 4;
A = gf(2,m)
pl = minpol(A)
```

The output is below. Notice that the row vector `[1 0 0 1 1]` represents the polynomial `D^4 + D + 1`.

```
A = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     2


pl = GF(2) array.

Array elements =

     1     0     0     1     1
```

Another example is in “Minimal Polynomials” on page 2-120.

See Also

`cosets`

Purpose

Matrix left division \ of Galois arrays

Syntax

```
x = A \ B;
```

Description

`x = A\B` divides the Galois array `A` into `B` to produce a particular solution of the linear equation `A*x = B`. In the special case when `A` is a nonsingular square matrix, `x` is the unique solution, `inv(A)*B`, to the equation.

Examples

The code below shows that `A \ eye(size(A))` is the inverse of the nonsingular square matrix `A`.

```
m = 4; A = gf([8 1 6; 3 5 7; 4 9 2],m);
Id = gf(eye(size(A)),m);
X = A \ Id;
ck1 = isequal(X*A, Id)
ck2 = isequal(A*X, Id)

ck1 =

     1


ck2 =

     1
```

Other examples are in “Solving Linear Equations” on page 2-112.

Limitations

The matrix `A` must be either

- A nonsingular square matrix
- A nonsquare matrix such that `A'*A` and `A*A'` are nonsingular

Algorithm

If `A` is an M-by-N tall matrix where M > N, then `A \ B` is the same as `(A'*A) \ (A'*B)`.

If `A` is an M-by-N wide matrix where M < N, then `A \ B` is the same as `A' * ((A*A') \ B)`. This solution is not unique.

modmap

Purpose

Map a digital signal to an analog signal

Syntax

```
modmap('method',...);
y = modmap(x,Fd,Fs,'ask',M);
y = modmap(x,Fd,Fs,'fsk',M,tone);
y = modmap(x,Fd,Fs,'msk');
y = modmap(x,Fd,Fs,'psk',M);
y = modmap(x,Fd,Fs,'qask',M);
y = modmap(x,Fd,Fs,'qask/arb',inphase,quadr);
y = modmap(x,Fd,Fs,'qask/cir',numsig,amp,phs);
```

Optional Inputs

Input	Default Value
`tone`	`Fd`
`amp`	`[1:length(numsig)]`
`phs`	`numsig*0`

Description

The digital modulation process consists of two steps: mapping the digital signal to an analog signal and modulating this analog signal. The function `modmap` performs the first step. You can perform the second step using `amod`, `amodce`, or your own custom modulator. The table below lists the digital modulation schemes that `modmap` supports.

Modulation Scheme	Value of 'method'
M-ary amplitude shift keying	'**ask**'
M-ary frequency shift keying	'**fsk**'
Minimum shift keying	'**msk**'
M-ary phase shift keying	'**psk**'
Quadrature amplitude shift keying	'**qask**', '**qask/arb**', or '**qask/cir**'

To Plot a Signal Constellation

`modmap('method',...)` creates a plot that characterizes the `M`-ary modulation method that '*method*' specifies. '*method*' is one of the entries in the

right-hand column of the table above. If '*method*' is a value other than '**fsk**' or '**msk**', then the plot shows the signal constellation; otherwise, it shows the spectrum.

For most methods, the input parameters that follow '*method*' in this syntax are the same as those that follow '*method*' in the corresponding mapping syntax. For more information about them, see "To Map a Digital Signal (Specific Syntax Information)" below.

However, if '*method*' is '**msk**', then the syntax is

```
modmap('msk',Fd)
```

where `Fd` is the sampling rate of the message signal.

To Map a Digital Signal (General Information)

The generic syntax `y = modmap(x,Fd,Fs,...)` maps the digital message signal `x` onto an analog signal. `x` is a matrix of nonnegative integers. The sizes of `x` and `y` depend on the modulation method:

- **(ASK, FSK, MSK methods)** If `x` is a vector of length n, then `y` is a column vector of length `n*Fs/Fd`. Otherwise, if `x` is n-by-m, then `y` is (`n*Fs/Fd`)-by-m and each column of `x` is processed separately.
- **(PSK, QASK methods)** If `x` is a vector of length n, then `y` is an `n*Fs/Fd`-by-2 matrix. Otherwise, if `x` is n-by-m, then `y` is (`n*Fs/Fd`)-by-2m and each column of `x` is processed separately. The odd-numbered columns in `y` represent in-phase components and the even-numbered columns represent quadrature components.

The sampling rates in hertz of `x` and `y`, respectively, are `Fd` and `Fs`. (Thus `1/Fd` represents the time interval between two consecutive samples in `x`, and similarly for `y`.) The ratio `Fs/Fd` must be a positive integer.

To Map a Digital Signal (Specific Syntax Information)

`y = modmap(x,Fd,Fs,'ask',M)` maps to an `M`-ary amplitude shift keying signal constellation. Each entry of `x` must be in the range [0, `M`-1]. Each entry of `y` is in the range [-1, 1].

`y = modmap(x,Fd,Fs,'fsk',M,tone)` maps to frequencies in an `M`-ary frequency shift keying set. Each entry of `x` must be in the range [0, `M`-1]. The

optional argument `tone` is the separation between successive frequencies in the FSK set. The default value of `tone` is `Fd`.

`y = modmap(x,Fd,Fs,'msk')` maps to frequencies in a minimum shift keying set. Each entry of `x` is either 0 or 1. The separation between the two frequencies is `Fd/2`.

`y = modmap(x,Fd,Fs,'psk',M)` maps to an `M`-ary phase shift keying signal constellation. Each entry of `x` must be in the range [0, `M`-1].

`y = modmap(x,Fd,Fs,'qask',M)` maps to an `M`-ary quadrature amplitude shift keying square signal constellation. The table below shows the maximum value of the in-phase and quadrature components in `y`, for several small values of `M`.

M	Maximum of y	M	Maximum of y
2	1	32	5
4	1	64	7
8	3 (quadrature maximum is 1)	128	11
16	3	256	15

Note To see how symbols are mapped to the constellation points, generate a square constellation plot using `qaskenco(M)` or `modmap('qask',M)`.

`y = modmap(x,Fd,Fs,'qask/arb',inphase,quadr)` maps to a quadrature amplitude shift keying signal constellation that you define using the vectors `inphase` and `quadr`. The signal constellation point for the kth message has in-phase component `inphase`(k+1) and quadrature component `quadr`(k+1).

`y = modmap(x,Fd,Fs,'qask/cir',numsig,amp,phs)` maps to a quadrature amplitude shift keying circular signal constellation. `numsig`, `amp`, and `phs` are vectors of the same length. The entries in `numsig` and `amp` must be positive. If k is an integer in the range [1, `length(numsig)`], then `amp`(k) is the radius of

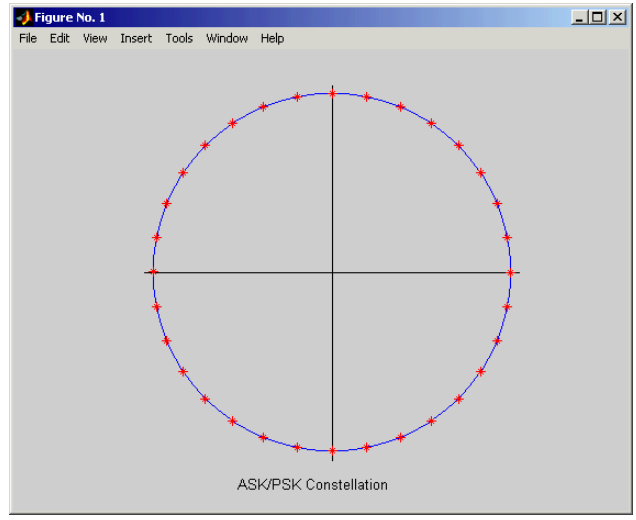
the kth circle, `numsig(k)` is the number of constellation points on the kth circle, and `phs(k)` is the phase of the first constellation point plotted on the kth circle. All points on the kth circle are evenly spaced. If you omit `phs`, then its default value is `numsig*0`. If you omit `amp`, then its default value is `[1:length(numsig)]`.

Note To see how symbols are mapped to the constellation points, generate a labeled circle constellation plot using `apkconst(numsig,amp,phs,'n')`.

Examples

The command below plots a phase shift keying (PSK) signal constellation with 32 points.

```
modmap('psk',32);
```



The script below maps a digital signal using the 32-point PSK constellation. It then adds noise and computes the resulting error rate while demapping. Your results might vary because the example uses random numbers.

```
M = 32; Fd = 1; Fs = 3;
x = randint(100,1,M); % Original signal
```

```
y = modmap(x,Fd,Fs,'psk',M); % Mapped signal, using 32-ary PSK
ynoisy = y+.1*rand(100*Fs,2); % Mapped signal with noise added
z = demodmap(ynoisy,Fd,Fs,'psk',M); % Demapped noisy signal
s = symerr(x,z) % Number of errors after demapping noisy signal

s =

     8
```

See Also

demodmap, dmod, dmodce, amod, amodce, apkconst

Purpose

Convert octal numbers to decimal numbers

Syntax

```
d = oct2dec(c)
```

Description

`d = oct2dec(c)` converts an octal matrix `c` to a decimal matrix `d`, element by element. In both octal and decimal representations, the rightmost digit is the least significant.

Examples

The command below converts a 2-by-2 octal matrix.

```
d = oct2dec([12 144;0 25])

d =

    10   100
     0    21
```

For instance, the octal number 144 is equivalent to the decimal number 100 because 144 (octal) = $1*8^2 + 4*8^1 + 4*8^0 = 64 + 32 + 4 = 100$.

See Also

`bi2de`

poly2trellis

Purpose

Convert convolutional code polynomials to trellis description

Syntax

```
trellis = poly2trellis(ConstraintLength,CodeGenerator);
trellis = poly2trellis(ConstraintLength,CodeGenerator,...
   FeedbackConnection);
```

Description

The `poly2trellis` function accepts a polynomial description of a convolutional encoder and returns the corresponding trellis structure description. The output of `poly2trellis` is suitable as an input to the `convenc` and `vitdec` functions, and as a mask parameter for the Convolutional Encoder, Viterbi Decoder, and APP Decoder blocks in the Communications Blockset.

`trellis = poly2trellis(ConstraintLength,CodeGenerator)` performs the conversion for a rate k/n feedforward encoder. `ConstraintLength` is a 1-by-k vector that specifies the delay for the encoder's k input bit streams. `CodeGenerator` is a k-by-n matrix of octal numbers that specifies the n output connections for each of the encoder's k input bit streams.

`trellis = poly2trellis(ConstraintLength,CodeGenerator,... FeedbackConnection)` is the same as the syntax above, except that it applies to a feedback, not feedforward, encoder. `FeedbackConnection` is a 1-by-k vector of octal numbers that specifies the feedback connections for the encoder's k input bit streams.

For both syntaxes, the output is a MATLAB structure whose fields are as in the table below.

Fields of the Output Structure trellis for a Rate k/n Code

Field in trellis Structure	Dimensions	Meaning
`numInputSymbols`	Scalar	Number of input symbols to the encoder: 2^k
`numOutputSymbols`	Scalar	Number of output symbols from the encoder: 2^n
`numStates`	Scalar	Number of states in the encoder

Fields of the Output Structure trellis for a Rate k/n Code (Continued)

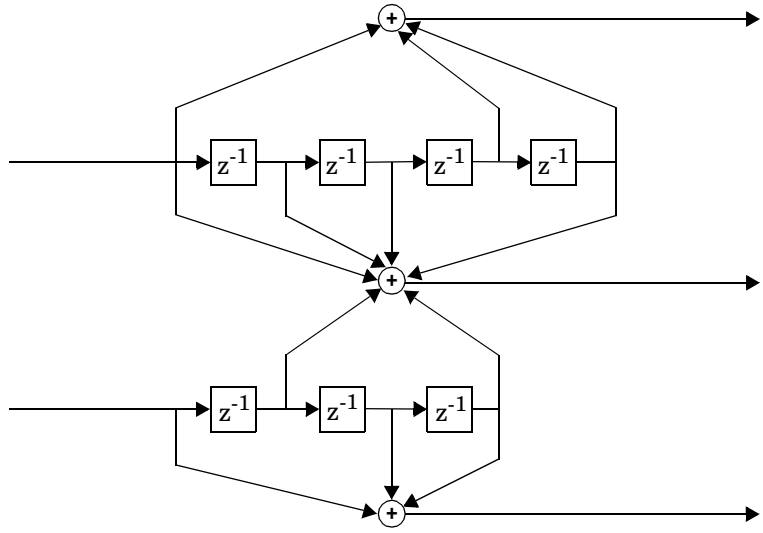
Field in trellis Structure	Dimensions	Meaning
`nextStates`	`numStates`-by-2^k matrix	Next states for all combinations of current state and current input
`outputs`	`numStates`-by-2^k matrix	Outputs (in octal) for all combinations of current state and current input

For more about this structure, see the reference page for the `istrellis` function.

Examples

An example of a rate 1/2 encoder is in "Polynomial Description of a Convolutional Encoder" on page 2-46.

As another example, consider the rate 2/3 feedforward convolutional encoder depicted in the figure below. The reference page for the `convenc` function includes an example that uses this encoder.



For this encoder, the `ConstraintLength` vector is [5,4] and the `CodeGenerator` matrix is [23,35,0; 0,5,13]. The output below reveals part of the corresponding trellis structure description of this encoder.

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])

trellis =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

The scalar field `trellis.numInputSymbols` has the value 4 because the combination of two input bit streams can produce four different input symbols. Similarly, `trellis.numOutputSymbols` is 8 because the three output bit streams can produce eight different output symbols.

The scalar field `trellis.numStates` is 128 (that is, 2^7) because each of the encoder's seven memory registers can have one of two binary values.

To get details about the matrix fields `trellis.nextStates` and `trellis.outputs`, inquire specifically about them. As an example, the command below displays the first five rows of the 128-by-4 matrix `trellis.nextStates`.

```
trellis.nextStates(1:5,:)

ans =

     0    64     8    72
     0    64     8    72
     1    65     9    73
     1    65     9    73
     2    66    10    74
```

This first row indicates that if the encoder starts in the zeroth state and receives input bits of 00, 01, 10, or 11, respectively, then the next state will be the 0th, 64th, 8th, or 72nd state, respectively. The 64th state means that the bottom-left memory register in the diagram contains the value 1, while the other six memory registers contain zeros.

See Also

`istrellis`, `convenc`, `vitdec`

Purpose

Find primitive polynomials for a Galois field

Syntax

```
pr = primpoly(m)
pr = primpoly(m,opt)
pr = primpoly(m...,'nodisplay')
```

Description

`pr = primpoly(m)` returns the primitive polynomial for GF(`2^m`), where `m` is an integer between `2` and `16`. The Command Window displays the polynomial using "`D`" as an indeterminate quantity. The output argument `pr` is an integer whose binary representation indicates the coefficients of the polynomial.

`pr = primpoly(m,opt)` returns one or more primitive polynomials for GF(`2^m`). The output `pol` depends on the argument *`opt`* as shown in the table below. Each element of the output argument `pr` is an integer, whose binary representation indicates the coefficients of the corresponding polynomial. If no primitive polynomial satisfies the constraints, then `pr` is empty.

opt	Meaning of pr
'**min**'	One primitive polynomial for GF(`2^m`) having the smallest possible number of nonzero terms
'**max**'	One primitive polynomial for GF(`2^m`) having the greatest possible number of nonzero terms
'**all**'	All primitive polynomials for GF(`2^m`)
Positive integer k	All primitive polynomials for GF(`2^m`) that have k nonzero terms

`pr = primpoly(m...,'nodisplay')` prevents the function from displaying the result as polynomials in "`D`" in the Command Window. The output argument `pr` is unaffected by the `'nodisplay'` option.

Examples

The example below illustrates the formats that `primpoly` uses in the Command Window and in the output argument `pr`.

```
pr = primpoly(4)
```

```
Primitive polynomial(s) =

D^4+D^1+1

pr =

    19
```

The examples below illustrate the display options and the use of the *opt* argument.

```
pr1 = primpoly(5,'max','nodisplay')

pr1 =

    61

pr2 = primpoly(5,'min')

Primitive polynomial(s) =

D^5+D^2+1

pr2 =

    37

pr3 = primpoly(5,2)
No primitive polynomial satisfies the given constraints.

pr3 =

     []
pr4 = primpoly(5,3);

Primitive polynomial(s) =

D^5+D^2+1
D^5+D^3+1
```

See Also

`isprimitive`

Purpose

Demap a message from a QASK square signal constellation

Syntax

```
msg = qaskdeco(inphase,quadr,M);
msg = qaskdeco(inphase,quadr,M,mnmx);
```

Description

`msg = qaskdeco(inphase,quadr,M)` demaps the message signal `msg` from the `M`-ary quadrature amplitude shift keying (QASK) square signal constellation points given in the vectors `inphase` and `quadr`. Here `inphase` lists the in-phase components of the points and `quadr` lists the corresponding quadrature components. `M` must be a power of 2. `qaskdeco` uses the default minimum/maximum value of the in-phase component and quadrature component. The defaults corresponding to small values of `M` are in the table on the reference page for the function `qaskenco`.

Note To see how symbols are mapped to the constellation points, generate a constellation plot using `qaskenco(M)`.

`msg = qaskdeco(inphase,quadr,M,mnmx)` is the same as the syntax above, except that `mnmx` specifies the minimum and maximum in-phase and quadrature component values. `mnmx` is a 2-by-2 matrix of the form shown below.

$$\text{mnmx} = \begin{bmatrix} \text{in-phase minimum} & \text{in-phase maximum} \\ \text{quadrature minimum} & \text{quadrature maximum} \end{bmatrix}$$

Examples

The commands below show that `qaskdeco` and `qaskenco` are inverse operations.

```
msg = [0 3 5 3 2 5]'; M = 8;
[inphase,quadr] = qaskenco(msg,M); % Map the message.
newmsg = qaskdeco(inphase,quadr,M) % Demap to recover data.

newmsg =

     0
     3
     5
```

```
     3
     2
     5
```

See Also qaskenco, decode, demodmap

Purpose

Map a message to a QASK square signal constellation

Syntax

```
qaskenco(M)
qaskenco(msg,M)
[inphase,quadr] = qaskenco(M)
[inphase,quadr] = qaskenco(msg,M)
```

Description

`qaskenco(M)` plots the square signal constellation for `M`-ary quadrature amplitude shift keying (QASK) modulation, labeling the `M` points with numbers in the range [0, `M`-1]. `M` must be a power of 2. If `M` is a perfect square, then `qaskenco` labels the constellation points so as to implement Gray code.

`qaskenco(msg,M)` is the same as the syntax above, except that only those points with labels in the vector `msg` are plotted. The elements in `msg` must be integers in the range [0, `M`-1].

`[inphase,quadr] = qaskenco(M)` returns vectors `inphase` and `quadr` that represent the coordinates of the points in the signal constellation for `M`-ary QASK modulation. `inphase` gives the in-phase component of each point and `quadr` gives the quadrature component of each point. `M` must be a power of 2.

`[inphase,quadr] = qaskenco(msg,M)` is the same as the syntax above, except that `inphase` and `quadr` represent only those constellation points with labels in the vector `msg`. (These labels are the same number labels that appear in the plot that the command `qaskenco(msg,M)` produces.) The elements in `msg` must be integers in the range [0, `M`-1].

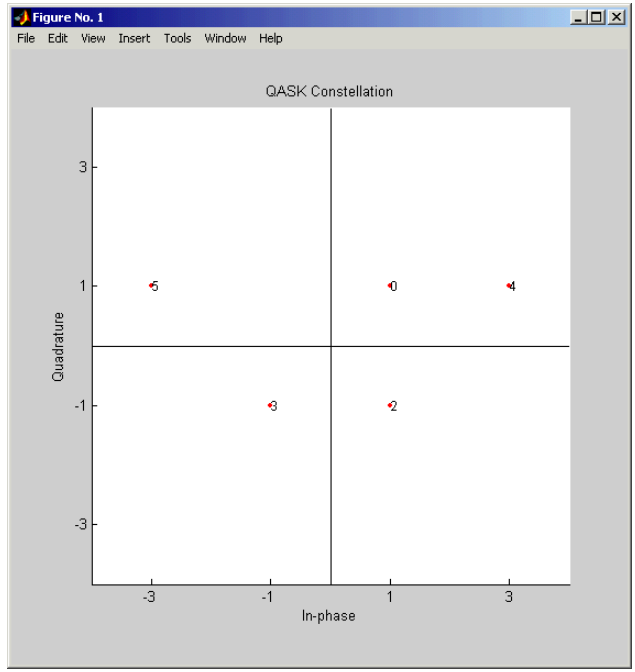
The table below shows the maximum value of `inphase` and `quadr`, for several small values of `M`.

M	Maximum of inphase and quadr	M	Maximum of inphase and quadr
2	1	32	5
4	1	64	7
8	3 (maximum of `quadr` is 1)	128	11
16	3	256	15

Examples

The command below displays that part of the 8-ary QASK square constellation that corresponds to the points in the digital message signal [0 3 4 3 2 5].

```
qaskenco([0 3 4 3 2 5],8)
```



The commands below capture the same information in vectors `inphase` and `quadr` instead of in a plot.

```
[inphase,quadr] = qaskenco([0 3 5 3 2 5],8);
inphase'

ans =

     1    -1    -3    -1     1    -3


quadr'
```

```
ans =

     1    -1     1    -1    -1     1
```

The command below captures in `inphase` and `quadr` the coordinates of all eight points in the 8-ary QASK square constellation.

```
[inphase2,quadr2] = qaskenco(8);
```

See Also

`encode`, `modmap`, `qaskdeco`

Purpose

Produce a quantization index and a quantized output value

Syntax

```
index = quantiz(sig,partition);
[index,quants] = quantiz(sig,partition,codebook);
[index,quants,distor] = quantiz(sig,partition,codebook);
```

Description

`index = quantiz(sig,partition)` returns the quantization levels in the real vector signal `sig` using the parameter `partition`. `partition` is a real vector whose entries are in strictly ascending order. If `partition` has length n, then `index` is a column vector whose kth entry is

- 0 if $sig(k) \le partition(1)$
- m if $partition(m) < sig(k) \le partition(m+1)$
- n if $partition(n) < sig(k)$

`[index,quants] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `codebook` prescribes a value for each partition in the quantization and `quants` contains the quantization of `sig` based on the quantization levels and prescribed values. `codebook` is a vector whose length exceeds the length of `partition` by one. `quants` is a row vector whose length is the same as the length of `sig`. `quants` is related to `codebook` and `index` by

```
quants(ii) = codebook(index(ii)+1);
```

where `ii` is an integer between 1 and `length(sig)`.

`[index,quants,distor] = quantiz(sig,partition,codebook)` is the same as the syntax above, except that `distor` estimates the mean square distortion of this quantization data set.

Examples

The command below rounds several numbers between 1 and 100 up to the nearest multiple of ten. `quants` contains the rounded numbers, and `index` tells which quantization level each number is in.

```
[index,quants] = quantiz([3 34 84 40 23],10:10:90,10:10:100)

index =

     0
     3
```

```
     8
     3
     2


quants =

    10    40    90    40    30
```

See Also

`lloyds`, `dpcmenco`, `dpcmdeco`

randerr

Purpose

Generate bit error patterns

Syntax

```
out = randerr(m);
out = randerr(m,n);
out = randerr(m,n,errors);
out = randerr(m,n,errors,state);
```

Description

For all syntaxes, `randerr` treats each row of `out` independently.

`out = randerr(m)` generates an `m`-by-`m` binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

`out = randerr(m,n)` generates an `m`-by-`n` binary matrix, each row of which has exactly one nonzero entry in a random position. Each allowable configuration has an equal probability.

`out = randerr(m,n,errors)` generates an `m`-by-`n` binary matrix, where `errors` determines how many nonzero entries are in each row:

- If `errors` is a scalar, then it is the number of nonzero entries in each row.
- If `errors` is a row vector, then it lists the possible number of nonzero entries in each row.
- If `errors` is a matrix having two rows, then the first row lists the possible number of nonzero entries in each row and the second row lists the probabilities that correspond to the possible error counts.

Once `randerr` determines the *number* of nonzero entries in a given row, each configuration of that number of nonzero entries has equal probability.

`out = randerr(m,n,prob,state)` is the same as the syntax above, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

Examples

To generate an 8-by-7 binary matrix, each row of which is equally likely to have either zero or two nonzero entries, use the command below.

```
out = randerr(8,7,[0 2])
```

```
out =

     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     1     0     0     0     1
     1     0     1     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     0     0     0
     0     0     0     0     1     1     0
     1     0     1     0     0     0     0
```

To alter the scenario above by making it three times as likely that a row has two nonzero entries, use the command below instead. Notice that the second row of the `error` parameter sums to one.

```
out2 = randerr(8,7,[0 2; .25 .75])

out =

     0     0     0     0     0     0     0
     1     0     0     0     0     0     1
     1     0     0     0     0     0     1
     0     0     0     1     0     1     0
     0     0     0     0     0     0     0
     0     1     0     0     0     0     1
     0     0     0     0     0     0     0
     1     0     0     0     1     0     0
```

See Also

`rand`, `randsrc`, `randint`

randint

Purpose

Generate matrix of uniformly distributed random integers

Syntax

```
out = randint
out = randint(m);
out = randint(m,n);
out = randint(m,n,rg);
out = randint(m,n,rg,state);
```

Description

`out = randint` generates a random scalar that is either 0 or 1, with equal probability.

`out = randint(m)` generates an `m`-by-`m` binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n)` generates an `m`-by-`n` binary matrix, each of whose entries independently takes the value 0 with probability 1/2.

`out = randint(m,n,rg)` generates an `m`-by-`n` integer matrix. If `rg` is zero, then `out` is a zero matrix. Otherwise, the entries are uniformly distributed and independently chosen from the range

- [0, `rg`-1] if `rg` is a positive integer
- [`rg`+1, 0] if `rg` is a negative integer
- Between `min` and `max`, inclusive, if `rg = [min,max]` or `[max,min]`

`out = randint(m,n,rg,state)` is the same as the syntax above, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed in the range from 0 to 7, you can use either of the following commands.

```
out = randint(10,10,[0,7]);
out = randint(10,10,8);
```

See Also

`rand`, `randsrc`, `randerr`

Purpose

Generate random matrix using prescribed alphabet

Syntax

```
out = randsrc;
out = randsrc(m);
out = randsrc(m,n);
out = randsrc(m,n,alphabet);
out = randsrc(m,n,[alphabet; prob]);
out = randsrc(m,n,...,state);
```

Description

`out = randsrc` generates a random scalar that is either -1 or 1, with equal probability.

`out = randsrc(m)` generates an `m`-by-`m` matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n)` generates an `m`-by-`n` matrix, each of whose entries independently takes the value -1 with probability 1/2, and 1 with probability 1/2.

`out = randsrc(m,n,alphabet)` generates an `m`-by-`n` matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Each entry in `alphabet` occurs in `out` with equal probability. Duplicate values in `alphabet` are ignored.

`out = randsrc(m,n,[alphabet; prob])` generates an `m`-by-`n` matrix, each of whose entries is independently chosen from the entries in the row vector `alphabet`. Duplicate values in `alphabet` are ignored. The row vector `prob` lists corresponding probabilities, so that the symbol `alphabet(k)` occurs with probability `prob(k)`, where `k` is any integer between one and the number of columns of `alphabet`. The elements of `prob` must add up to one.

`out = randsrc(m,n,...,state);` is the same as the two preceding syntaxes, except that it first resets the state of the uniform random number generator `rand` to the integer `state`.

Examples

To generate a 10-by-10 matrix whose elements are uniformly distributed among members of the set {-3,-1,1,3}, you can use either of these commands.

```
out = randsrc(10,10,[-3 -1 1 3]);
out = randsrc(10,10,[-3 -1 1 3; .25 .25 .25 .25]);
```

To skew the probability distribution so that -1 and 1 each occur with probability .3, while -3 and 3 each occur with probability .2, use this command.

```
out = randsrc(10,10,[-3 -1 1 3; .2 .3 .3 .2]);
```

See Also

`rand`, `randint`, `randerr`

Purpose

Design a raised cosine FIR filter

Syntax

```
b = rcosfir(R,n_T,rate,T);
b = rcosfir(R,n_T,rate,T,filter_type);
rcosfir(...);
rcosfir(...,colr);
[b,sample_time] = rcosfir(...);
```

Optional Inputs

Input	Default Value
`n_T`	`[-3,3]`
`rate`	`5`
`T`	`1`

Description

The `rcosfir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes `'fir'`. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t/T)}{(\pi t/T)} \cdot \frac{\cos(\pi R t/T)}{(1-4R^2t^2/T^2)}$$

`b = rcosfir(R,n_T,rate,T)` designs a raised cosine filter and returns a vector `b` of length `(n_T(2) - n_T(1))*rate + 1`. The filter's rolloff factor is `R`, where $0 \le R \le 1$. `T` is the duration of each bit in seconds. `n_T` is a length-two vector that indicates the number of symbol periods before and after the peak response. `rate` is the number of points in each input symbol period of length `T`. `rate` must be greater than 1. The input sample rate is `T` samples per second, while the output sample rate is `T*rate` samples per second.

The order of the FIR filter is

```
(n_T(2)-n_T(1))*rate
```

The arguments `n_T`, `rate`, and `T` are optional inputs whose default values are [-3,3], 5, and 1, respectively.

`b = rcosfir(R,n_T,rate,T,filter_type)` designs a square-root raised cosine filter if *`filter_type`* is '**`sqrt`**'. If *`filter_type`* is '**`normal`**' then this syntax is the same as the previous one.

The impulse response of a square root raised cosine filter is

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

`rcosfir(...)` produces plots of the time and frequency responses of the raised cosine filter.

`rcosfir(...,colr)` uses the string `colr` to determine the plotting color. The choices for `colr` are the same as those listed for the `plot` function.

`[b,sample_time] = rcosfir(...)` returns the FIR filter and its sample time.

Examples

The commands below compare different rolloff factors.

```
rcosfir(0);
subplot(211); hold on;
subplot(212); hold on;
rcosfir(.5,[],[],[],[],'r-');
rcosfir(1,[],[],[],[],'g-');
```

See Also

`rcosiir`, `rcosflt`, `rcosine`, `firrcos`, `rcosdemo`

References

Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose

Filter the input signal using a raised cosine filter

Syntax

```
y = rcosflt(x,Fd,Fs);
y = rcosflt(x,Fd,Fs,'filter_type',r,delay,tol);
y = rcosflt(x,Fd,Fs,'filter_type/Fs',r,delay,tol);
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den);
y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den,delay);
y = rcosflt(x,Fd,Fs,'filter_type/filter/Fs',num,den...);
[y,t] = rcosflt(...);
```

Optional Inputs

Input	Default Value
filter_type	**fir/normal**
r	0.5
delay	3
tol	0.01
den	1

Description

The function `rcosflt` passes an input signal through a raised cosine filter. You can either let `rcosflt` design a raised cosine filter automatically or you can specify the raised cosine filter yourself using input arguments.

Designing the Filter Automatically

`y = rcosflt(x,Fd,Fs)` designs a raised cosine FIR filter and then filters the input signal `x` using it. The sample frequency for the digital input signal `x` is `Fd`, and the sample frequency for the output signal `y` is `Fs`. The ratio `Fs/Fd` must be an integer. In the course of filtering, `rcosflt` upsamples the data by a factor of `Fs/Fd`, by inserting zeros between samples. The order of the filter is `1+2*delay*Fs/Fd`, where `delay` is 3 by default. If `x` is a vector, then the sizes of `x` and `y` are related by this equation.

```
length(y) = (length(x) + 2 * delay)*Fs/Fd
```

Otherwise, `y` is a matrix, each of whose columns is the result of filtering the corresponding column of `x`.

`y = rcosflt(x,Fd,Fs,'filter_type',r,delay,tol)` designs a raised cosine FIR or IIR filter and then filters the input signal `x` using it. The ratio `Fs/Fd` must be an integer. `r` is the rolloff factor for the filter, a real number in the range [0, 1]. `delay` is the filter's group delay, measured in input samples. The actual group delay in the filter design is `delay/Fd` seconds. The input `tol` is the tolerance in the IIR filter design. FIR filter design does not use `tol`.

The characteristics of `x`, `Fd`, `Fs`, and `y` are as in the first syntax.

The fourth input argument, '*filter_type*', is a string that determines the type of filter that `rcosflt` should design. Use one of the values in the table below.

Values of filter_type to Determine the Type of Filter

Type of Filter	Value of opt
FIR raised cosine filter	**fir** or **fir/normal**
IIR raised cosine filter	**iir** or **iir/normal**
Square-root FIR raised cosine filter	**fir/sqrt**
Square-root IIR raised cosine filter	**iir/sqrt**

`y = rcosflt(x,Fd,Fs,'filter_type/Fs',r,delay,tol)` is the same as the previous syntax, except that it assumes that `x` has sample frequency `Fs`. This syntax does not upsample `x` any further. If `x` is a vector, then the relative sizes of `x` and `y` are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

As before, if `x` is a nonvector matrix, then `y` is a matrix each of whose columns is the result of filtering the corresponding column of `x`.

Specifying the Filter Using Input Arguments

`y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den)` filters the input signal `x` using a filter whose transfer function numerator and denominator are given in `num` and den, respectively. If *`filter_type`* includes **`fir`**, then omit `den`. This syntax uses the same arguments `x`, `Fd`, `Fs`, and *`filter_type`* as explained in the first and second syntaxes above.

`y = rcosflt(x,Fd,Fs,'filter_type/filter',num,den,delay)` uses `delay` in the same way that the `rcosine` function uses it. This syntax assumes that the filter described by `num`, `den`, and `delay` was designed using `rcosine`.

As before, if `x` is a nonvector matrix, then `y` is a matrix each of whose columns is the result of filtering the corresponding column of `x`.

`y = rcosflt(x,Fd,Fs,'filter_type/filter/Fs',num,den...)` is the same as the earlier syntaxes, except that it assumes that `x` has sample frequency `Fs` instead of `Fd`. This syntax does not upsample `x` any further. If `x` is a vector, then the relative sizes of `x` and `y` are related by this equation.

```
length(y) = length(x) + (2 * delay * Fs/Fd)
```

Additional Output

`[y,t] = rcosflt(...)` outputs `t`, a vector that contains the sampling time points of `y`.

See Also

`rcosine`, `rcosfir`, `rcosiir`, `rcosdemo`, `grpdelay`

References

Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

Purpose

Design a raised cosine IIR filter

Syntax

```
[num,den] = rcosiir(R,T_delay,rate,T,tol);
[num,den] = rcosiir(R,T_delay,rate,T,tol,filter_type);
rcosiir(...);
rcosiir(...,colr);
[num,den,sample_time] = rcosiir(...);
```

Optional Inputs

Input	Default Value
`T_delay`	3
`rate`	5
`T`	1
`tol`	0.01

Description

The `rcosiir` function designs the same filters that the `rcosine` function designs when the latter's `type_flag` argument includes `'iir'`. However, `rcosine` is somewhat easier to use.

The time response of the raised cosine filter has the form

$$h(t) = \frac{\sin(\pi t/T)}{(\pi t/T)} \cdot \frac{\cos(\pi R t/T)}{(1-4R^2t^2/T^2)}$$

`[num,den] = rcosiir(R,T_delay,rate,T,tol)` designs an IIR approximation of an FIR raised cosine filter, and returns the numerator and denominator of the IIR filter. The filter's rolloff factor is `R`, where $0 \le R \le 1$. `T` is the symbol period in seconds. The filter's group delay is `T_delay` symbol periods. `rate` is the number of sample points in each interval of duration `T`. `rate` must be greater than 1. The input sample rate is `T` samples per second, while the output sample rate is `T*rate` samples per second. If `tol` is an integer greater than 1, then it becomes the order of the IIR filter; if `tol` is less than 1, then it indicates the relative tolerance for `rcosiir` to use when selecting the order based on the singular values.

The arguments `T_delay`, `rate`, `T`, and `tol` are optional inputs whose default values are 3, 5, 1, and 0.01, respectively.

`[num,den] = rcosiir(R,T_delay,rate,T,tol,`*`filter_type`*`)` designs a square-root raised cosine filter if *`filter_type`* is '**`sqrt`**'. If *`filter_type`* is '**`normal`**' then this syntax is the same as the previous one.

`rcosiir(...)` plots the time and frequency responses of the raised cosine filter.

`rcosiir(...,colr)` uses the string `colr` to determine the plotting color. The choices for `colr` are the same as those listed for the `plot` function.

`[num,den,sample_time] = rcosiir(...)` returns the transfer function and the sample time of the IIR filter.

Examples

The script below compares different values of `T_delay`.

```
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 ii = R
   rcosiir(0,ii,[],[],[],[],col(find(R==ii),:));
end;
```

This example shows how the filter's frequency response more closely approximates that of the ideal raised cosine filter as `T_delay` increases.

See Also

`rcosfir`, `rcosflt`, `rcosine`, `rcosdemo`, `grpdelay`

References

Kailath, Thomas, *Linear Systems*, Englewood Cliffs, N.J., Prentice-Hall, 1980.

Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

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,r);
[num,den] = rcosine(Fd,Fs,type_flag,r,delay);
[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol);
```

Description

`num = rcosine(Fd,Fs)` designs a finite impulse response (FIR) raised cosine filter and returns its transfer function. The digital input signal has sampling frequency `Fd`. The sampling frequency for the filter is `Fs`. The ratio `Fs/Fd` must be a positive integer greater than 1. The default rolloff factor is .5. The filter's group delay, which is the time between the input to the filter and the filter's peak response, is three input samples. Equivalently, the group delay is `3/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag)` designs a raised cosine filter using directions in the string variable *`type_flag`*. Filter types are listed in the table below, along with the corresponding values of *`type_flag`*.

Types of Filter and Corresponding Values of type_flag

Type of Filter	Value of type_flag
Finite impulse response (FIR)	'**`default`**' or '**`fir/normal`**'
Infinite impulse response (IIR)	'**`iir`**' or '**`iir/normal`**'
Square-root raised cosine FIR	'**`sqrt`**' or '**`fir/sqrt`**'
Square-root raised cosine IIR	'**`iir/sqrt`**'

The default tolerance value in IIR filter design is 0.01.

`[num,den] = rcosine(Fd,Fs,type_flag,r)` specifies the rolloff factor, `r`. The rolloff factor is a real number in the range [0, 1].

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay)` specifies the filter's group delay, measured in input samples. `delay` is a positive integer. The actual group delay in the filter design is `delay/Fd` seconds.

`[num,den] = rcosine(Fd,Fs,type_flag,r,delay,tol)` specifies the tolerance in the IIR filter design. FIR filter design does not use `tol`.

See Also

`rcosflt`, `rcosiir`, `rcosfir`, `rcosdemo`, `grpdelay`

References

Korn, Israel, *Digital Communications*, New York, Van Nostrand Reinhold, 1985.

rsdec

Purpose

Reed-Solomon decoder

Syntax

```
decoded = rsdec(code,n,k)
decoded = rsdec(code,n,k,genpoly)
decoded = rsdec(...,paritypos)
[decoded,cnumerr] = rsdec(...)
[decoded,cnumerr,ccode] = rsdec(...)
```

Description

`decoded = rsdec(code,n,k)` attempts to decode the received signal in `code` using an [`n`,`k`] Reed-Solomon decoding process with the narrow-sense generator polynomial. `code` is a Galois array of symbols having m bits each. Each `n`-element row of `code` represents a corrupted systematic codeword, where the parity symbols are at the end and the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, then `rsdec` assumes that `code` is a corrupted version of a shortened code.

In the Galois array `decoded`, each row represents the attempt at decoding the corresponding row in `code`. A *decoding failure* occurs if a row of `code` contains more than `(n-k)/2` errors. In this case, `rsdec` forms the corresponding row of `decoded` by merely removing `n-k` symbols from the end of the row of `code`.

`decoded = rsdec(code,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree `n-k`. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

`decoded = rsdec(...,paritypos)` specifies whether `rsdec` appends or prepends the parity symbols to the input message to form `decoded`. The string *`paritypos`* can be either '**`end`**' or '**`beginning`**'. The default is '**`end`**'. If *`paritypos`* is '**`beginning`**', then a decoding failure causes `rsdec` to remove `n-k` symbols from the beginning rather than the end of the row.

`[decoded,cnumerr] = rsdec(...)` returns a column vector `cnumerr`, each element of which is the number of corrected errors in the corresponding row of `code`. A value of `-1` in `cnumerr` indicates a decoding failure in that row in `code`.

`[decoded,cnumerr,ccode] = rsdec(...)` returns `ccode`, the corrected version of `code`. The Galois array `ccode` has the same format as `code`. If a decoding failure occurs in a certain row of `code`, then the corresponding row in `ccode` contains that row unchanged.

Examples

The example below encodes three message words using a (7,3) Reed-Solomon encoder. It then corrupts the code by introducing one error in the first code word, two errors in the second code word, and three errors in the third code word. Then `rsdec` tries to decode the corrupted code.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6; 5 1 1],m); % Three rows of m-bit symbols
code = rsenc(msg,n,k);
errors = gf([2 0 0 0 0 0 0; 3 4 0 0 0 0 0; 5 6 7 0 0 0 0],m);
noisycode = code + errors;
[dec,cnumerr] = rsdec(noisycode,n,k)

dec = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     2     7     3
     4     0     6
     4     0     0


cnumerr =

     1
     2
    -1
```

The output shows that `rsdec` successfully corrects the errors in the first two code words and recovers the first two original message words. However, a (7,3) Reed-Solomon code can correct at most two errors in each word, so `rsdec` cannot recover the third message word. The elements of the vector `cnumerr` indicate the number of corrected errors in the first two words and also indicate the decoding failure in the third word.

For additional examples, see “Creating and Decoding Reed-Solomon Codes” on page 2-41.

Algorithm

`rsdec` uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” below.

Limitations

`n` and `k` must differ by an even integer. The maximum allowable value of `n` is 65535.

See Also

`rsenc`, `gf`, `rsgenpoly`

References

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

[2] Berlekamp, Elwyn R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

Purpose

Decode an ASCII file that was encoded using Reed-Solomon code

Syntax

```
rsdecof(file_in,file_out);
rsdecof(file_in,file_out,err_cor);
```

Description

This function is the inverse process of the function `rsencof` in that it decodes a file that `rsencof` encoded.

`rsdecof(file_in,file_out)` decodes the ASCII file `file_in` that was previously created by the function `rsencof` using an error-correction capability of 5. The decoded message is written to `file_out`. Both `file_in` and `file_out` are string variables.

Note If the number of characters in `file_in` is not an integer multiple of 127, then the function appends `char(4)` symbols to the data it must decode. If you encode and then decode a file using `rsencof` and `rsdecof`, respectively, then the decoded file might have `char(4)` symbols at the end that the original file does not have.

`rsdecof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 $*$ `err_cor`. The value in `err_cor` must match the value used in `rsencof` when `file_in` was created.

Examples

An example is on the reference page for `rsencof`.

See Also

`rsencof`

Purpose

Reed-Solomon encoder

Syntax

```
code = rsenc(msg,n,k);
code = rsenc(msg,n,k,genpoly);
code = rsenc(...,paritypos);
```

Description

`code = rsenc(msg,n,k)` encodes the message in `msg` using an [`n`,`k`] Reed-Solomon code with the narrow-sense generator polynomial. `msg` is a Galois array of symbols having m bits each. Each `k`-element row of `msg` represents a message word, where the leftmost symbol is the most significant symbol. `n` is at most 2^m-1. If `n` is not exactly 2^m-1, then `rsenc` uses a shortened Reed-Solomon code. Parity symbols are at the end of each word in the output Galois array `code`.

`code = rsenc(msg,n,k,genpoly)` is the same as the syntax above, except that a nonempty value of `genpoly` specifies the generator polynomial for the code. In this case, `genpoly` is a Galois row vector that lists the coefficients, in order of descending powers, of the generator polynomial. The generator polynomial must have degree `n-k`. To use the default narrow-sense generator polynomial, set `genpoly` to `[]`.

`code = rsenc(...,`*`paritypos`*`)` specifies whether `rsenc` appends or prepends the parity symbols to the input message to form `code`. The string *`paritypos`* can be either '**`end`**' or '**`beginning`**'. The default is '**`end`**'.

Examples

The example below encodes two message words using a (7,3) Reed-Solomon encoder.

```
m = 3; % Number of bits per symbol
n = 2^m-1; k = 3; % Word lengths for code
msg = gf([2 7 3; 4 0 6],m); % Two rows of m-bit symbols
code = rsenc(msg,n,k)

code = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     2     7     3     3     6     7     6
     4     0     6     4     2     2     0
```

For additional examples, see "Representing Words for Reed-Solomon Codes" on page 2-39 and "Creating and Decoding Reed-Solomon Codes" on page 2-41.

Limitations

`n` and `k` must differ by an even integer. The maximum allowable value of `n` is 65535.

See Also

`rsdec`, `gf`, `rsgenpoly`

rsencof

Purpose

Encode an ASCII file using Reed-Solomon code

Syntax

```
rsencof(file_in,file_out);
rsencof(file_in,file_out,err_cor);
```

Description

`rsencof(file_in,file_out)` encodes the ASCII file `file_in` using (127, 117) Reed-Solomon code. The error-correction capability of this code is 5 for each block of 127 codeword characters. This function writes the encoded text to the file `file_out`. Both `file_in` and `file_out` are string variables.

`rsencof(file_in,file_out,err_cor)` is the same as the first syntax, except that `err_cor` specifies the error-correction capability for each block of 127 codeword characters. The message length is 127 - 2 * `err_cor`.

Note If the number of characters in `file_in` is not an integer multiple of 127 - 2 * `err_cor`, then the function appends `char(4)` symbols to `file_out`.

Examples

The file *`matlabroot`*`/toolbox/comm/comm/oct2dec.m` contains text help for the `oct2dec` function in this toolbox. The commands below encode the file using `rsencof` and then decode it using `rsdecof`.

```
file_in = [matlabroot '/toolbox/comm/comm/oct2dec.m'];
file_out = 'encodedfile'; % Or use another filename
rsencof(file_in,file_out) % Encode the file.

file_in = file_out;
file_out = 'decodedfile'; % Or use another filename
rsdecof(file_in,file_out) % Decode the file.
```

To see the original file and the decoded file in the MATLAB workspace, use the commands below (or similar ones if you modified the filenames above).

```
type oct2dec.m

type decodedfile
```

See Also

`rsdecof`

Purpose

Generator polynomial of Reed-Solomon code

Syntax

```
genpoly = rsgenpoly(n,k)
genpoly = rsgenpoly(n,k,prim_poly)
genpoly = rsgenpoly(n,k,prim_poly,b)
[genpoly,t] = rsgenpoly(...)
```

Description

`genpoly = rsgenpoly(n,k)` returns the narrow-sense generator polynomial of a Reed-Solomon code with codeword length `n` and message length `k`. The codeword length `n` must have the form 2^m-1 for some integer m, and `n-k` must be an even integer. The output `genpoly` is a Galois row vector that represents the coefficients of the generator polynomial in order of descending powers. The narrow-sense generator polynomial is $(X - A^1)(X - A^2)...(X - A^{2t})$ where A is a root of the default primitive polynomial for the field GF(`n+1`) and t is the code's error-correction capability, `(n-k)/2`.

`genpoly = rsgenpoly(n,k,prim_poly)` is the same as the syntax above, except that `prim_poly` specifies the primitive polynomial for GF(`n+1`) that has A as a root. `prim_poly` is an integer whose binary representation indicates the coefficients of the primitive polynomial. To use the default primitive polynomial GF(`n+1`), set `prim_poly` to `[]`.

`genpoly = rsgenpoly(n,k,prim_poly,b)` returns the generator polynomial $(X - A^b)(X - A^{b+1})...(X - A^{b+2t-1})$ where `b` is an integer, A is a root of `prim_poly` and t is the code's error-correction capability, `(n-k)/2`.

`[genpoly,t] = rsgenpoly(...)` returns `t`, the code error-correction capability of the code.

Examples

The examples below create Galois row vectors that represent generator polynomials for a [7,3] Reed-Solomon code. The vectors `g` and `g2` both represent the narrow-sense generator polynomial, but with respect to different primitive elements A. More specifically, `g2` is defined such that A is a root of the primitive polynomial $D^3 + D^2 + 1$ for GF(8), not of the default primitive polynomial $D^3 + D + 1$. The vector `g3` represents the generator polynomial $(X - A^3)(X - A^4)(X - A^5)(X - A^6)$, where A is a root of $D^3 + D^2 + 1$ in GF(8).

```
g = rsgenpoly(7,3)
```

```
g = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

     1     3     1     2     3

g2 = rsgenpoly(7,3,13) % Use nondefault primitive polynomial.

g2 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     1     4     5     1     5

g3 = rsgenpoly(7,3,13,3) % Use b = 3.

g3 = GF(2^3) array. Primitive polynomial = D^3+D^2+1 (13 decimal)

Array elements =

     1     7     1     6     7
```

As another example, the command below shows that the default narrow-sense generator polynomial for a [15,11] Reed-Solomon code is $X^4 + (A^3 + A^2 + 1)X^3 + (A^3 + A^2)X^2 + A^3X + (A^2 + A + 1)$ where A is a root of the default primitive polynomial for GF(16).

```
gp = rsgenpoly(15,11)

gp = GF(2^4) array. Primitive polynomial = D^4+D+1 (19 decimal)

Array elements =

     1    13    12     8     7
```

For additional examples, see “Parameters for Reed-Solomon Codes” on page 2-40.

Limitations

n and k must differ by an even integer. The maximum allowable value of n is 65535.

See Also `gf`, `rsenc`, `rsdec`

scatterplot

Purpose

Generate a scatter plot

Syntax

```
scatterplot(x);
scatterplot(x,n);
scatterplot(x,n,offset);
scatterplot(x,n,offset,plotstring);
scatterplot(x,n,offset,plotstring,h);
h = scatterplot(...);
```

Description

`scatterplot(x)` produces a scatter plot for the signal `x`. The interpretation of `x` depends on its shape and complexity:

- If `x` is a real two-column matrix, then `scatterplot` interprets the first column as in-phase components and the second column as quadrature components.
- If `x` is a complex vector, then `scatterplot` interprets the real part as in-phase components and the imaginary part as quadrature components.
- If x is a real vector, then `scatterplot` interprets it as a real signal.

`scatterplot(x,n)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the first value. That is, the function decimates `x` by a factor of `n` before plotting.

`scatterplot(x,n,offset)` is the same as the first syntax, except that the function plots every `n`th value of the signal, starting from the `(offset+1)`st value in `x`.

`scatterplot(x,n,offset,plotstring)` is the same as the syntax above, except that `plotstring` determines the plotting symbol, line type, and color for the plot. `plotstring` is a string whose format and meaning are the same as in the `plot` function.

`scatterplot(x,n,offset,plotstring,h)` is the same as the syntax above, except that the scatter plot is in the figure whose handle is `h`, rather than a new figure. `h` must be a handle to a figure that `scatterplot` previously generated. To plot multiple signals in the same figure, use `hold on`.

`h = scatterplot(...)` is the same as the earlier syntaxes, except that `h` is the handle to the figure that contains the scatter plot.

Examples

See "Example: Scatter Plots" on page 2-11 or the example on the reference page for `demodmap`. Both examples illustrate how to plot multiple signals in a single scatter plot.

For an online demonstration, type `playshow scattereyedemo`.

See Also

`eyediagram`, `plot`, `scattereyedemo`, `scatter`

shift2mask

Purpose

Convert shift to mask vector for a shift register configuration

Syntax

```
mask = shift2mask(prpoly,shift)
```

Description

`mask = shift2mask(prpoly,shift)` returns the mask that is equivalent to the shift (or offset) specified by `shift`, for a linear feedback shift register whose connections are specified by the primitive polynomial `prpoly`. The `prpoly` input can have one of these formats:

- A binary vector that lists the coefficients of the primitive polynomial in order of descending powers
- An integer scalar whose binary representation gives the coefficients of the primitive polynomial, where the least significant bit is the constant term

The `shift` input is an integer scalar.

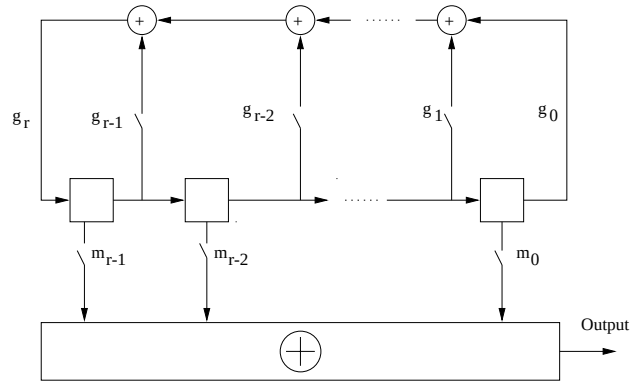
Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, then the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

Definition of Equivalent Mask

The equivalent mask for the shift s is the remainder after dividing the polynomial x^s by the primitive polynomial. The vector `mask` represents the remainder polynomial by listing the coefficients in order of descending powers.

Shifts, Masks, and Pseudonoise Sequence Generators

Linear feedback shift registers are part of an implementation of a pseudonoise sequence generator. Below is a schematic diagram of a pseudonoise sequence generator. All adders perform addition modulo 2.



The primitive polynomial determines the state of each switch labeled g_k, while the mask determines the state of each switch labeled m_k. The lower half of the diagram shows the implementation of the shift, which delays the starting point of the output sequence. If the shift is zero, then the m_0 switch is closed while all other m_k switches are open. The table below indicates how the shift affects the shift register's output.

	T = 0	**T = 1**	**T = 2**	…	**T = s**	**T = s+1**
Shift = 0	x_0	x_1	x_2	…	x_s	x_{s+1}
Shift = s > 0	x_s	x_{s+1}	x_{s+2}	…	x_{2s}	x_{2s+1}

If you have the Communications Blockset and want to generate a pseudonoise sequence in a Simulink model, see the reference page for the PN Sequence Generator block in the blockset's documentation set.

Examples

The command below converts a shift of 5 into the equivalent mask $x^3 + x + 1$, for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$.

```
mk = shift2mask([1 1 0 0 1],5)
```

```
mk =

     1     0     1     1
```

See Also

`mask2shift`, `deconv`, `isprimitive`, `primpoly`

References

[1] Lee, J. S., and L. E. Miller, *CDMA Systems Engineering Handbook*, Boston, Artech House, 1998.

[2] Simon, Marvin K., Jim K. Omura, et al., *Spread Spectrum Communications Handbook*, New York, McGraw-Hill, 1994.

Purpose

Compute number of symbol errors and symbol error rate

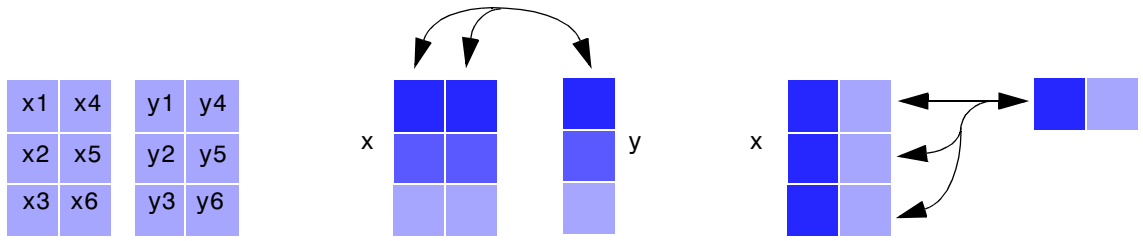
Syntax

```
[number,ratio] = symerr(x,y);
[number,ratio] = symerr(x,y,flg);
[number,ratio,loc] = symerr(...)
```

Description

For All Syntaxes

The `symerr` function compares binary representations of elements in `x` with those in `y`. The schematics below illustrate how the shapes of `x` and `y` determine which elements `symerr` compares.



(a) Compares x1 with y1, x2 with y2, and so on.

(b) Compares column vector y with each column of matrix x

(c) Compares row vector y with each row of matrix x

The output `number` is a scalar or vector that indicates the number of elements that differ. The size of `number` is determined by the optional input *`flg`* and by the dimensions of `x` and `y`. The output `ratio` equals `number` divided by the total number of elements in the *smaller* input.

For Specific Syntaxes

`[number,ratio] = symerr(x,y)` compares the elements in `x` and `y`. The sizes of `x` and `y` determine which elements are compared:

- If `x` and `y` are matrices of the same dimensions, then `symerr` compares `x` and `y` element-by-element. `number` is a scalar. See schematic (a) in the figure.
- If one is a row (respectively, column) vector and the other is a two-dimensional matrix, then `symerr` compares the vector element-by-element with *each row (resp., column)* of the matrix. The length of the vector must equal the number of columns (resp., rows) in the matrix.

`number` is a column (resp., row) vector whose mth entry indicates the number of elements that differ when comparing the vector with the mth row (resp., column) of the matrix. See schematics (b) and (c) in the figure.

`[number,ratio] = symerr(x,y,flg)` is similar to the previous syntax, except that *`flg`* can override the defaults that govern which elements `symerr` compares and how `symerr` computes the outputs. The values of *`flg`* are '**`overall`**', '**`column-wise`**', and '**`row-wise`**'. The table below describes the differences that result from various combinations of inputs. In all cases, `ratio` is `number` divided by the total number of elements in `y`.

Comparing a Two-Dimensional Matrix x with Another Input y

Shape of y	flg	Type of Comparison	number
Two-dimensional matrix	'**`overall`**' (default)	Element-by-element	Total number of symbol errors
	'**`column-wise`**'	mth column of x vs. mth column of y	Row vector whose entries count symbol errors in each column
	'**`row-wise`**'	mth row of x vs. mth row of y	Column vector whose entries count symbol errors in each row
Column vector	'**`overall`**'	y vs. each column of x	Total number of symbol errors
	'**`column-wise`**' (default)	y vs. each column of x	Row vector whose entries count symbol errors in each column of x
Row vector	'**`overall`**'	y vs. each row of x	Total number of symbol errors
	'**`row-wise`**' (default)	y vs. each row of x	Column vector whose entries count symbol errors in each row of x

`[number,ratio,loc] = symerr(...)` returns a binary matrix `loc` that indicates which elements of `x` and `y` differ. An element of `loc` is zero if the corresponding comparison yields no discrepancy, and one otherwise.

Examples

On the reference page for `biterr`, the last example uses `symerr`.

The command below illustrates how `symerr` works when one argument is a vector and the other is a matrix. It compares the vector `[1,2,3]'` to the columns

$$\begin{bmatrix}1\\3\\3\end{bmatrix}, \begin{bmatrix}1\\2\\3\end{bmatrix}, \begin{bmatrix}3\\2\\8\end{bmatrix}, \text{ and } \begin{bmatrix}1\\2\\3\end{bmatrix}$$

of the matrix.

```
num = symerr([1 2 3]',[1 1 3 1;3 2 2 2; 3 3 8 3])

num =

     1     0     2     0
```

As another example, the command below illustrates the use of *`flg`* to override the default row-by-row comparison. Notice that `number` and `ratio` are scalars.

```
format rat; [number,ratio,loc] = symerr([1 2; 3 4],...
[1 3],'overall')

number =

      3


ratio =

     3/4


loc =

      0            1
      1            1
```

See Also

`biterr`

syndtable

Purpose

Produce syndrome decoding table

Syntax

```
t = syndtable(h);
```

Description

`t = syndtable(h)` returns a decoding table for an error-correcting binary code having codeword length n and message length k. `h` is an (n-k)-by-n parity-check matrix for the code. `t` is a 2^{n-k}-by-n binary matrix. The rth row of `t` is an error pattern for a received binary codeword whose syndrome has decimal integer value r-1. (The syndrome of a received codeword is its product with the transpose of the parity-check matrix.) In other words, the rows of `t` represent the coset leaders from the code's standard array.

When converting between binary and decimal values, the leftmost column is interpreted as the *most* significant digit. This differs from the default convention in the `bi2de` and `de2bi` commands.

Examples

An example is in "Decoding Table" on page 2-32.

See Also

`decode`, `hammgen`, `gfcosets`

References

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

Purpose

Convert a vector into a matrix

Syntax

```
mat = vec2mat(vec,matcol);
mat = vec2mat(vec,matcol,padding);
[mat,padded] = vec2mat(...);
```

Description

`mat = vec2mat(vec,matcol)` converts the vector `vec` into a matrix with `matcol` columns, creating one row at a time. If the length of `vec` is not a multiple of `matcol`, then extra zeros are placed in the last row of `mat`. The matrix `mat` has `ceil(length(vec)/matcol)` rows.

`mat = vec2mat(vec,matcol,padding)` is the same as the first syntax, except that the extra entries placed in the last row of `mat` are not necessarily zeros. The extra entries are taken from the matrix `padding`, in order. If `padding` has fewer entries than are needed, then the last entry is used repeatedly.

`[mat,padded] = vec2mat(...)` returns an integer `padded` that indicates how many extra entries were placed in the last row of `mat`.

Note `vec2mat` is similar to the built-in MATLAB function `reshape`. However, given a vector input, `reshape` creates a matrix one *column* at a time instead of one row at a time. Also, `reshape` requires the input and output matrices to have the same number of entries, whereas `vec2mat` places extra entries in the output matrix if necessary.

Examples

```
vec = [1 2 3 4 5];
[mat,padded] = vec2mat(vec,3)

mat =

     1     2     3
     4     5     0


padded =

     1
```

```
[mat2,padded2] = vec2mat(vec,4)

mat2 =

     1     2     3     4
     5     0     0     0


padded2 =

     3

mat3 = vec2mat(vec,4,[10 9 8; 7 6 5; 4 3 2])

mat3 =

     1     2     3     4
     5    10     7     4
```

See Also

reshape

Purpose

Convolutionally decode binary data using the Viterbi algorithm

Syntax

```
decoded = vitdec(code,trellis,tblen,opmode,dectype);
decoded = vitdec(code,trellis,tblen,opmode,'soft',nsdec);
decoded = vitdec(...,'cont',...,initmetric,initstates,initinputs);
[decoded,finalmetric,finalstates,finalinputs] =...
   vitdec(...,'cont',...);
```

Description

`decoded = vitdec(code,trellis,tblen,opmode,dectype)` decodes the vector `code` using the Viterbi algorithm. The MATLAB structure `trellis` specifies the convolutional encoder that produced `code`; the format of `trellis` is described in "Trellis Description of a Convolutional Encoder" on page 2-50 and the reference page for the `istrellis` function. `code` contains one or more symbols, each of which consists of `log2(trellis.numOutputSymbols)` bits. Each symbol in the vector `decoded` consists of `log2(trellis.numInputSymbols)` bits. `tblen` is a positive integer scalar that specifies the traceback depth.

The string *opmode* indicates the decoder's operation mode and its assumptions about the corresponding encoder's operation. Choices are in the table below.

Values of opmode Input

Value	Meaning
`'trunc'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric.
`'term'`	The encoder is assumed to have both started and ended at the all-zeros state. The decoder traces back from the all-zeros state.
`'cont'`	The encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. A delay equal to `tblen` symbols elapses before the first decoded symbol appears in the output.

The string *dectype* indicates the type of decision that the decoder makes, and influences the type of data the decoder expects in `code`. Choices are in the table below.

Values of dectype Input

Value	Meaning
'**unquant**'	`code` contains real input values, where 1 represents a logical zero and -1 represents a logical one.
'**hard**'	`code` contains binary input values.
'**soft**'	For soft-decision decoding, use the syntax below. Note that `nsdec` is required for soft-decision decoding.

Syntax for Soft Decision Decoding

`decoded = vitdec(code,trellis,tblen,`*`opmode`*`,'soft',nsdec)` decodes the vector `code` using soft-decision decoding. `code` consists of integers between 0 and `2^nsdec-1`, where 0 represents the most confident 0 and `2^nsdec-1` represents the most confident 1.

Additional Syntaxes for Continuous Operation Mode

`decoded = vitdec(...,'cont',...,initmetric,initstates,initinputs)` is the same as the earlier syntaxes, except that the decoder starts with its state metrics, traceback states, and traceback inputs specified by `initmetric`, `initstates`, and `initinputs`, respectively. Each real number in `initmetric` represents the starting state metric of the corresponding state. `initstates` and `initinputs` jointly specify the initial traceback memory of the decoder; both are `trellis.numStates`-by-`tblen` matrices. `initstates` consists of integers between 0 and `trellis.numStates-1`. If the encoder schematic has more than one input stream, then the shift register that receives the first input stream provides the least significant bits in `initstates`, while the shift register that receives the last input stream provides the most significant bits in `initstates`. The vector `initinputs` consists of integers between 0 and `trellis.numInputSymbols-1`. To use default values for all of the last three arguments, specify them as `[],[],[]`.

`[decoded,finalmetric,finalstates,finalinputs] = ...`
`vitdec(...,'cont',...)` is the same as the earlier syntaxes, except that the final three output arguments return the state metrics, traceback states, and traceback inputs, respectively, at the end of the decoding process. `finalmetric` is a vector with `trellis.numStates` elements that correspond to the final state metrics. `finalstates` and `finalinputs` are both matrices of size `trellis.numStates`-by-`tblen`. The elements of `finalstates` have the same format as those of `initstates`.

Examples

The example below encodes random data and adds noise. Then it decodes the noisy code three times to illustrate the three decision types that `vitdec` supports. Notice that for unquantized and soft decisions, the output of `convenc` does not have the same data type that `vitdec` expects for the input code, so it is necessary to manipulate `ncode` before invoking `vitdec`.

```
trel = poly2trellis(3,[6 7]); % Define trellis.
msg = randint(100,1,2,123); % Random data
code = convenc(msg,trel); % Encode.
ncode = rem(code + randerr(200,1,[0 1;.95 .05]),2); % Add noise.
tblen = 3; % Traceback length
% Use hard decisions.
decoded1 = vitdec(ncode,trel,tblen,'cont','hard');
% Use unquantized decisions.
ucode = 1-2*ncode; % +1 & -1 represent zero & one, respectively.
decoded2 = vitdec(ucode,trel,tblen,'cont','unquant');
% Use soft decisions.
% To prepare for soft-decision decoding, map to decision values.
[x,qcode] = quantiz(1-2*ncode,[-.75 -.5 -.25 0 .25 .5 .75],...
[7 6 5 4 3 2 1 0]); % Values in qcode are between 0 and 2^3-1.
decoded3 = vitdec(qcode',trel,tblen,'cont','soft',3);

% Compute bit error rates, using the fact that the decoder
% output is delayed by tblen symbols.
[n1,r1] = biterr(decoded1(tblen+1:end),msg(1:end-tblen));
[n2,r2] = biterr(decoded2(tblen+1:end),msg(1:end-tblen));
[n3,r3] = biterr(decoded3(tblen+1:end),msg(1:end-tblen));
disp(['The bit error rates are:   ',num2str([r1 r2 r3])])

The bit error rates are:   0.020619    0.020619    0.020619
```

The example below illustrates how to use the final state and initial state arguments when invoking `vitdec` repeatedly. Notice that `[decoded4;decoded5]` is the same as `decoded6`.

```
trel = poly2trellis(3,[6 7]);
code = convenc(randint(100,1,2,123),trel);
% Decode part of code, recording final state for later use.
[decoded4,f1,f2,f3] = vitdec(code(1:100),trel,3,'cont','hard');
% Decode the rest of code, using state input arguments.
decoded5 = vitdec(code(101:200),trel,3,'cont','hard',f1,f2,f3);
% Decode the entire code in one step.
decoded6 = vitdec(code,trel,3,'cont','hard');
isequal(decoded6,[decoded4; decoded5])

ans =

     1
```

See Also

`convenc`, `poly2trellis`, `istrellis`, `vitsimdemo`

References

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

Purpose

Generate white Gaussian noise

Syntax

```
y = wgn(m,n,p);
y = wgn(m,n,p,imp);
y = wgn(m,n,p,imp,state);
y = wgn(...,powertype);
y = wgn(...,outputtype);
```

Description

`y = wgn(m,n,p)` generates an `m`-by-`n` matrix of white Gaussian noise. `p` specifies the power of `y` in decibels relative to a watt. The default load impedance is 1 ohm.

`y = wgn(m,n,p,imp)` is the same as the previous syntax, except that `imp` specifies the load impedance in ohms.

`y = wgn(m,n,p,imp,state)` is the same as the previous syntax, except that `wgn` first resets the state of the normal random number generator `randn` to the integer `state`.

`y = wgn(...,`*`powertype`*`)` is the same as the previous syntaxes, except that the string *`powertype`* specifies the units of `p`. Choices for *`powertype`* are '**`dBW`**', '**`dBm`**', and '**`linear`**'.

`y = wgn(...,`*`outputtype`*`)` is the same as the previous syntaxes, except that the string *`outputtype`* specifies whether the noise is real or complex. Choices for *`outputtype`* are '**`real`**' and '**`complex`**'. If *`outputtype`* is '**`complex`**', then the real and imaginary parts of `y` each have a noise power of `p`/2.

Examples

To generate a column vector of length 100 containing real white Gaussian noise of power 0 dBW, use this command:

```
y1 = wgn(100,1,0);
```

To generate a column vector of length 100 containing complex white Gaussian noise, each component of which has a noise power of 0 dBW, use this command:

```
y2 = wgn(100,1,0,'complex');
```

See Also

`randn`, `awgn`

A

Appendix: Galois Fields of Odd Characteristic

A *Galois field* is an algebraic field that has a finite number of members. The number of elements is always of the form p^m, where p is a prime number and m is a positive integer. This section describes how to work with fields that have p^m, where p is an *odd* prime number. To work with Galois fields having an even number of elements, see "Galois Field Computations" on page 2-93. The topics covered here are as follows.

Galois Field Terminology

Throughout this section, p is an odd prime number and m is a positive integer.

Also, this document uses a few terms that are not used consistently in the literature. The definitions adopted here appear in van Lint [4].

- A *primitive element* of $GF(p^m)$ is a cyclic generator of the group of nonzero elements of $GF(p^m)$. This means that every nonzero element of the field can be expressed as the primitive element raised to some integer power. Primitive elements are called A throughout this section.
- A *primitive polynomial* for $GF(p^m)$ is the minimal polynomial of some primitive element of $GF(p^m)$. As a consequence, it has degree m and is irreducible.

Representing Elements of Galois Fields

This section discusses how to represent Galois field elements using this toolbox's exponential format and polynomial format. It also describes a way to list all elements of the Galois field, because some functions use such a list as an input argument. Finally, it discusses the nonuniqueness of representations of Galois field elements.

The elements of GF(p) can be represented using the integers from 0 to p-1.

When m is at least 2, $GF(p^m)$ is called an extension field. Integers alone cannot represent the elements of $GF(p^m)$ in a straightforward way. MATLAB uses two main conventions for representing elements of $GF(p^m)$: the exponential format and the polynomial format.

Note Both the exponential format and the polynomial format are relative to your choice of a particular primitive element A of $GF(p^m)$.

Exponential Format

This format uses the property that every nonzero element of $GF(p^m)$ can be expressed as A^c for some integer c between 0 and p^m-2. Higher exponents are not needed, because the theory of Galois fields implies that every nonzero element of $GF(p^m)$ satisfies the equation $x^{q-1} = 1$ where $q = p^m$.

The use of the exponential format is shown in the table below.

Element of $GF(p^m)$	MATLAB Representation of the Element
0	`-Inf`
$A^0 = 1$	`0`
A^1	`1`
…	…
A^{q-2} where `q` = p^m	`q-2`

Although `-Inf` is the standard exponential representation of the zero element, all negative integers are equivalent to `-Inf` when used as *input* arguments in exponential format. This equivalence can be useful; for example, see the concise line of code at the end of the section "Default Primitive Polynomials" on page A-7.

Note The equivalence of all negative integers and `-Inf` as exponential formats means that, for example, -1 does *not* represent A^{-1}, the multiplicative inverse of A. Instead, -1 represents the zero element of the field.

Polynomial Format

The polynomial format uses the property that every element of GF(p^m) can be expressed as a polynomial in A with exponents between 0 and m-1, and coefficients in GF(p). In the polynomial format, the element

$$A(1) + A(2)\,A + A(3)\,A^2 + \ldots + A(m)\,A^{m-1}$$

is represented in MATLAB by the vector

```
[A(1) A(2) A(3) ... A(m)]
```

Note The Galois field functions in this toolbox represent a polynomial as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

List of All Elements of a Galois Field

Some Galois field functions in this toolbox require an argument that lists all elements of an extension field GF(p^m). This is again relative to a particular primitive element A of GF(p^m). The proper format for the list of elements is that of a matrix having p^m rows, one for each element of the field. The matrix has m columns, one for each coefficient of a power of A in the polynomial format shown in "Polynomial Format" above. The first row contains only zeros because it corresponds to the zero element in GF(p^m). If k is between 2 and p^m, then the kth row specifies the polynomial format of the element A^{k-2}.

The minimal polynomial of A aids in the computation of this matrix, because it tells how to express A^m in terms of lower powers of A. For example, the table below lists the elements of $GF(3^2)$, where A is a root of the primitive polynomial $2 + 2x + x^2$. This polynomial allows repeated use of the substitution

$$A^2 = -2 - 2A = 1 + A$$

when performing the computations in the middle column of the table.

Elements of GF(9)

Exponential Format	Polynomial Format	Row of MATLAB Matrix of Elements
A^{-Inf}	0	0 0
A^0	1	1 0
A^1	A	0 1
A^2	1+A	1 1
A^3	$A + A^2 = A + 1 + A = 1 + 2A$	1 2
A^4	$A + 2A^2 = A + 2 + 2A = 2$	2 0
A^5	2A	0 2
A^6	$2A^2 = 2 + 2A$	2 2
A^7	$2A + 2A^2 = 2A + 2 + 2A = 2 + A$	2 1

Example

An automatic way to generate the matrix whose rows are in the third column of the table above is to use the code below.

```
p = 3; m = 2;
% Use the primitive polynomial 2 + 2x + x^2 for GF(9).
prim_poly = [2 2 1];
field = gftuple([-1:p^m-2]',prim_poly,p);
```

The `gftuple` function is discussed in more detail in “Converting and Simplifying Element Formats” on page A-8.

Nonuniqueness of Representations

A given field has more than one primitive element. If two primitive elements have different minimal polynomials, then the corresponding matrices of elements will have their rows in a different order. If the two primitive elements share the same minimal polynomial, then the matrix of elements of the field is the same.

Note You can use whatever primitive element you want, as long as you understand how the inputs and outputs of Galois field functions depend on the choice of *some* primitive polynomial. It is usually best to use the same primitive polynomial throughout a given script or function.

Other ways in which representations of elements are not unique arise from the equations that Galois field elements satisfy. For example, an exponential format of 8 in GF(9) is really the same as an exponential format of 0, because $A^8 = 1 = A^0$ in GF(9). As another example, the substitution mentioned just before the table "Elements of GF(9)" shows that the polynomial format [0 0 1] is really the same as the polynomial format [1 1].

Default Primitive Polynomials

This toolbox provides a *default* primitive polynomial for each extension field. You can retrieve this polynomial using the `gfprimdf` function. The command

```
prim_poly = gfprimdf(m,p); % If m and p are already defined
```

produces the standard row-vector representation of the default minimal polynomial for GF(p^m).

For example, the command below shows that the default primitive polynomial for GF(9) is $2 + x + x^2$, *not* the polynomial used in "List of All Elements of a Galois Field" on page A-4.

```
gfprimdf(2,3)

ans =

     2     1     1
```

To generate a list of elements of GF(p^m) using the default primitive polynomial, use the command

```
field = gftuple([-1:p^m-2]',m,p);
```

Converting and Simplifying Element Formats

This section describes how to convert between the exponential and polynomial formats for Galois field elements, as well as how to simplify a given representation.

Converting to Simplest Polynomial Format

The `gftuple` function produces the simplest polynomial representation of an element of $GF(p^m)$, given either an exponential representation or a polynomial representation of that element. This can be useful for generating the list of elements of $GF(p^m)$ that other functions require.

Using `gftuple` requires three arguments: one representing an element of $GF(p^m)$, one indicating the primitive polynomial that MATLAB should use when computing the output, and the prime p. The table below indicates how `gftuple` behaves when given the first two arguments in various formats.

Behavior of gftuple Depending on Format of First Two Inputs

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Exponential format; c = any integer	Integer m > 1	Polynomial format of A^c, where A is a root of the *default* primitive polynomial for $GF(p^m)$
Example: `tp = gftuple(6,2,3); % c = 6 here`		
Exponential format; c = any integer	Vector of coefficients of primitive polynomial	Polynomial format of A^c, where A is a root of the *given* primitive polynomial
Example: `polynomial = gfprimdf(2,3); tp = gftuple(6,polynomial,3); % c = 6 here`		

Behavior of gftuple Depending on Format of First Two Inputs (Continued)

How to Specify Element	How to Indicate Primitive Polynomial	What gftuple Produces
Polynomial format of any degree	Integer m > 1	Polynomial format of degree < m, using *default* primitive polynomial for GF(p^m) to simplify
Example: `tp = gftuple([0 0 0 0 0 0 1],2,3);`		
Polynomial format of any degree	Vector of coefficients of primitive polynomial	Polynomial format of degree < m, using the *given* primitive polynomial for GF(p^m) to simplify
Example: `polynomial = gfprimdf(2,3); tp = gftuple([0 0 0 0 0 0 1],polynomial,3);`		

The four examples that appear in the table above all produce the same vector `tp = [2, 1]`, but their different inputs to `gftuple` correspond to the lines of the table. Each example expresses the fact that

$$A^6 = 2+A$$

where A is a root of the (default) primitive polynomial $2 + x + x^2$ for GF(3^2).

Example

This example shows how `gfconv` and `gftuple` combine to multiply two polynomial-format elements of GF(3^4). Initially, `gfconv` multiplies the two polynomials, treating the primitive element as if it were a variable. This produces a high-order polynomial, which `gftuple` simplifies using the polynomial equation that the primitive element satisfies. The final result is the simplest polynomial format of the product.

```
p = 3; m = 4;
a = [1 2 0 1]; b = [2 2 1 2];
notsimple = gfconv(a,b,p) % a times b, using high powers of alpha

notsimple =

     2     0     2     0     0     1     2

simple = gftuple(notsimple,m,p) %Highest exponent of alpha is m-1
```

```
simple =

     2     1     0     1
```

Example: Generating a List of Galois Field Elements

This example applies the conversion functionality to the task of generating a matrix that lists all elements of a Galois field. A matrix that lists all field elements is an input argument in functions such as `gfadd` and `gfmul`. The variables `field1` and `field2` below have the format that such functions expect.

```
p = 5; % Or any prime number
m = 4; % Or any positive integer
field1 = gftuple([-1:p^m-2]',m,p);

prim_poly = gfprimdf(m,p); % Or any primitive polynomial
% for GF(p^m)
field2 = gftuple([-1:p^m-2]',prim_poly,p);
```

Converting to Simplest Exponential Format

The same function `gftuple` also produces the simplest exponential representation of an element of $GF(p^m)$, given either an exponential representation or a polynomial representation of that element. To retrieve this output, use the syntax

```
[polyformat, expformat] = gftuple(...)
```

The input format and the output `polyformat` are as in the table "Behavior of gftuple Depending on Format of First Two Inputs" on page A-8. In addition, the variable `expformat` contains the simplest exponential format of the element represented in `polyformat`. It is *simplest* in the sense that the exponent is either `-Inf` or a number between 0 and p^m-2.

Example

To recover the exponential format of the element 2 + A that the previous section considered, use the commands below. In this case, `polyformat` contains redundant information, while `expformat` contains the desired result.

```
[polyformat, expformat] = gftuple([2 1],2,3)
```

```
polyformat =

     2     1

expformat =

     6
```

This output appears at first to contradict the information in the table "Elements of GF(9)", but in fact it does not. The table uses a different primitive element; two plus that primitive element has the polynomial and exponential formats shown below. The output below reflects the information in the bottom line of the table.

```
prim_poly = [2 2 1];
[polyformat, expformat] = gftuple([2 1],prim_poly,3)

polyformat =

     2     1


expformat =

     7
```

Arithmetic in Galois Fields

You can add, subtract, multiply, and divide elements of Galois fields using the functions `gfadd`, `gfsub`, `gfmul`, and `gfdiv`, respectively. Each of these functions has a mode for prime fields and a mode for extension fields.

Arithmetic in Prime Fields

Arithmetic in GF(p) is the same as arithmetic modulo p. The functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` accept two arguments that represent elements of GF(p) as integers between 0 and p-1. The third argument specifies p.

Example: Addition Table for GF(5)

The code below constructs an addition table for GF(5). If `a` and `b` are between 0 and 4, then the element `gfp_add(a+1,b+1)` represents the sum `a+b` in GF(5). For example, `gfp_add(3,5) = 1` because 2+4 is 1 modulo 5.

```
p = 5;
row = 0:p-1;
table = ones(p,1)*row;
gfp_add = gfadd(table,table',p)

gfp_add =

     0     1     2     3     4
     1     2     3     4     0
     2     3     4     0     1
     3     4     0     1     2
     4     0     1     2     3
```

Other values of `p` produce tables for different prime fields GF(p). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(p).

Arithmetic in Extension Fields

The same arithmetic functions can add elements of $GF(p^m)$ when $m > 1$, but the format of the arguments is more complicated than in the case above. In general, arithmetic in extension fields is more complicated than arithmetic in prime fields; see the works listed in “Selected Bibliography for Galois Fields” on page A-20 for details about how the arithmetic operations work.

When working in extension fields, the functions `gfadd`, `gfmul`, `gfsub`, and `gfdiv` use the first two arguments to represent elements of $GF(p^m)$ in exponential format. The third argument, which is required, lists all elements of $GF(p^m)$ as described in "List of All Elements of a Galois Field" on page A-4. The result is in exponential format.

Example: Addition Table for GF(9)

The code below constructs an addition table for $GF(3^2)$, using exponential formats relative to a root of the default primitive polynomial for GF(9). If `a` and `b` are between -1 and 7, then the element `gfpm_add(a+2,b+2)` represents the sum of A^a and A^b in GF(9). For example, `gfpm_add(4,6) = 5` because

$$A^2 + A^4 = A^5$$

Using the fourth and sixth rows of the matrix `field`, you can verify that

$$A^2 + A^4 = (1 + 2A) + (2 + 0A) = 3 + 2A = 0 + 2A = A^5 \text{ modulo } 3.$$

```
p = 3; m = 2; % Work in GF(3^2).
field = gftuple([-1:p^m-2]',m,p); % Construct list of elements.
row = -1:p^m-2;
table = ones(p^m,1)*row;
gfpm_add = gfadd(table,table',field)

gfpm_add =

  -Inf     0     1     2     3     4     5     6     7
     0     4     7     3     5  -Inf     2     1     6
     1     7     5     0     4     6  -Inf     3     2
     2     3     0     6     1     5     7  -Inf     4
     3     5     4     1     7     2     6     0  -Inf
     4  -Inf     6     5     2     0     3     7     1
     5     2  -Inf     7     6     3     1     4     0
     6     1     3  -Inf     0     7     4     2     5
     7     6     2     4  -Inf     1     0     5     3
```

Note If you used a different primitive polynomial, then the tables would look different. This makes sense because the ordering of the rows and columns of the tables was based on that particular choice of primitive polynomial and not on any natural ordering of the elements of GF(9).

Other values of `p` and `m` produce tables for different prime fields GF(p^m). Replacing `gfadd` by `gfmul`, `gfsub`, or `gfdiv` produces a table for the corresponding arithmetic operation in GF(p^m).

Polynomials over Prime Fields

A polynomial over GF(p) is a polynomial whose coefficients are elements of GF(p). The Communications Toolbox provides functions for

- Changing polynomials in cosmetic ways
- Performing polynomial arithmetic
- Characterizing polynomials as primitive or irreducible
- Finding roots of polynomials in a Galois field

Note The Galois field functions in this toolbox represent a polynomial over GF(p) for odd values of p as a vector that lists the coefficients in order of *ascending* powers of the variable. This is the opposite of the order that other MATLAB functions use.

Cosmetic Changes of Polynomials

To display the traditionally formatted polynomial that corresponds to a row vector containing coefficients, use `gfpretty`. To truncate a polynomial by removing all zero-coefficient terms that have exponents *higher* than the degree of the polynomial, use `gftrunc`. For example,

```
polynom = gftrunc([1 20 394 10 0 0 29 3 0 0])

polynom =

     1    20   394    10     0     0    29     3

gfpretty(polynom)

                                 2       3       6       7
                    1 + 20 X + 394 X  + 10 X  + 29 X  + 3 X
```

Note If you do not use a fixed-width font, then the spacing in the display might not look correct.

Polynomial Arithmetic

The functions `gfadd` and `gfsub` add and subtract, respectively, polynomials over GF(p). The `gfconv` function multiplies polynomials over GF(p). The `gfdeconv` function divides polynomials in GF(p), producing a quotient polynomial and a remainder polynomial. For example, the commands below show that $2 + x + x^2$ times $1 + x$ over the field GF(3) is $2 + 2x^2 + x^3$.

```
a = gfconv([2 1 1],[1 1],3)

a =

     2     0     2     1

[quot, remd] = gfdeconv(a,[2 1 1],3)

quot =

     1     1

remd =


     0
```

The previously discussed functions `gfadd` and `gfsub` add and subtract, respectively, polynomials. Because it uses a vector of coefficients to represent a polynomial, MATLAB does not distinguish between adding two polynomials and adding two row vectors elementwise.

Characterization of Polynomials

Given a polynomial over GF(p), the `gfprimck` function determines whether it is irreducible and/or primitive. By definition, if it is primitive then it is irreducible; however, the reverse is not necessarily true. The `gfprimdf` and `gfprimfd` functions return primitive polynomials.

Given an element of $GF(p^m)$, the `gfminpol` function computes its minimal polynomial over GF(p).

Example

For example, the code below reflects the irreducibility of all minimal polynomials. However, the minimal polynomial of a nonprimitive element is not a primitive polynomial.

```
p = 3; m = 4;
% Use default primitive polynomial here.

prim_poly = gfminpol(1,m,p);
ckprim = gfprimck(prim_poly,p);
% ckprim = 1, since prim_poly represents a primitive polynomial.

notprimpoly = gfminpol(5,m,p);
cknotprim = gfprimck(notprimpoly,p);
% cknotprim = 0 (irreducible but not primitive)
% since alpha^5 is not a primitive element when p = 3.

ckreducible = gfprimck([0 1 1],p);
% ckreducible = -1 since the polynomial is reducible.
```

Roots of Polynomials

Given a polynomial over GF(p), the `gfroots` function finds the roots of the polynomial in a suitable extension field GF(p^m). There are two ways to tell MATLAB the degree m of the extension field GF(p^m), as shown in the table below.

Formats for Second Argument of gfroots

Second Argument	Represents
A positive integer	m as in GF(p^m). MATLAB uses the default primitive polynomial in its computations.
A row vector	A primitive polynomial for GF(p^m). Here m is the degree of this primitive polynomial.

Example: Roots of a Polynomial in GF(9)

The code below finds roots of the polynomial $1 + x^2 + x^3$ in GF(9) and then checks that they are indeed roots. The exponential format of elements of GF(9) is used throughout.

```
p = 3; m = 2;
field = gftuple([-1:p^m-2]',m,p); % List of all elements of GF(9)
% Use default primitive polynomial here.
polynomial = [1 0 1 1]; % 1 + x^2 + x^3
rts =gfroots(polynomial,m,p) % Find roots in exponential format
% Check that each one is actually a root.
for ii = 1:3
   root = rts(ii);
   rootsquared = gfmul(root,root,field);
   rootcubed = gfmul(root,rootsquared,field);
   answer(ii)=...
      gfadd(gfadd(0,rootsquared,field),rootcubed,field);
   % Recall that 1 is really alpha to the zero power.
   % If answer = -Inf, then the variable root represents
   % a root of the polynomial.
end
answer
```

The output shows that A^0 (which equals 1), A^5, and A^7 are roots.

```
roots =

     0
     5
     7


answer =

  -Inf  -Inf  -Inf
```

See the reference page for `gfroots` to see how `gfroots` can also provide you with the polynomial formats of the roots and the list of all elements of the field.

Other Galois Field Functions

See the reference pages for information about these other Galois field functions in the Communications Toolbox:

- `gfcosets`, which produces cyclotomic cosets
- `gffilter`, which filters data using GF(p) polynomials
- `gfprimfd`, which finds primitive polynomials
- `gfrank`, which computes the rank of a matrix over GF(p)
- `gfrepcov`, which converts one binary polynomial representation to another

Selected Bibliography for Galois Fields

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, Mass., Addison-Wesley, 1983.

[2] Lang, Serge, *Algebra*, Third Edition, Reading, Mass., Addison-Wesley, 1993.

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

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

Index

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

V

W