Communications Toolbox™

Reference

MATLAB®&SIMULINK®

R2020b

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
☎ Phone:	508-647-7000

✉ The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Communications Toolbox™ Reference

Trademarks

Patents

Revision History

April 2011	First printing	New for Version 5.0
September 2011	Online only	Revised for Version 5.1 (R2011b)
March 2012	Online only	Revised for Version 5.2 (R2012a)
September 2012	Online only	Revised for Version 5.3 (R2012b)
March 2013	Online only	Revised for Version 5.4 (R2013a)
September 2013	Online only	Revised for Version 5.5 (R2013b)
March 2014	Online only	Revised for Version 5.6 (R2014a)
October 2014	Online only	Revised for Version 5.7 (R2014b)
March 2015	Online only	Revised for Version 6.0 (R2015a)
September 2015	Online only	Revised for Version 6.1 (R2015b)
March 2016	Online only	Revised for Version 6.2 (R2016a)
September 2016	Online only	Revised for Version 6.3 (R2016b)
March 2017	Online only	Revised for Version 6.4 (R2017a)
September 2017	Online only	Revised for Version 6.5 (R2017b)
March 2018	Online only	Revised for Version 6.6 (Release 2018a)
September 2018	Online only	Revised for Version 7.0 (Release 2018b)
March 2019	Online only	Revised for Version 7.1 (Release 2019a)
September 2019	Online only	Revised for Version 7.2 (Release 2019b)
March 2020	Online only	Revised for Version 7.3 (Release 2020a)
September 2020	Online only	Revised for Version 7.4 (Release 2020b)

Contents

1

Apps

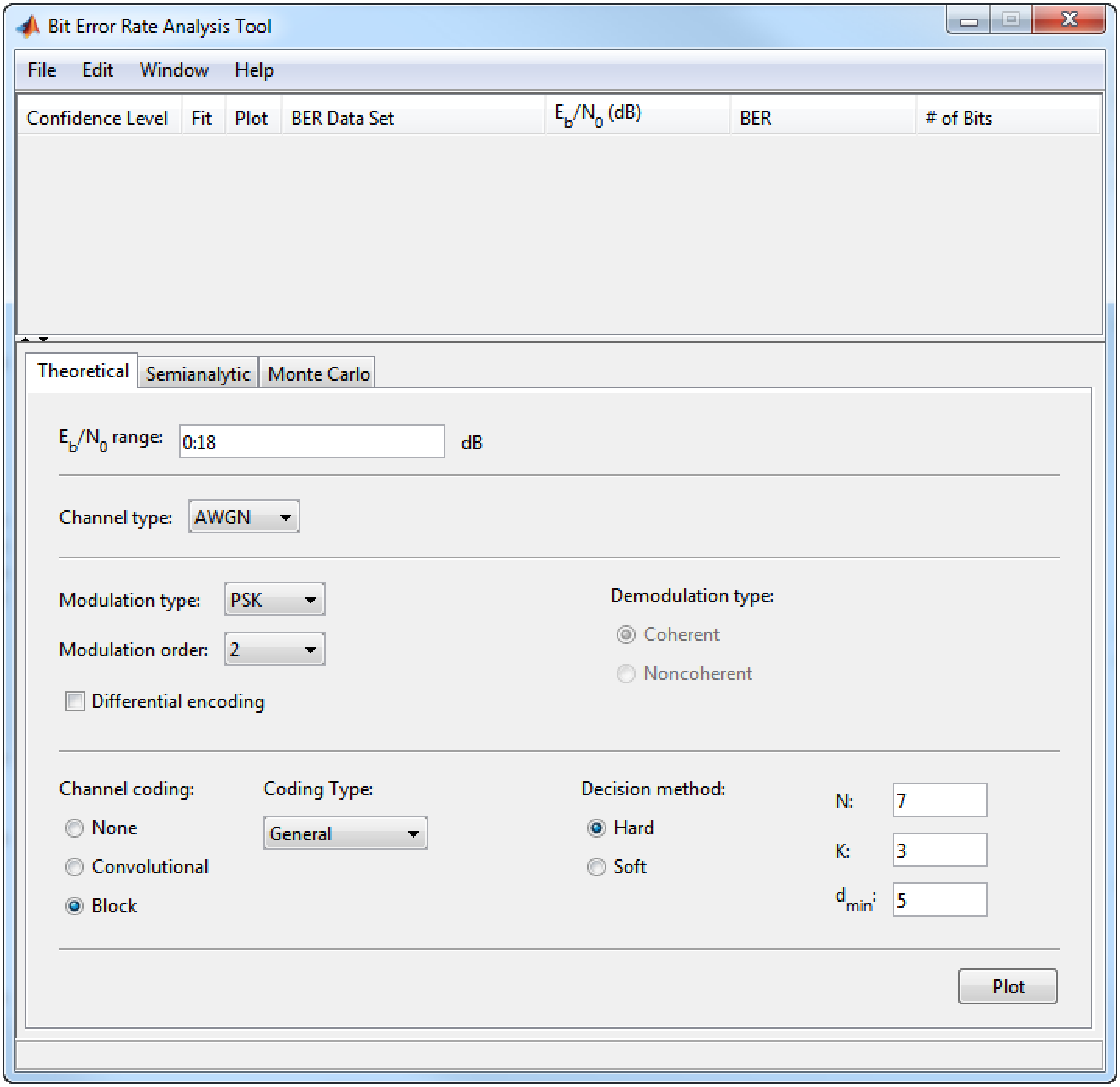
BER Analyzer

Analyze bit error rate (BER) performance of communications systems

Description

The **BER Analyzer** app calculates BER as a function of the energy per bit to noise power spectral density ratio (E_b/N_0). Using this app, you can:

- Plot theoretical BER vs. E_b/N_0 estimates and upper bounds.
- Plot BER vs. E_b/N_0 using the semianalytic technique. The semianalytic technique estimates BER performance by using a combination of simulation and analysis. Use this technique when the system error rate is small, for example, $< 10^{-6}$.
- Estimate BER performance by using MATLAB® functions or Simulink® models.



E_b/N_0 range: 0:18 dB

Channel type: AWGN

Modulation type: PSK

Modulation order: 2

□ Differential encoding

Demodulation type: ● Coherent ○ Noncoherent

Channel coding: ○ None ○ Convolutional ● Block

Coding Type: General

Decision method: ● Hard ○ Soft

N: 7

K: 3

d_{min}: 5

Open the BER Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `bertool`.

Examples

Theoretical Plot

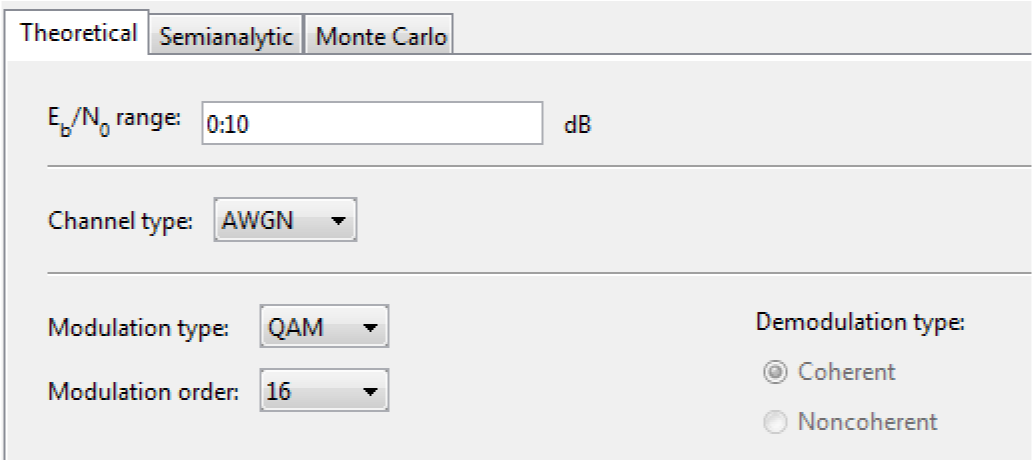
Generate a theoretical estimate of BER performance for a 16-QAM link in AWGN.

Open the **BER Analysis** app.

```
bertool
```

Specify the **E_b/N_0 range** as `0:10`.

Set **Modulation type** to `QAM` and **Modulation order** to `16`.



E_b/N_0 range: 0:10 dB

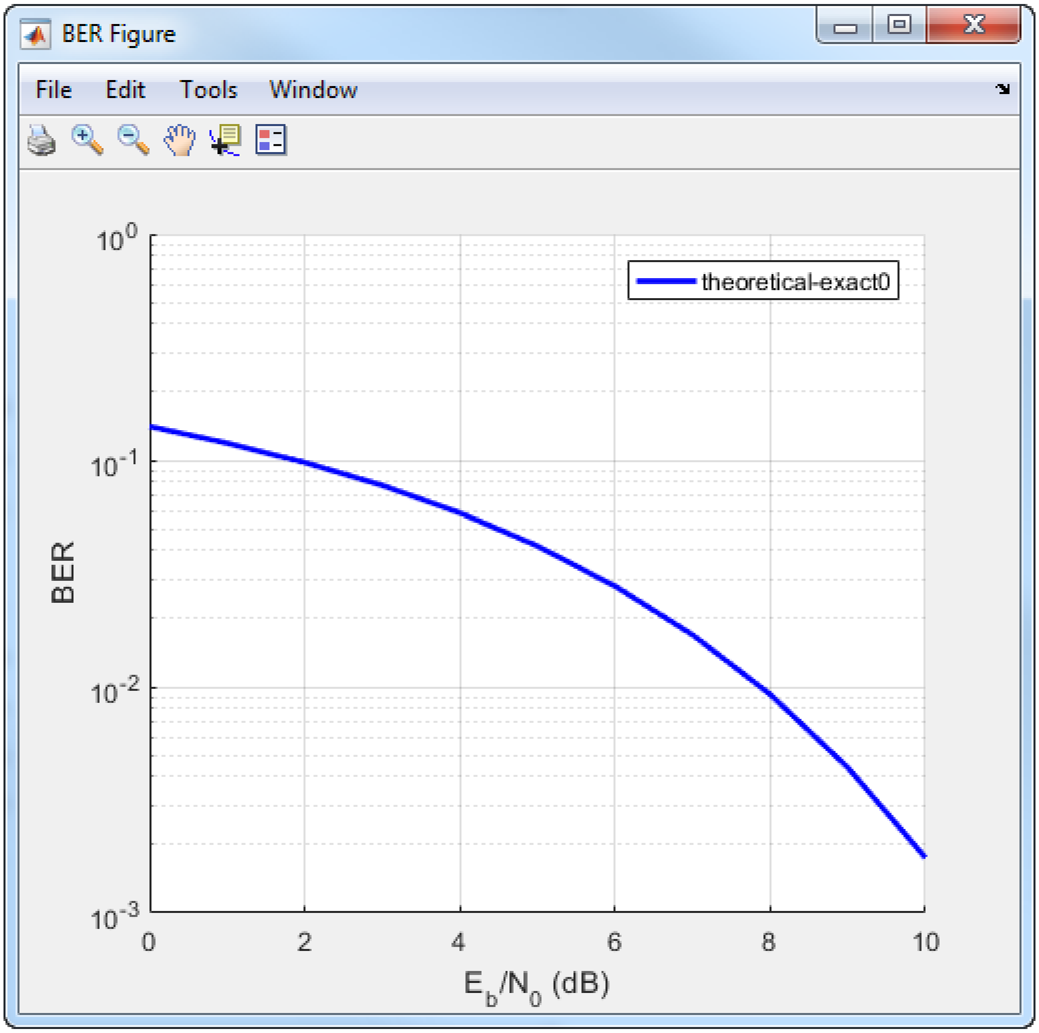
Channel type: AWGN

Modulation type: QAM

Modulation order: 16

Demodulation type:
● Coherent
○ Noncoherent

Plot the BER curve by clicking **Plot**.



Semianalytic Plot

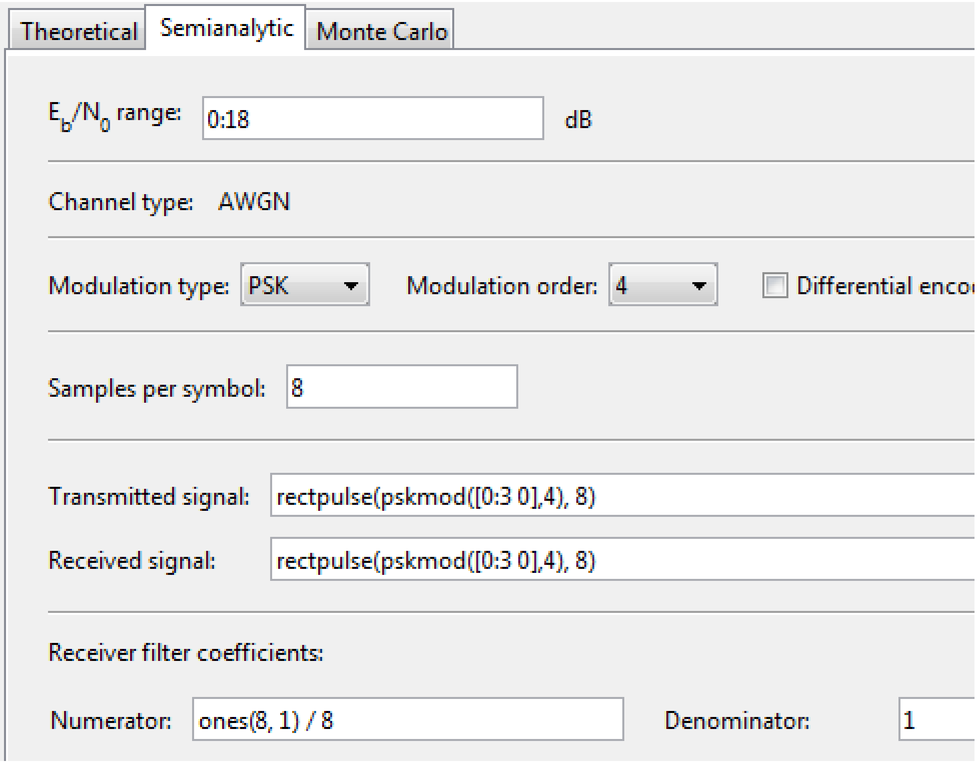
Use the semianalytic technique to plot the BER for a QPSK link having rectangular pulses.

Open the **BER Analysis** app.

```
bertool
```

On the **Semianalytic** tab, set these parameters:

- Set the **Modulation order** to `4`.
- Set the **Samples per symbol** parameter to `8`.
- Set the **Transmitted signal** and **Received signal** parameters to `rectpulse(pskmod([0:3 0],4),8)`. To use the semianalytic technique, the number of symbols must exceed M^L, where M is the modulation order and L is the impulse response length. The impulse response is 1, so a minimum of five symbols is required.
- Specify the **Numerator** as `ones(8,1)/8`. These coefficients specify an ideal integrator having eight samples per symbol.



E_b/N_0 range: 0:18 dB

Channel type: AWGN

Modulation type: PSK Modulation order: 4 □ Differential enco

Samples per symbol: 8

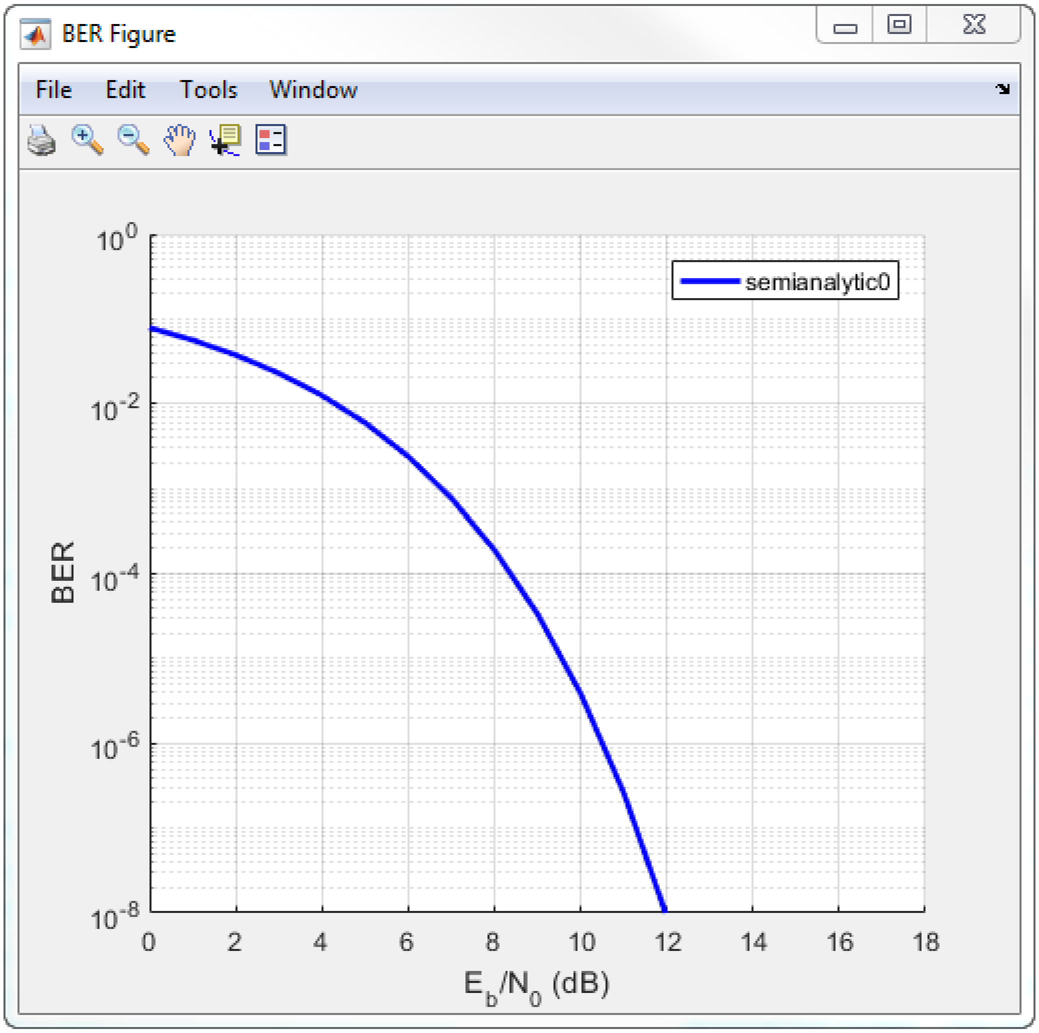
Transmitted signal: rectpulse(pskmod([0:3 0],4), 8)

Received signal: rectpulse(pskmod([0:3 0],4), 8)

Receiver filter coefficients:

Numerator: ones(8, 1) / 8 Denominator: 1

Plot the BER vs. E_b/N_0 curve by clicking **Plot**.



Monte Carlo Simulation

Simulate BER using a custom MATLAB function.

Open the **BER Analysis** app.

```
bertool
```

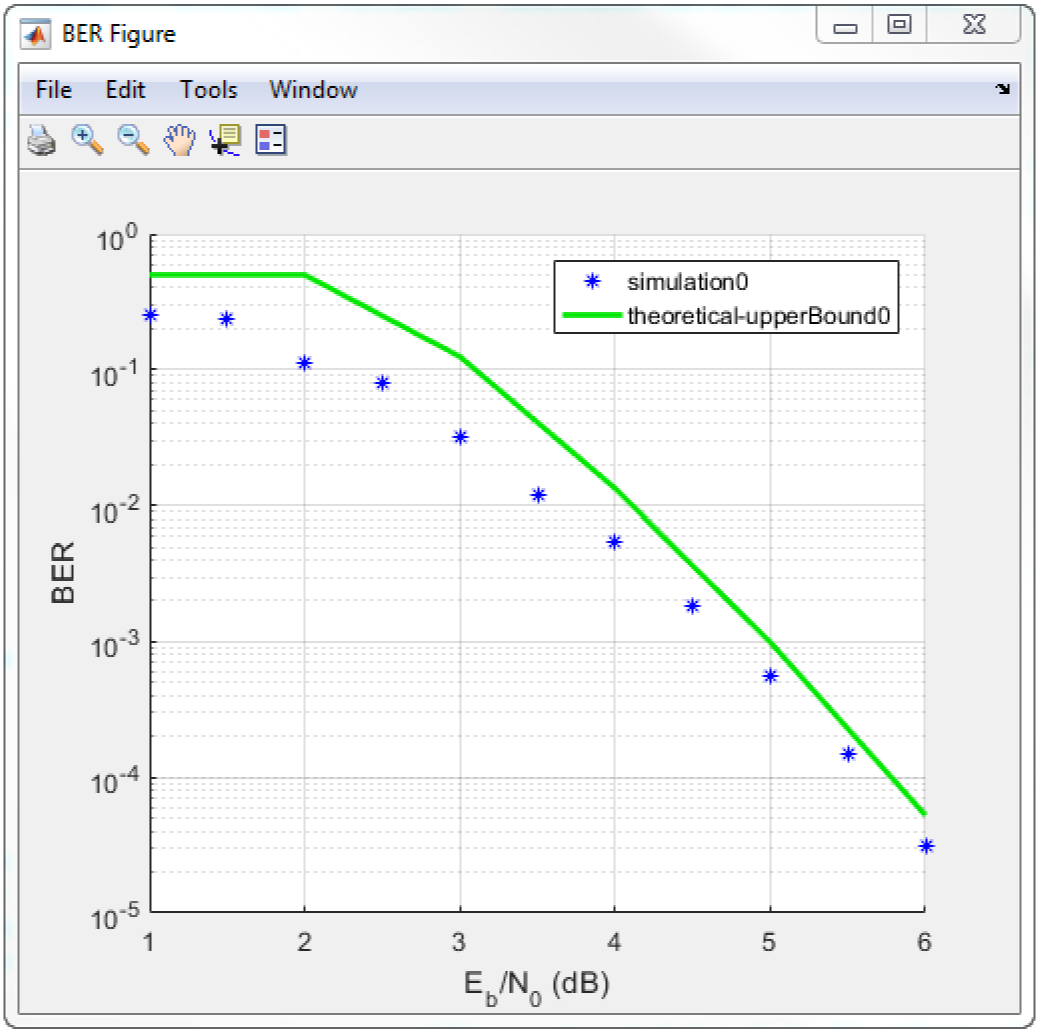
On the **Monte Carlo** tab, specify E_b/N_0 range as `1:.5:6`.

To plot estimated BER values, run the simulation by clicking **Run**.

On the **Theoretical** tab, specify E_b/N_0 range as `1:6` and set **Modulation order** to 4.

Enable convolutional coding by selecting the **Convolutional** check box.

Plot the upper bound of the BER curve by clicking **Plot**.



Parameters

Theoretical

E_b/N_0 range — Range of E_b/N_0 values over which the BER is evaluated
0:18 (default) | vector

Specifies the range of E_b/N_0 values, in dB, over which the BER is evaluated. The values in the range vector must be real.

Example: 5:10

Channel type — Type of channel over which the BER is evaluated
AWGN (default) | Rayleigh | Rician

Specifies the type of channel over which the BER is evaluated. The Rayleigh and Rician options correspond to flat fading channels.

Modulation type — Modulation type of the communication link
PSK (default) | DPSK | OQPSK | PAM | QAM | FSK | MSK | CPFSK

Specifies the modulation type of the communication link.

`Modulation order` — Modulation order of the communication link

2 (default) | 4 | 8 | 16 | 32 | 64

Specifies the modulation order of the communication link.

`Differential encoding` — Differential encoding of the input data

`Off` (default) | `On`

Specifies if the input data sequence is differentially encoded.

`Demodulation type` — Demodulation type

`Coherent` | `Noncoherent`

Specifies if `Coherent` or `Noncoherent` demodulation is used. This parameter is available only when the **Modulation type** is FSK or MSK.

`Channel coding` — Channel coding used in estimating the BER

`None` (default) | `Convolutional` | `Block`

Specifies the type of channel coding used to estimate the theoretical BER.

`Synchronization` — Synchronization error

`Perfect synchronization` (default) | `Normalized timing error` | `RMS phase noise level`

Specifies the synchronization error in the demodulation process. This parameter is available only when the **Modulation type** is PSK and the **Modulation order** is 2.

- When **Synchronization** is `Normalized timing error`, specify the normalized error as a real number from `0` to `0.5`.
- When **Synchronization** is `RMS phase noise level`, specify the RMS phase noise as a nonnegative real number.

`Decision method` — Decoding decision method

`Hard` (default) | `Soft`

Specify the method used to decode the received data. This parameter is available when either of these conditions exist:

- **Channel coding** is set to **Convolutional**
- **Channel coding** is set to **Block** and **Coding Type** is `General`

`Trellis` — Convolutional code trellis

`poly2trellis(7,[171 133])` (default) | structure

Specify the convolutional code trellis as a structure variable. You can generate this structure by using the `poly2trellis` function. The parameter is available only when the **Channel coding** parameter is `Convolutional`.

`Coding type` — Specify block coding type

`General` (default) | `Hamming` | `Golay` | `Reed-Solomon`

Specify the block code used in the BER evaluation.

`N` — Codeword length

positive integer

Specify the codeword length as a positive integer.

`K` — Message length

positive integer

Specify the message length as a positive integer such that **K** is less than **N**.

d_{min} — Minimum code distance

positive integer

Specify the minimum distance of the (N,K) block code as a positive integer. This parameter is available when the **Coding type** is `General`.

Semianalytic

`Samples per symbol` — Samples per symbol

`16` (default) | positive integer

Specify the number of samples per symbol as a positive integer.

`Transmitted signal` — Transmitted sample sequence

`rectpulse(step(comm.BPSKModulator, [0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0].'), 16)` (default) | vector

Specify the transmit sequence as a real or complex column vector.

Data Types: `double`

`Received signal` — Received sample sequence

`rectpulse(step(comm.BPSKModulator, [0 1 1 0 0 1 1 1 1 0 1 1 0 0 0 0].'), 16)` (default) | vector

Specify the received sequence as a real or complex column vector.

Data Types: `double`

`Numerator` — Numerator of the receive filter coefficients

`ones(16,1)/16` (default) | scalar | vector

Specify the numerator of the receive filter coefficients as a vector.

`Denominator` — Denominator of the receive filter coefficients

`1` (default) | scalar | vector

Specify the denominator of the receive filter coefficients as a vector.

Monte Carlo

`Simulation MATLAB file or Simulink model` — File name of the BER simulation

character vector

Specify the name of the MATLAB file or Simulink model containing the simulation code.

`BER variable name` — Name of the variable containing the BER simulation data
character vector

Specify the name of the MATLAB workspace variable that contains the BER simulation data.

`Number of errors` — Number of errors measured before simulation stop
`100` (default) | positive integer

Specify the number of errors that must be measured before the simulation stops. Typically, 100 measured errors are enough to produce an accurate BER estimate.

`Number of bits` — Number of bits processed before simulation stop
`1e8` (default) | positive integer

Specify the number of bits that must be processed before the simulation stops. This parameter is used to prevent the simulation from running too long.

Note The Monte Carlo simulation stops when either the number of errors or number of bits threshold is reached.

See Also

Functions
`berawgn` | `bercoding` | `berfading` | `berfit`

Topics
“Bit Error Rate (BER)”

Introduced before R2006a

Eye Diagram Analyzer

(Removed) Visualize and measure effects of impairments

Note **eyescope** has been removed. Use `eyediagram` instead.

Description

The **Eye Diagram Analyzer** app displays and measures the effects of various impairments. Using this app, you can:

- Visualize eye diagrams.
- Measure these quantities:
  - Horizontal and vertical eye openings
  - Random, deterministic, total, RMS, and peak-to-peak jitter
  - Rise and fall times
  - Signal-to-noise ratio
- Import and compare measurement results for eye diagrams of multiple signals.

Open the Eye Diagram Analyzer App

- MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon.
- MATLAB command prompt: Enter `eyescope`.

Programmatic Use

`eyescope` calls an empty scope.

`eyescope(obj)` calls the eye scope and displays object `obj`.

Compatibility Considerations

eyescope has been removed
Errors starting in R2020a

Eye Diagram Analyzer has been removed. Use `eyediagram` instead.

See Also

`eyediagram`

Topics
"Eye Diagram Analysis"

Introduced in R2008b

Wireless Waveform Generator

Create, impair, visualize, and export modulated waveforms

Description

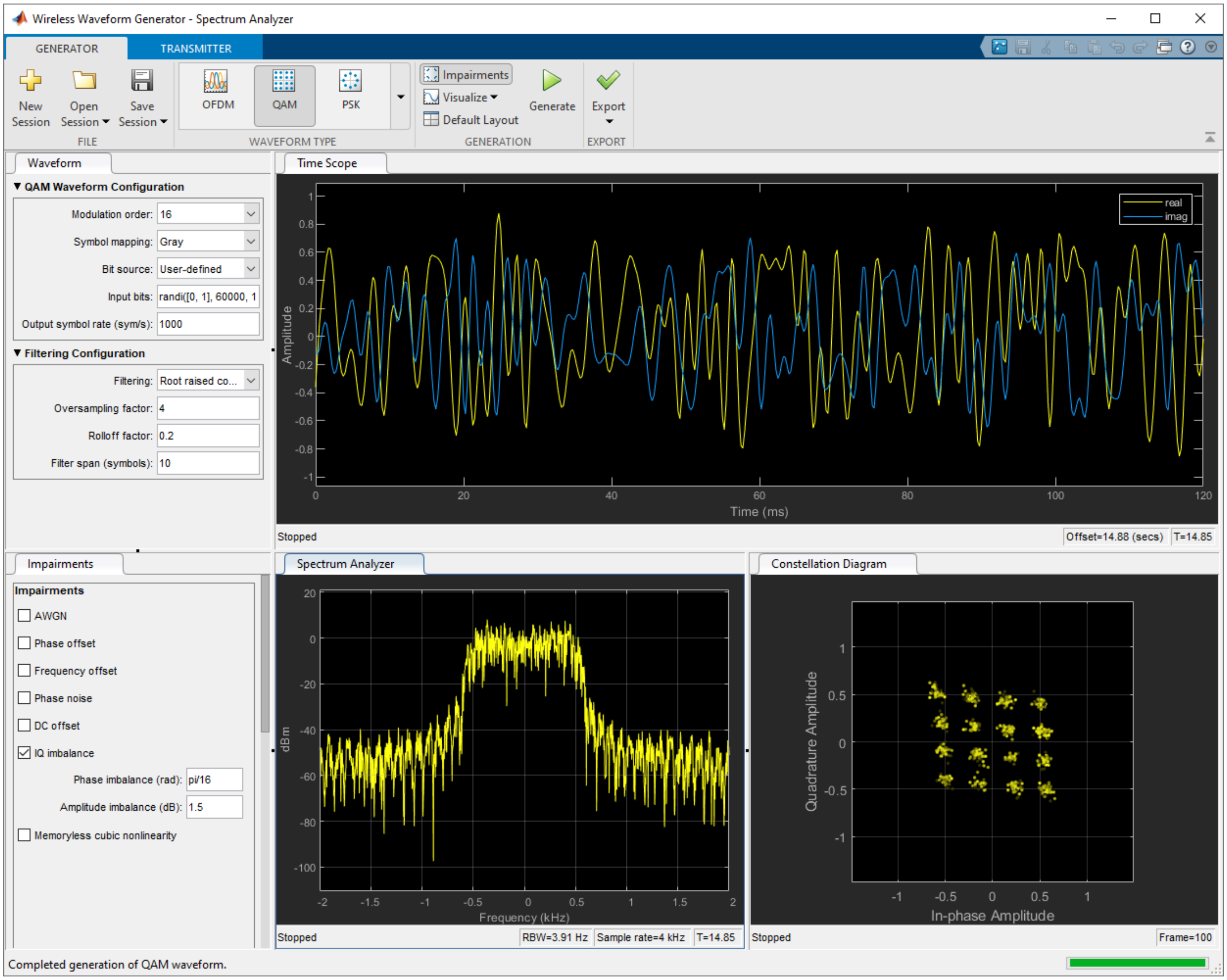
The **Wireless Waveform Generator** app enables you to create, impair, visualize, and export modulated waveforms.

Using the app, you can:

- Generate custom OFDM, QAM, and PSK modulated waveforms.
- Generate sine wave test waveforms.
- Generate NR test models (NR-TM) and NR uplink and downlink fixed reference channel (FRC) waveforms. This feature requires “5G Toolbox”.
- Generate LTE modulated waveforms. This feature requires the “LTE Toolbox”.
- Generate WLAN (802.11™) modulated waveforms. This feature requires the “WLAN Toolbox”.
- Generate Bluetooth modulated waveforms. This feature requires the Communications Toolbox Library for the Bluetooth® Protocol. To download this library, go to Get Add-Ons.
- Distort the waveform by adding RF impairments, such as AWGN, phase offset, frequency offset, DC offset, IQ imbalance, and memoryless cubic nonlinearity.
- Visualize the waveform in constellation diagram, spectrum analyzer, OFDM grid, and time scope plots.
- Export the waveform to your workspace as a structure, to a `.mat` or a `.bb` file, or to a runnable MATLAB script.

  Note You can use the MATLAB script to reproduce your waveform outside of the **Wireless Waveform Generator** app.
- Generate a waveform that you can transmit using a connected lab test instrument. The app can transmit a waveform by using instruments supported by the `rfsiggen` function. Use of the transmit feature in the app requires Instrument Control Toolbox™ software. For more information, see the documentation for “Instrument Control Toolbox”.

For more information, see “Using Wireless Waveform Generator App”.



▼ QAM Waveform Configuration

Modulation order: 16

Symbol mapping: Gray

Bit source: User-defined

Input bits: randi([0, 1], 60000, 1

Output symbol rate (sym/s): 1000

▼ Filtering Configuration

Filtering: Root raised co...

Oversampling factor: 4

Rolloff factor: 0.2

Filter span (symbols): 10

Impairments

☐ AWGN

☐ Phase offset

☐ Frequency offset

☐ Phase noise

☐ DC offset

☑ IQ imbalance

Phase imbalance (rad): pi/16

Amplitude imbalance (dB): 1.5

☐ Memoryless cubic nonlinearity

Completed generation of QAM waveform.

Open the Wireless Waveform Generator App

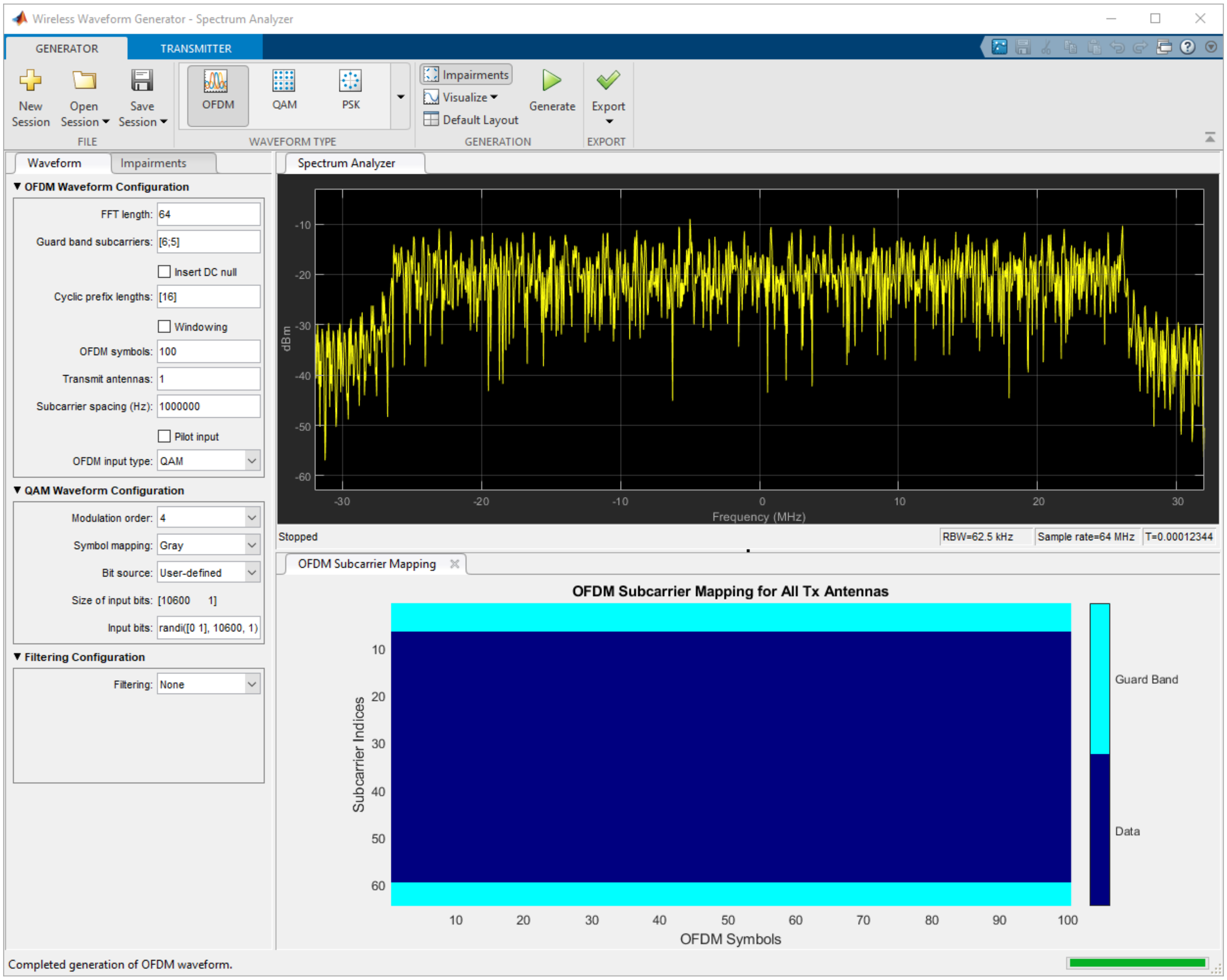
MATLAB Toolstrip: On the **Apps** tab, under **Signal Processing and Communications**, click the app icon. Wireless Waveform G...

MATLAB Command Prompt: Enter `wirelessWaveformGenerator`.

Examples

Generate OFDM Waveform

Open the **Wireless Waveform Generator** app and generate the default waveform by clicking **Generate**. The displayed waveform is an OFDM waveform with QPSK-modulated symbols.



OFDM Waveform Configuration

FFT length: 64

Guard band subcarriers: [6;5]

□ Insert DC null

Cyclic prefix lengths: [16]

□ Windowing

OFDM symbols: 100

Transmit antennas: 1

Subcarrier spacing (Hz): 1000000

□ Pilot input

OFDM input type: QAM

QAM Waveform Configuration

Modulation order: 4

Symbol mapping: Gray

Bit source: User-defined

Size of input bits: [10600 1]

Input bits: randi([0 1], 10600, 1)

Filtering Configuration

Filtering: None

Stopped

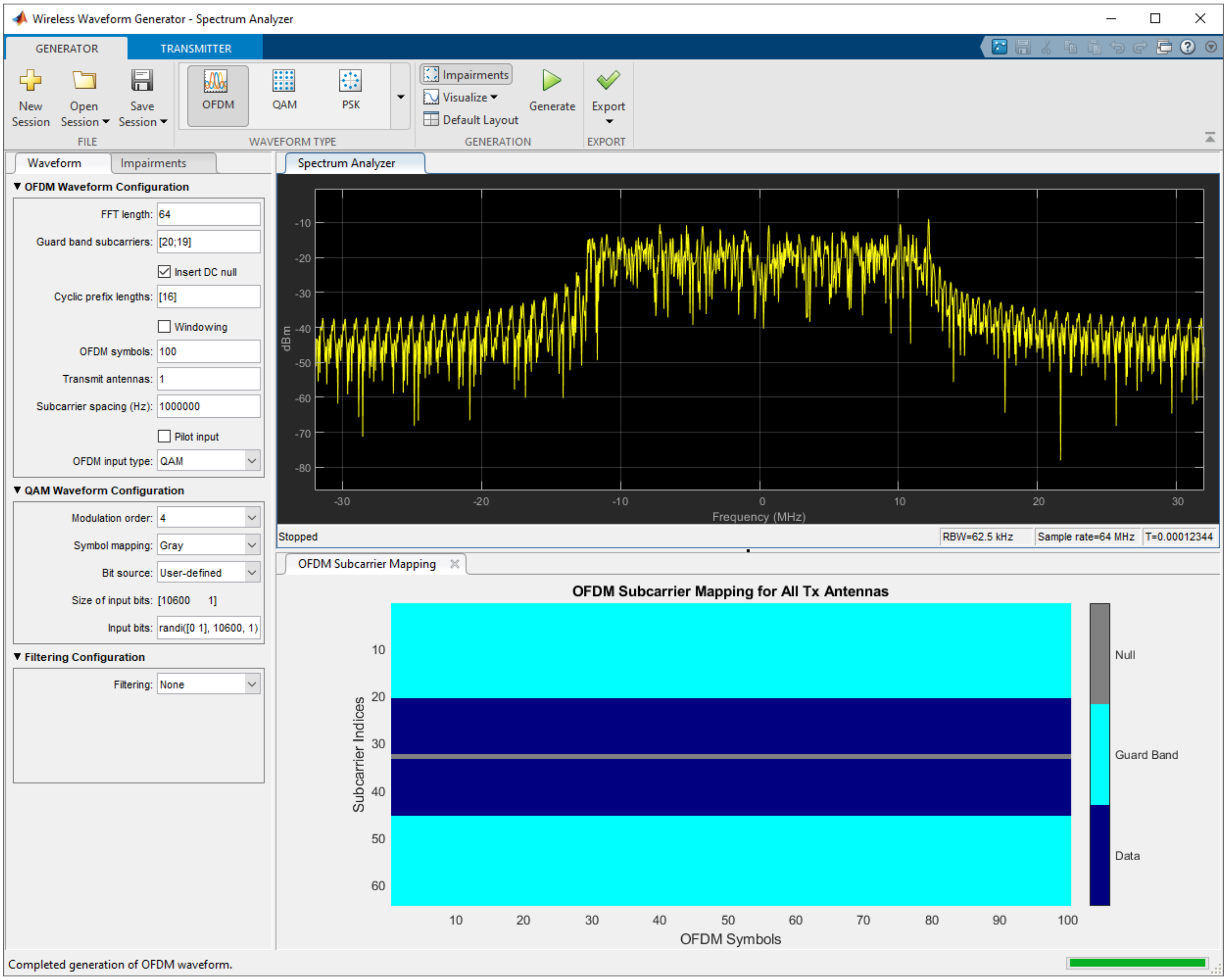
RBW=62.5 kHz

Sample rate=64 MHz

T=0.00012344

Completed generation of OFDM waveform.

Click **Insert DC null** and increase the **Guard band subcarriers** to `[20;19]`. Click **Generate** again. The plotted waveform changes to reflect the updated configuration.



▼ OFDM Waveform Configuration

FFT length: 64

Guard band subcarriers: [20;19]

☑ Insert DC null

Cyclic prefix lengths: [16]

□ Windowing

OFDM symbols: 100

Transmit antennas: 1

Subcarrier spacing (Hz): 1000000

□ Pilot input

OFDM input type: QAM

▼ QAM Waveform Configuration

Modulation order: 4

Symbol mapping: Gray

Bit source: User-defined

Size of input bits: [10600 1]

Input bits: randi([0 1], 10600, 1)

▼ Filtering Configuration

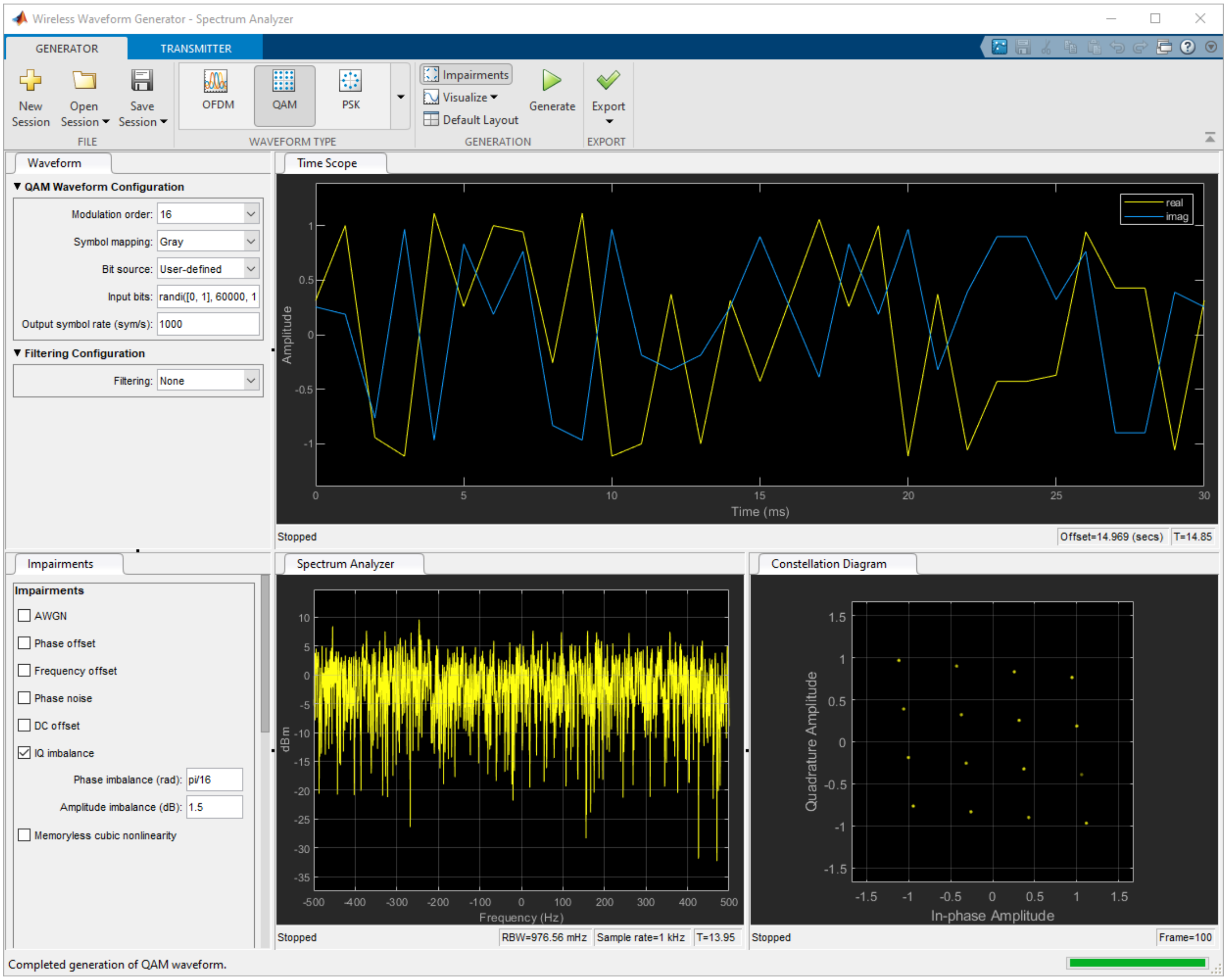
Filtering: None

Stopped | RBW=62.5 kHz | Sample rate=64 MHz | T=0.00012344

Completed generation of OFDM waveform.

Generate 16-QAM Waveform with Impairments

Open the **Wireless Waveform Generator** app and configure a 16-QAM waveform. Specify a phase imbalance of 11.25 degrees (`pi/16` radians) and an amplitude imbalance of 1.5 dB. Click **Generate** to generate the waveform.



▼ QAM Waveform Configuration

Modulation order: 16

Symbol mapping: Gray

Bit source: User-defined

Input bits: randi([0, 1], 60000, 1

Output symbol rate (sym/s): 1000

▼ Filtering Configuration

Filtering: None

Impairments

□ AWGN

□ Phase offset

□ Frequency offset

□ Phase noise

□ DC offset

☑ IQ imbalance

Phase imbalance (rad): pi/16

Amplitude imbalance (dB): 1.5

□ Memoryless cubic nonlinearity

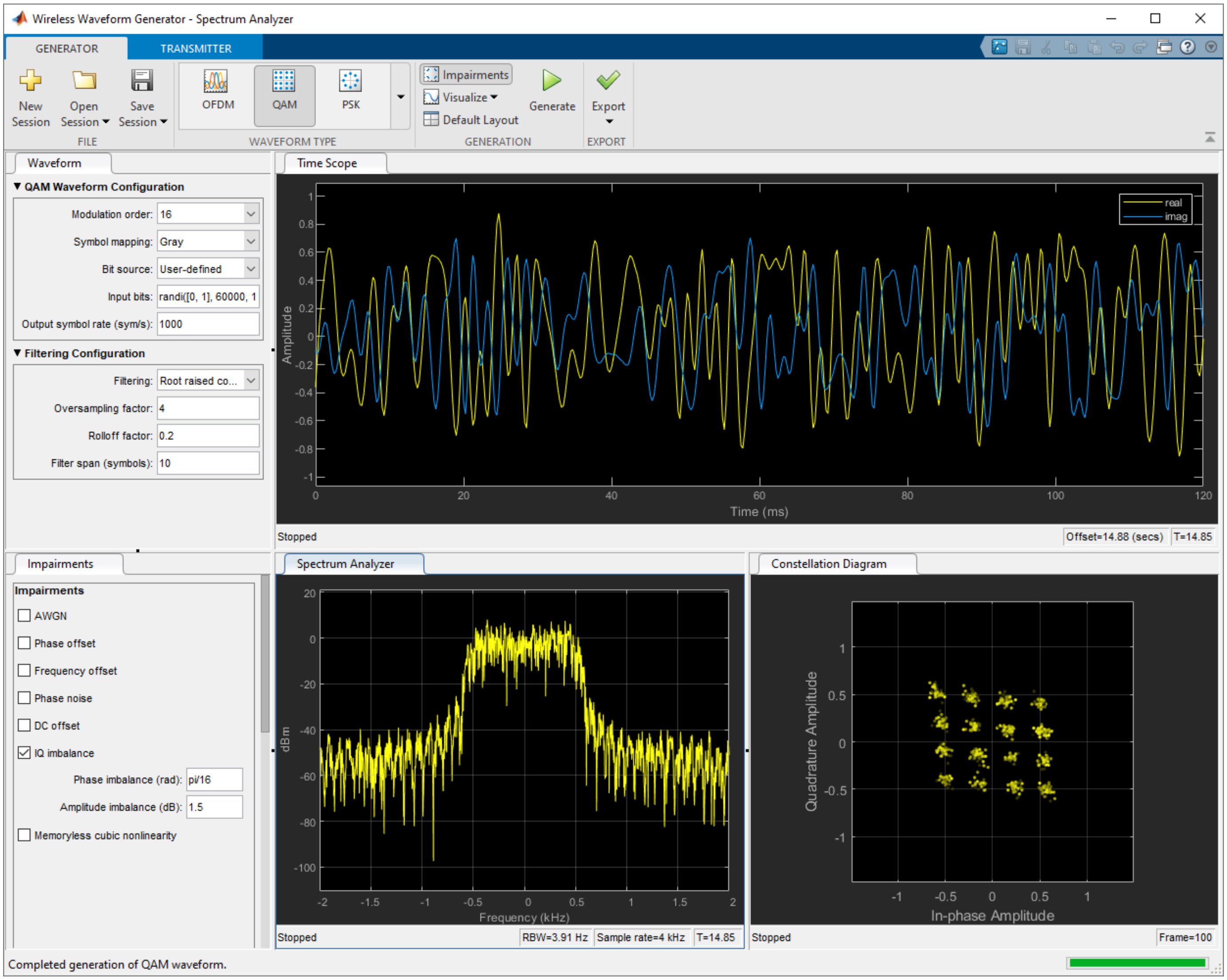
Stopped Offset=14.969 (secs) T=14.85

Stopped RBW=976.56 mHz Sample rate=1 kHz T=13.95

Stopped Frame=100

Completed generation of QAM waveform.

Select the **Filtering** parameter and apply root raised cosine filtering. Click **Generate** again to generate a waveform using the current configuration. The plotted waveform changes to reflect the updated configuration.



▼ QAM Waveform Configuration

Modulation order: 16

Symbol mapping: Gray

Bit source: User-defined

Input bits: randi([0, 1], 60000, 1

Output symbol rate (sym/s): 1000

▼ Filtering Configuration

Filtering: Root raised co...

Oversampling factor: 4

Rolloff factor: 0.2

Filter span (symbols): 10

Impairments

□ AWGN

□ Phase offset

□ Frequency offset

□ Phase noise

□ DC offset

☑ IQ imbalance

Phase imbalance (rad): pi/16

Amplitude imbalance (dB): 1.5

□ Memoryless cubic nonlinearity

Completed generation of QAM waveform.

App-Based 5G NR-TM and FRC Waveform Generation

This example shows how to use the **5G Waveform Generator** app to generate NR test models (NR-TM) and NR uplink and downlink fixed reference channel (FRC) waveforms.

Open 5G Waveform Generator App

On the **Apps** tab of the MATLAB® toolstrip, under **Signal Processing and Communications**, click the **5G Waveform Generator** app icon. This app opens the **Wireless Waveform Generator** app configured for 5G waveform generation.

Select 5G NR Waveform

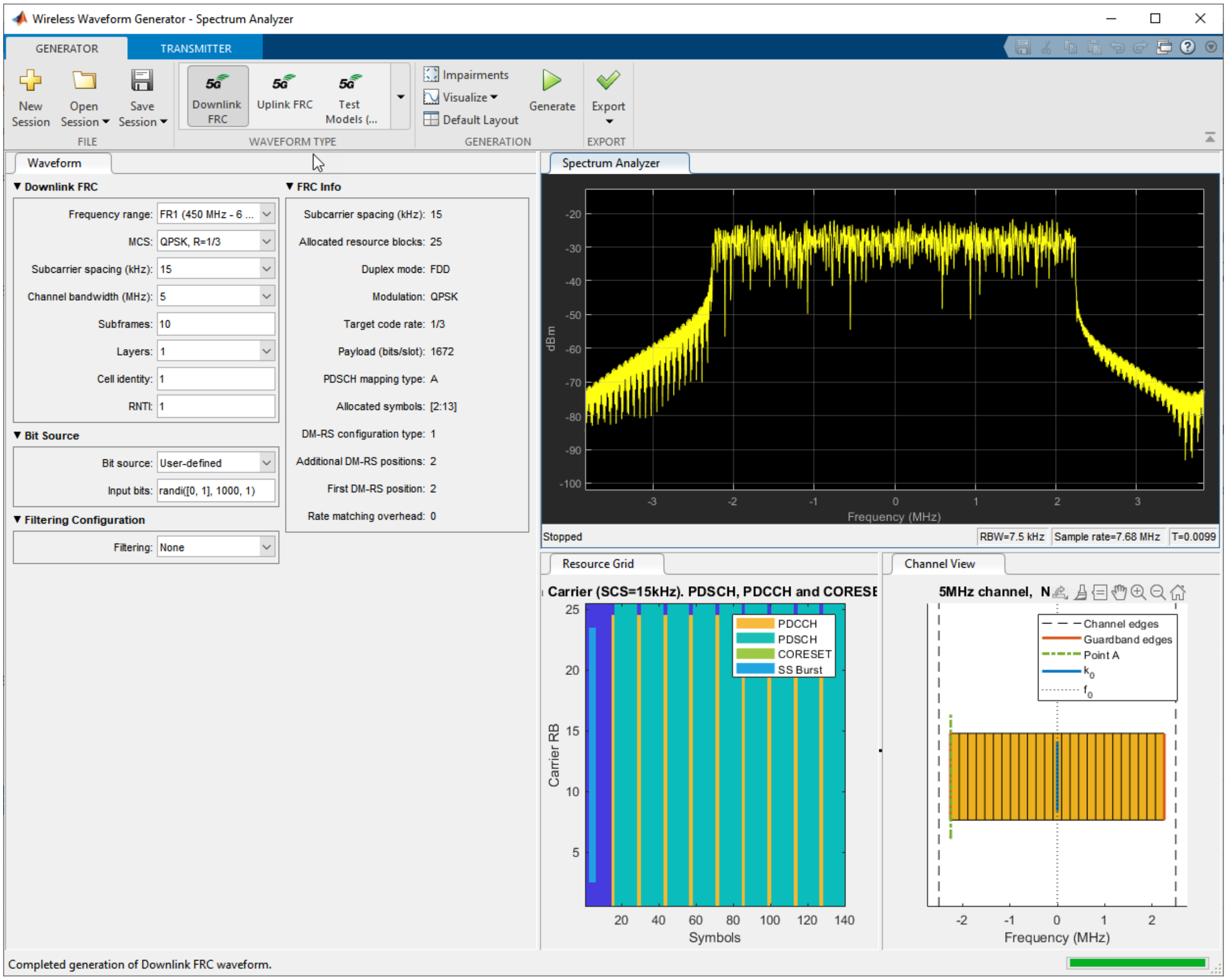
In the **Waveform Type** section on the app toolstrip, click the waveform you want to generate. You can select any of these waveforms.

- `5G Downlink FRC`
- `5G Uplink FRC`
- `5G Test Models`

Generate 5G NR Waveform

In the left pane of the app, on the **Waveform** tab, you can set the parameters of the selected waveform. On the app toolstrip, in the **Generation** section, you can add impairments and set visualization tools. To visualize the waveform, click **Generate**. You can export the generated waveform to the MATLAB workspace as a structure in a `.mat` or `.bb` file.

For example, this picture shows the visualization results of a downlink FRC waveform using default parameters.



▼ Downlink FRC

Frequency range:	FR1 (450 MHz - 6 ...
MCS:	QPSK, R=1/3
Subcarrier spacing (kHz):	15
Channel bandwidth (MHz):	5
Subframes:	10
Layers:	1
Cell identity:	1
RNTI:	1

▼ Bit Source

Bit source:	User-defined
Input bits:	randi([0, 1], 1000, 1)

▼ Filtering Configuration

Filtering:	None

▼ FRC Info

Subcarrier spacing (kHz):	15
Allocated resource blocks:	25
Duplex mode:	FDD
Modulation:	QPSK
Target code rate:	1/3
Payload (bits/slot):	1672
PDSCH mapping type:	A
Allocated symbols:	[2:13]
DM-RS configuration type:	1
Additional DM-RS positions:	2
First DM-RS position:	2
Rate matching overhead:	0

Stopped RBW=7.5 kHz Sample rate=7.68 MHz T=0.0099

Completed generation of Downlink FRC waveform.

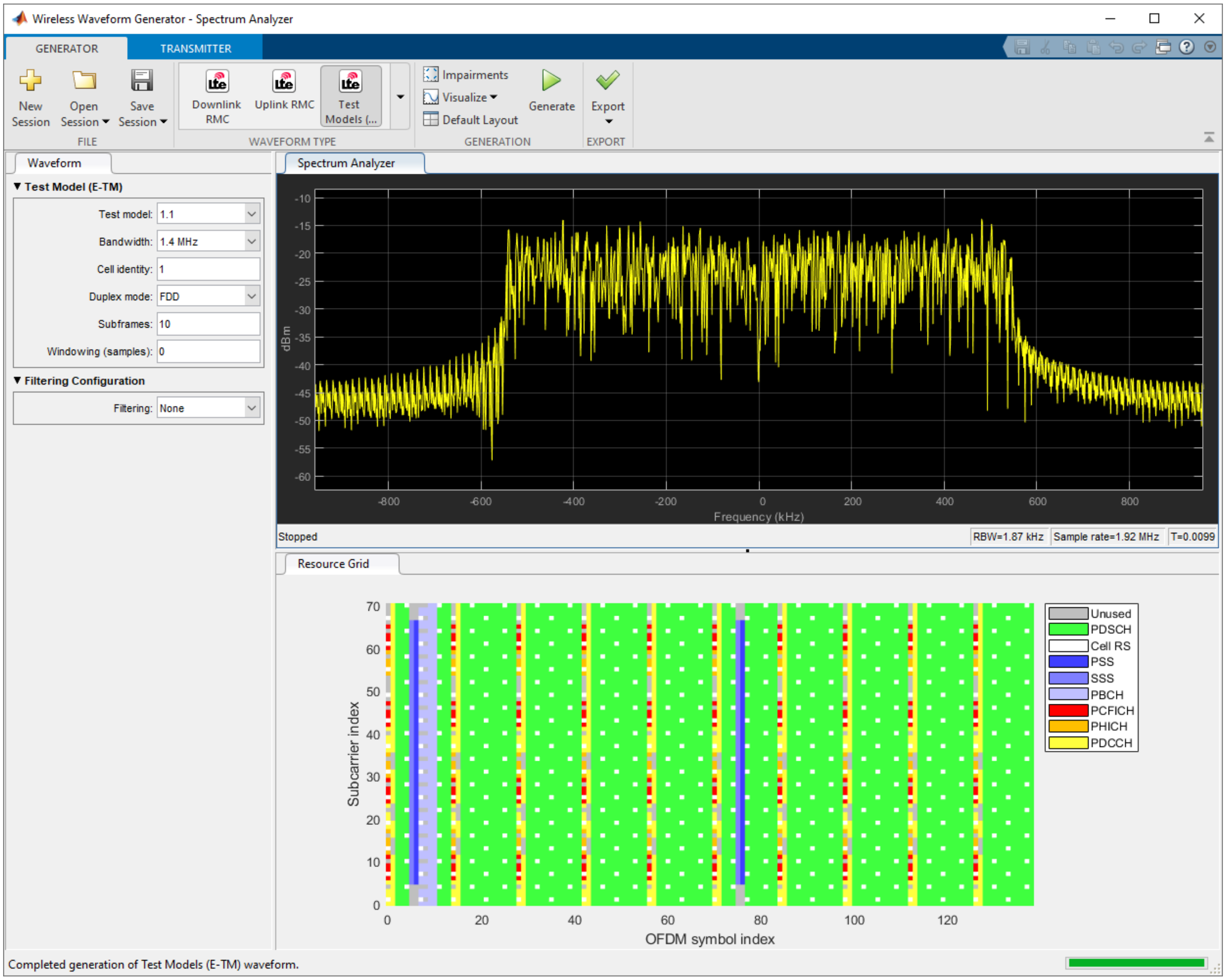
Transmit 5G NR Waveform

This feature requires “Instrument Control Toolbox”. To transmit the generated waveform, on the app toolstrip, click on the **Transmitter** tab and set up the instruments. You can use all the instruments supported by the `rfsiggen` (Instrument Control Toolbox) function.

Generate LTE E-TM Test Model Waveforms

Open the **Wireless Waveform Generator** app and configure to generate LTE E-TM Test Model waveforms. Using this feature of the app requires the LTE Toolbox™.

In the **Waveform Type** section, select `Test Model (E-TM) waveforms, specified in TS 36.141, Sec. 6`. To display an LTE E-TM Test Model 1.1 frame with a 1.4 MHz channel bandwidth, click **Generate**.



Wireless Waveform Generator - Spectrum Analyzer

Test Model (E-TM)

Test model: 1.1

Bandwidth: 1.4 MHz

Cell identity: 1

Duplex mode: FDD

Subframes: 10

Windowing (samples): 0

Filtering Configuration

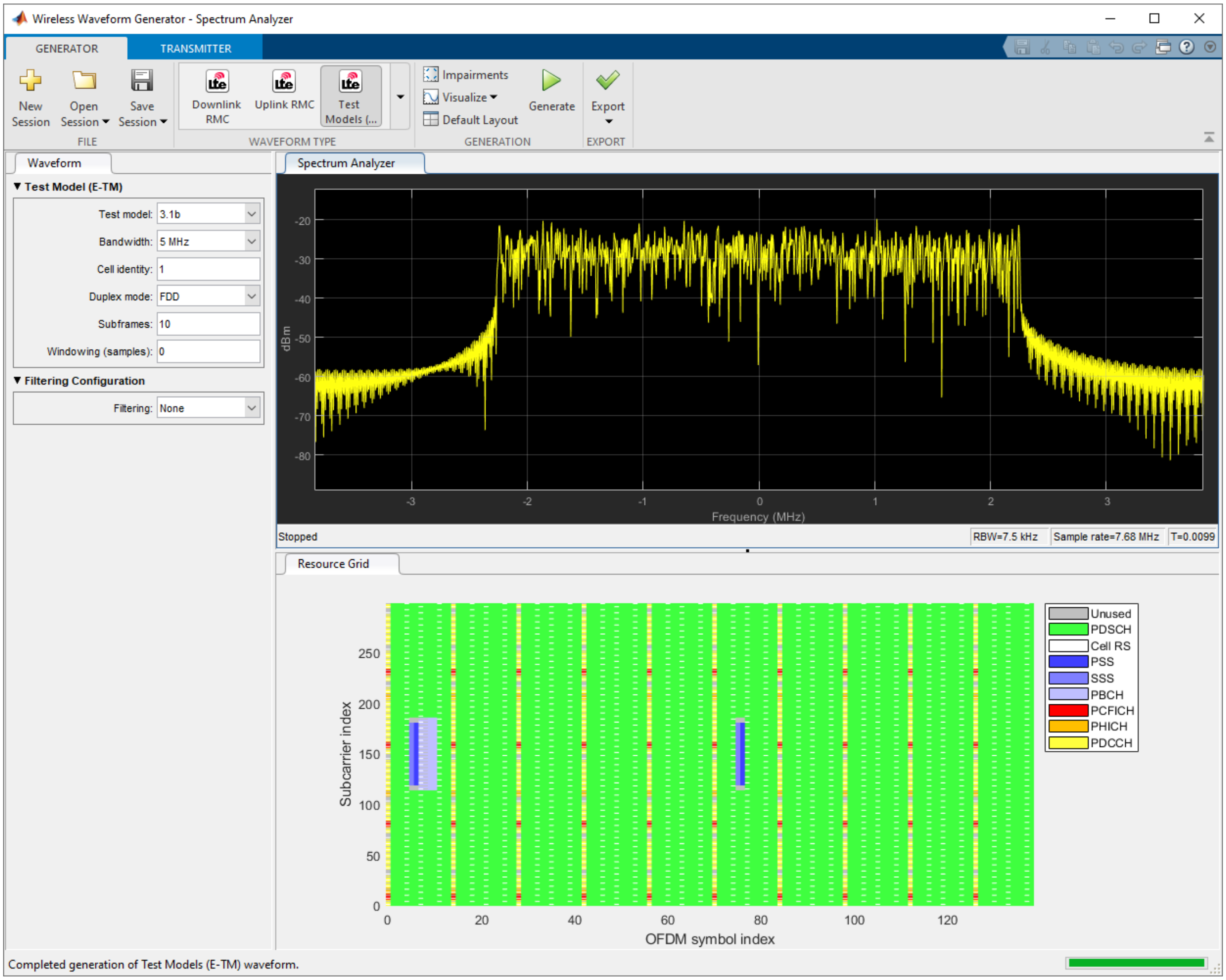
Filtering: None

Stopped

RBW=1.87 kHz | Sample rate=1.92 MHz | T=0.0099

Completed generation of Test Models (E-TM) waveform.

Configure the app to generate an LTE E-TM Test Model 3.1b frame with a channel bandwidth of 5 MHz. To visualize the waveform, click **Generate**.



▼ Test Model (E-TM)

Test model: 3.1b

Bandwidth: 5 MHz

Cell identity: 1

Duplex mode: FDD

Subframes: 10

Windowing (samples): 0

▼ Filtering Configuration

Filtering: None

Stopped

RBW=7.5 kHz

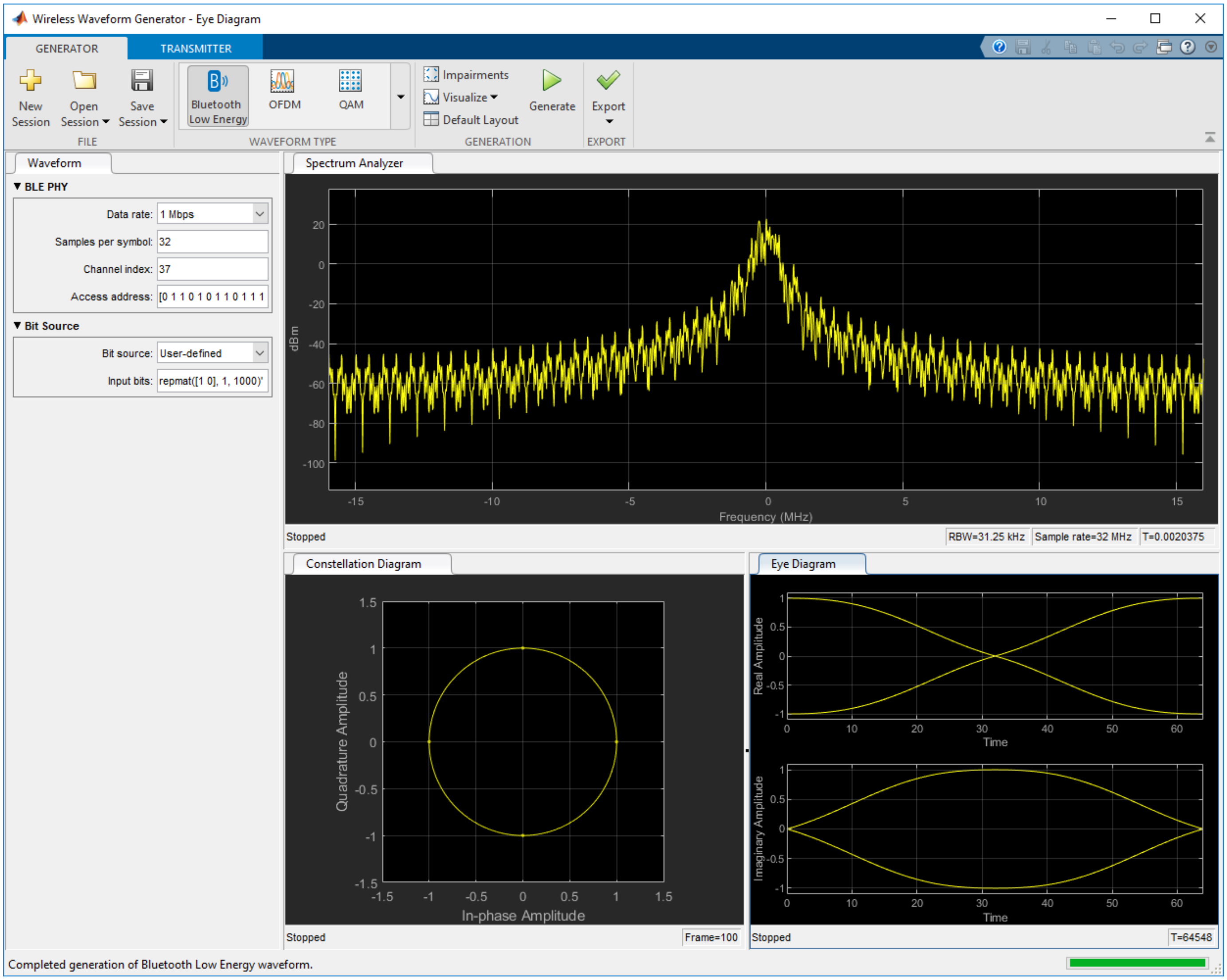
Sample rate=7.68 MHz

T=0.0099

Completed generation of Test Models (E-TM) waveform.

Generate BLE Waveform

Open the **Wireless Waveform Generator** app and configure a Bluetooth low energy (BLE) waveform. To use the BLE feature, the app requires Communications Toolbox Library for the Bluetooth Protocol . In the **Waveform Type** section, click `Bluetooth Low Energy`. Change the bit source to `User-defined` and enter the input bits. Specify the value of **Samples per symbol** as 32. Click **Generate** to generate the BLE waveform. The plotted waveform is a BLE waveform with GMSK-modulated samples. Change the visible plots by clicking **Visualize** and then selecting one or more options between `Constellation Diagram`, `Time Scope` and `Eye Diagram`.



▼ BLE PHY

Data rate: 1 Mbps

Samples per symbol: 32

Channel index: 37

Access address: [0 1 1 0 1 0 1 1 0 1 1 1

▼ Bit Source

Bit source: User-defined

Input bits: repmat([1 0], 1, 1000)'

Stopped RBW=31.25 kHz Sample rate=32 MHz T=0.0020375

Stopped Frame=100

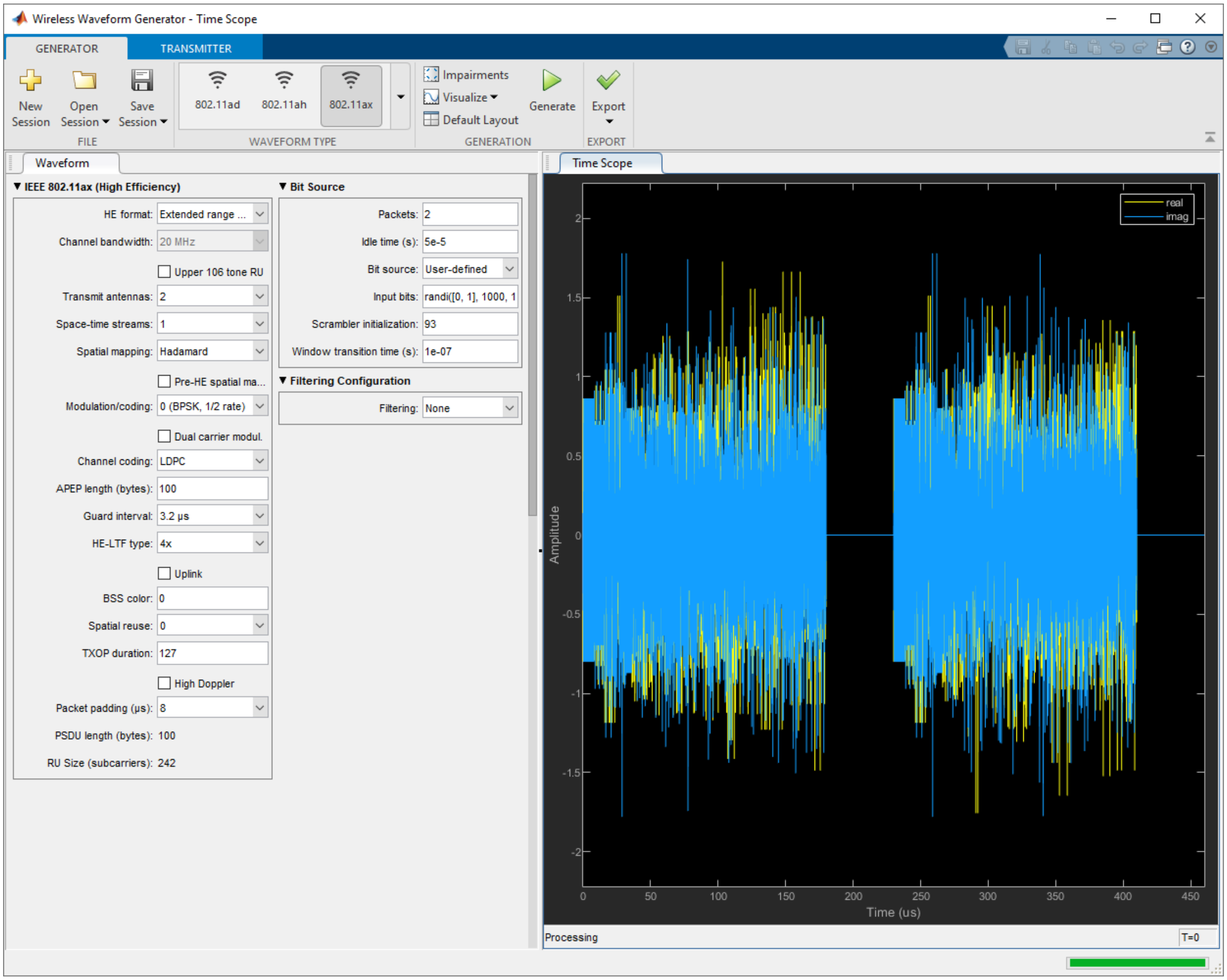
Stopped T=64548

Completed generation of Bluetooth Low Energy waveform.

Generate WLAN HE ER SU PPDU with Packet Extension

Open **Wireless Waveform Generator** and configure an HE extended-range (ER) SU PPDU. Using this feature of the app requires the WLAN Toolbox™.

In the **Waveform Type** section, select `802.11ax`. Configure the app to generate an HE ER SU PPDU with two packets, specifying a delay of 50 microseconds between packets. Specify two transmit antennas, and a nominal packet padding of eight microseconds. Click **Visualize** and configure to select `Time Scope` only. To generate the waveform, click **Generate**.



▼ IEEE 802.11ax (High Efficiency)

HE format: Extended range ...

Channel bandwidth: 20 MHz

□ Upper 106 tone RU

Transmit antennas: 2

Space-time streams: 1

Spatial mapping: Hadamard

□ Pre-HE spatial ma...

Modulation/coding: 0 (BPSK, 1/2 rate)

□ Dual carrier modul.

Channel coding: LDPC

APEP length (bytes): 100

Guard interval: 3.2 µs

HE-LTF type: 4x

□ Uplink

BSS color: 0

Spatial reuse: 0

TXOP duration: 127

□ High Doppler

Packet padding (µs): 8

PSDU length (bytes): 100

RU Size (subcarriers): 242

▼ Bit Source

Packets: 2

Idle time (s): 5e-5

Bit source: User-defined

Input bits: randi([0, 1], 1000, 1

Scrambler initialization: 93

Window transition time (s): 1e-07

▼ Filtering Configuration

Filtering: None

- “Using Wireless Waveform Generator App”

See Also

Apps
BER Analyzer

Topics
“Using Wireless Waveform Generator App”

Introduced in R2018b

2

Functions

algdeintrlv

Restore ordering of symbols using algebraically derived permutation table

Syntax

```
deintrlvd = algdeintrlv(data,num,'takeshita-costello',k,h)
deintrlvd = algdeintrlv(data,num,'welch-costas',alph)
```

Description

`deintrlvd = algdeintrlv(data,num,'takeshita-costello',k,h)` restores the original ordering of the elements in `data` using a permutation table that is algebraically derived using the Takeshita-Costello method. `num` is the number of elements in `data` if `data` is a vector, or the number of rows of `data` if `data` is a matrix with multiple columns. In the Takeshita-Costello method, `num` must be a power of 2. The multiplicative factor, `k`, must be an odd integer less than `num`, and the cyclic shift, `h`, must be a nonnegative integer less than `num`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`deintrlvd = algdeintrlv(data,num,'welch-costas',alph)` uses the Welch-Costas method. In the Welch-Costas method, `num+1` must be a prime number. `alph` is an integer between 1 and `num` that represents a primitive element of the finite field GF(`num+1`).

To use this function as an inverse of the `algintrlv` function, use the same inputs in both functions, except for the `data` input. In that case, the two functions are inverses in the sense that applying `algintrlv` followed by `algdeintrlv` leaves data unchanged.

Examples

Interleave and Deinterleave Symbols

This example uses the Takeshita-Costello method of `algintrlv` and `algdeintrlv`.

Generate random data symbols to interleave. The number of rows of input data, `num`, must be a power of two.

```
num = 16;
ncols = 3;
data = rand(num,ncols)
```

```
data = 16×3

    0.8147    0.4218    0.2769
    0.9058    0.9157    0.0462
    0.1270    0.7922    0.0971
    0.9134    0.9595    0.8235
    0.6324    0.6557    0.6948
    0.0975    0.0357    0.3171
    0.2785    0.8491    0.9502
    0.5469    0.9340    0.0344
    0.9575    0.6787    0.4387
    0.9649    0.7577    0.3816
      ⋮
```

Interleave the symbols using the Takeshita-Costello method. Set the multiplicative factor, `k`, to an odd integer less than `num`, and the cyclic shift, `h`, to a nonnegative integer less than `num`.

```
k = 3;
h = 4;
intdata = algintrlv(data,num,'takeshita-costello',k,h)
```

```
intdata = 16×3

    0.9572    0.6555    0.1869
    0.2785    0.8491    0.9502
    0.1576    0.7431    0.7655
    0.0975    0.0357    0.3171
    0.8147    0.4218    0.2769
    0.1270    0.7922    0.0971
    0.9058    0.9157    0.0462
    0.9575    0.6787    0.4387
    0.5469    0.9340    0.0344
    0.1419    0.0318    0.6463
      ⋮
```

Deinterleave the symbols to obtain the original order.

```
deintdata = algdeintrlv(intdata,num,'takeshita-costello',k,h)
```

```
deintdata = 16×3

    0.8147    0.4218    0.2769
    0.9058    0.9157    0.0462
    0.1270    0.7922    0.0971
    0.9134    0.9595    0.8235
    0.6324    0.6557    0.6948
    0.0975    0.0357    0.3171
    0.2785    0.8491    0.9502
    0.5469    0.9340    0.0344
    0.9575    0.6787    0.4387
    0.9649    0.7577    0.3816
      ⋮
```

References

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

[2] Takeshita, O. Y., and D. J. Costello, Jr., “New Classes Of Algebraic Interleavers for Turbo-Codes,” *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16–21, 1998. p. 419.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`algintrlv`

Topics
“Interleaving”

Introduced before R2006a

algintrlv

Reorder symbols using algebraically derived permutation table

Syntax

```
intrlvd = algintrlv(data,num,'takeshita-costello',k,h)
intrlvd = algintrlv(data,num,'welch-costas',alph)
```

Description

`intrlvd = algintrlv(data,num,'takeshita-costello',k,h)` rearranges the elements in `data` using a permutation table that is algebraically derived using the Takeshita-Costello method. `num` is the number of elements in `data` if `data` is a vector, or the number of rows of `data` if `data` is a matrix with multiple columns. In the Takeshita-Costello method, `num` must be a power of 2. The multiplicative factor, `k`, must be an odd integer less than `num`, and the cyclic shift, `h`, must be a nonnegative integer less than `num`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`intrlvd = algintrlv(data,num,'welch-costas',alph)` uses the Welch-Costas method. In the Welch-Costas method, `num+1` must be a prime number. `alph` is an integer between 1 and `num` that represents a primitive element of the finite field GF(`num+1`). This means that every nonzero element of GF(`num+1`) can be expressed as `alph` raised to some integer power.

Examples

Reorder Symbols Using Algebraically Derived Permutation Table

This example illustrates how to use the Welch-Costas method of algebraic interleaving.

Define `num` such that `num+1` is prime. Create data to interleave.

```
num = 10;
ncols = 3; % Number of columns of data to interleave
data = randi([0 num-1],num,ncols); % Random data to interleave
```

Find primitive polynomials of the finite field GF(num+1). The `gfprimfd` function represents each primitive polynomial as a row containing the coefficients in order of ascending powers.

```
pr = gfprimfd(1,'all',num+1)
```

```
pr = 4×2

     3     1
     4     1
     5     1
     9     1
```

Notice from the output that `pr` has two columns and that the second column consists solely of 1s. In other words, each primitive polynomial is a monic degree-one polynomial. This is because `num+1` is prime. As a result, to find the primitive element that is a root of each primitive polynomial, find a root of the polynomial by subtracting the first column of pr from num+1 .

```
primel = (num+1)-pr(:,1) % Primitive elements of GF(num+1)
```

```
primel = 4×1

     8
     7
     6
     2
```

Now define alph as one of the elements of primel and use algintrlv to interleave.

```
alph = primel(1);
intrlvd = algintrlv(data,num,'Welch-Costas',alph);
```

Algorithms

- A Takeshita-Costello interleaver uses a length-num cycle vector whose nth element is mod(k*(n-1)*n/2, num) for integers n between 1 and num. The function creates a permutation vector by listing, for each element of the cycle vector in ascending order, one plus the element's successor. The interleaver's actual permutation table is the result of shifting the elements of the permutation vector left by h. (The function performs all computations on numbers and indices modulo num.)
- A Welch-Costas interleaver uses a permutation that maps an integer K to mod(A^K,num+1)-1.

References

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

[2] Takeshita, O. Y., and D. J. Costello, Jr., “New Classes Of Algebraic Interleavers for Turbo-Codes,” *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16–21, 1998. p. 419.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

algdeintrlv

Topics

“Interleaving”

Introduced before R2006a

amdemod

Amplitude demodulation

Syntax

```
z = amdemod(y,Fc,Fs)
z = amdemod(y,Fc,Fs,ini_phase)
z = amdemod(y,Fc,Fs,ini_phase,carramp)
z = amdemod(y,Fc,Fs,ini_phase,carramp,num,den)
```

Description

`z = amdemod(y,Fc,Fs)` returns a demodulated signal `z`, given the input amplitude modulated (AM) signal `y`, where the carrier signal has frequency `Fc`. The carrier signal and `y` have sampling frequency `Fs`. The modulated signal `y` has zero initial phase and zero carrier amplitude, resulting from a suppressed-carrier modulation.

Note The value of `Fs` must satisfy `Fs` ≥ `2Fc`.

`z = amdemod(y,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = amdemod(y,Fc,Fs,ini_phase,carramp)` demodulates a signal created through transmitted-carrier modulation instead of suppressed-carrier modulation, where `carramp` is the carrier amplitude of the modulated signal.

`z = amdemod(y,Fc,Fs,ini_phase,carramp,num,den)` specifies the numerator and denominator of the lowpass Butterworth filter used in the demodulation. The numerator and denominator are generated by [num,den] = `butter(n,Fc*2/Fs)`, where `n` is the order of the lowpass filter.

Examples

Demodulate AM Signal

Set the carrier frequency to 10 kHz and sampling frequency to 80 kHz. Generate a time vector having a duration of 0.01 s.

```
fc = 10e3;
fs = 80e3;
t = (0:1/fs:0.01)';
```

Create a two-tone sinusoidal signal with frequencies 300 and 600 Hz.

```
s = sin(2*pi*300*t)+2*sin(2*pi*600*t);
```

Create a lowpass filter.

```
[num,den] = butter(10,fc*2/fs);
```

Amplitude modulate the signal s.

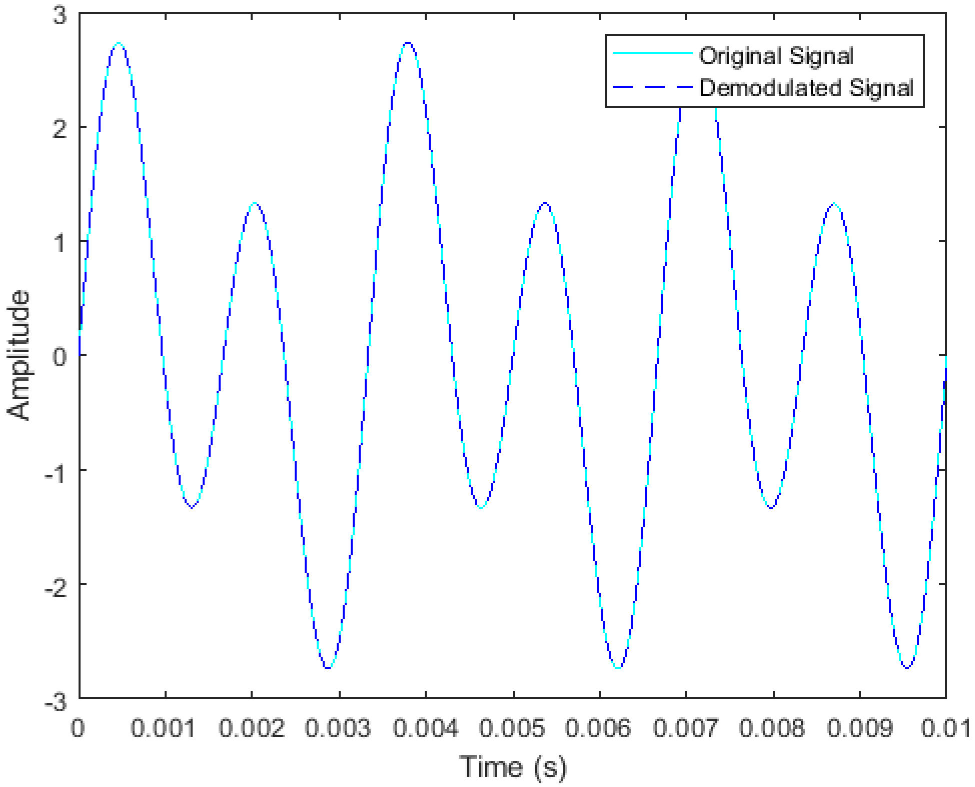
```
y = ammod(s,fc,fs);
```

Demodulate the received signal.

```
z = amdemod(y,fc,fs,0,0,num,den);
```

Plot the original and demodulated signals.

```
plot(t,s,'c',t,z,'b--')
legend('Original Signal','Demodulated Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



The demodulated signal is nearly identical to the original signal.

Input Arguments

y — Amplitude modulated input signal

scalar | vector | matrix | 3-D array

Amplitude modulated input signal, specified as a scalar, vector, matrix, or 3-D array. Each element of y must be real.

Data Types: double | single

`Fc` — Carrier signal frequency

positive scalar

Carrier signal frequency in hertz (Hz), specified as a positive scalar.

Data Types: `double`

`Fs` — Sampling frequency

positive scalar

Sampling frequency of the carrier signal and input message signal in hertz (Hz), specified as a positive scalar. To avoid aliasing, the value of `Fs` must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of the original modulated signal.

Data Types: `double`

`ini_phase` — Initial phase

scalar

Initial phase of the modulated signal in radians, specified as a scalar.

Data Types: `double`

`carramp` — Carrier amplitude

scalar

Carrier amplitude of the modulated signal, specified as a scalar.

Data Types: `double`

`num` — Lowpass Butterworth filter numerator

scalar

Lowpass Butterworth filter numerator, specified as a scalar.

Data Types: `double`

`den` — Lowpass Butterworth filter denominator

scalar

Lowpass Butterworth filter denominator, specified as a scalar.

Data Types: `double`

Output Arguments

`z` — Amplitude demodulated output signal

scalar | vector | matrix | 3-D array

Amplitude demodulated output signal, returned as a scalar, vector, matrix, or 3-D array.

See Also

`ammod` | `fmdemod` | `pmdemod` | `ssbdemod`

Topics

"Analog Passband Modulation"

Introduced before R2006a

ammod

Amplitude modulation

Syntax

```
y = ammod(x,Fc,Fs)
y = ammod(x,Fc,Fs,ini_phase)
y = ammod(x,Fc,Fs,ini_phase,carramp)
```

Description

`y = ammod(x,Fc,Fs)` returns an amplitude modulated (AM) signal `y`, given the input message signal `x`, where the carrier signal has frequency `Fc`. The carrier signal and `x` have a sampling frequency `Fs`. The modulated signal has zero initial phase and zero carrier amplitude, so the result is suppressed-carrier modulation.

Note The value of `Fs` must satisfy `Fs` ≥ 2`Fc`.

`y = ammod(x,Fc,Fs,ini_phase)` specifies the initial phase in the modulated signal `y` in radians.

`y = ammod(x,Fc,Fs,ini_phase,carramp)` performs transmitted-carrier modulation instead of suppressed-carrier modulation where `carramp` is the carrier amplitude of the modulated signal.

Examples

Compare Double-Sideband and Single-Sideband Amplitude Modulation

Set the sample rate to 100 Hz. Create a time vector 100 seconds long.

```
fs = 100;
t = (0:1/fs:100)';
```

Set the carrier frequency to 10 Hz. Generate a sinusoidal signal.

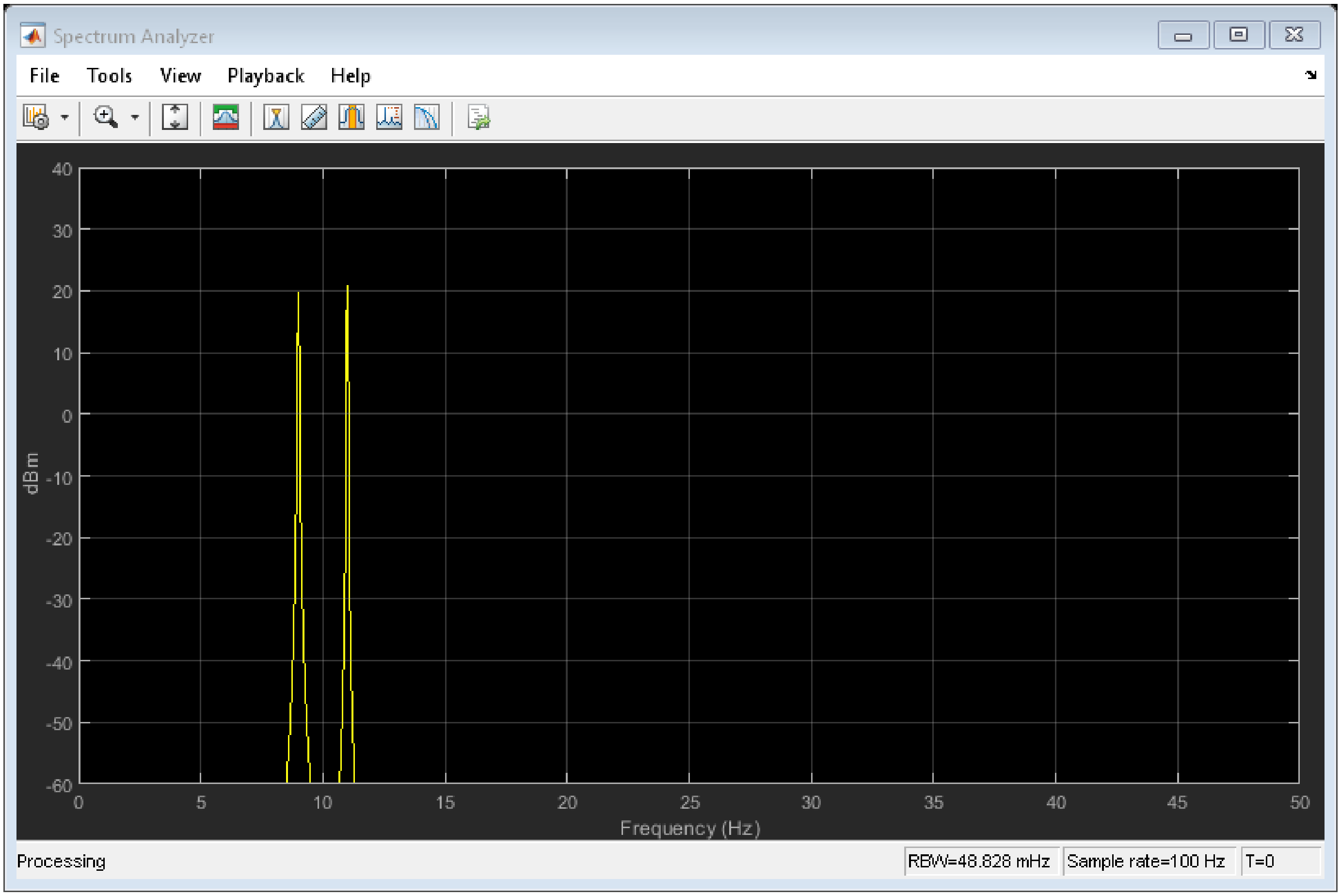
```
fc = 10;
x = sin(2*pi*t);
```

Modulate `x` using single- and double-sideband AM.

```
ydouble = ammod(x,fc,fs);
ysingle = ssbmod(x,fc,fs);
```

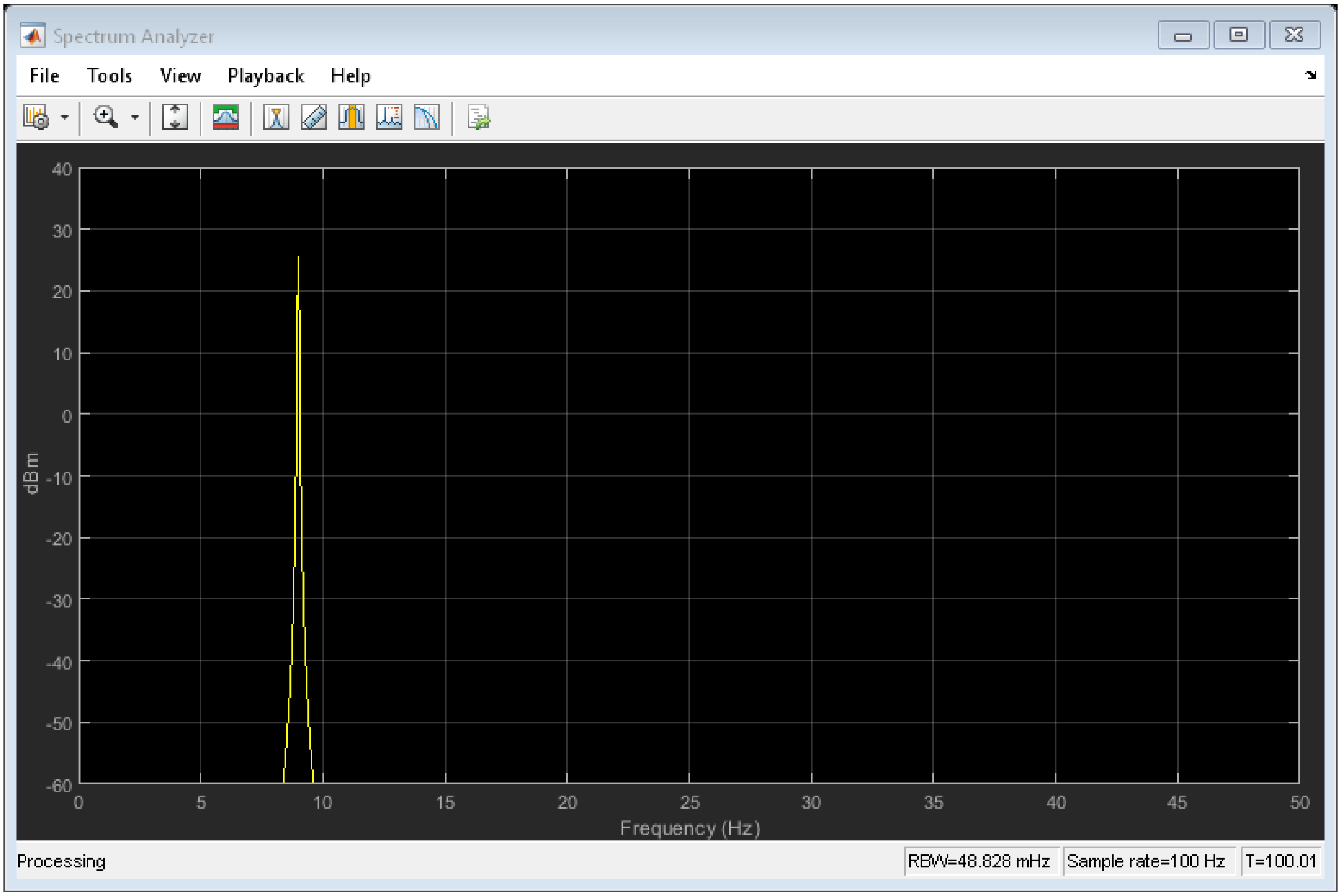
Create a spectrum analyzer object to plot the spectra of the two signals. Plot the spectrum of the double-sideband signal.

```
sa = dsp.SpectrumAnalyzer('SampleRate',fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'YLimits',[-60 40]);
step(sa,ydouble)
```



Plot the single-sideband spectrum.

```
step(sa,ysingle)
```



Input Arguments

x — Input message signal

scalar | vector | matrix | 3-D array

Input message signal, specified as a scalar, vector, matrix, or a 3-D array. Each element of `x` must be real.

Data Types: `single` | `double`

Fc — Carrier signal frequency

positive real scalar

Carrier signal frequency in hertz (Hz), specified as a positive real scalar.

Data Types: `single` | `double`

Fs — Sampling frequency

positive real scalar

Sampling frequency of carrier signal and input message signal in hertz (Hz), specified as a positive real scalar. To avoid aliasing, the value of `Fs` must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of `x`.

Data Types: `single` | `double`

`ini_phase` — Initial phase

real scalar

Initial phase of the modulated signal in radians, specified as a real scalar.

Data Types: `single` | `double`

`carramp` — Carrier amplitude

real scalar

Carrier amplitude of the modulated signal, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

y — Amplitude modulated output signal

scalar | vector | matrix | 3-D array

Amplitude modulated signal, returned as a scalar, vector, matrix, or 3-D array.

See Also

`amdemod` | `fmmod` | `pmmod` | `ssbmod`

Topics

“Analog Passband Modulation”

Introduced before R2006a

apskdemod

Amplitude phase shift keying (APSK) demodulation

Syntax

```
z = apskdemod(y,M,radii)
z = apskdemod(y,M,radii,phaseoffset)
z = apskdemod( ___ ,Name,Value)
```

Description

`z = apskdemod(y,M,radii)` performs APSK demodulation of the input signal `y`, based on the specified number of constellation points per PSK ring, `M`, and the radius of each PSK ring, `radii`. For a description of APSK demodulation, see “APSK Hard Demodulation” on page 2-22 and “APSK Soft Demodulation” on page 2-23.

Note apskdemod specifically applies to multiple ring PSK constellations. For a single ring PSK constellation, use `pskdemod`.

`z = apskdemod(y,M,radii,phaseoffset)` specifies an initial phase offset for each PSK ring of the APSK modulated signal.

`z = apskdemod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate 16-APSK Signal

Demodulate a 16-APSK signal that has an unequal number of constellation points on each circle. Plot the received constellation.

Define vectors for modulation order and PSK ring radii. Generate random 16-ary data symbols.

```
M = [4 12];
radii = [1 2];
modOrder = sum(M);

x = randi([0 modOrder-1],1000,1);
```

Apply APSK modulation to the data.

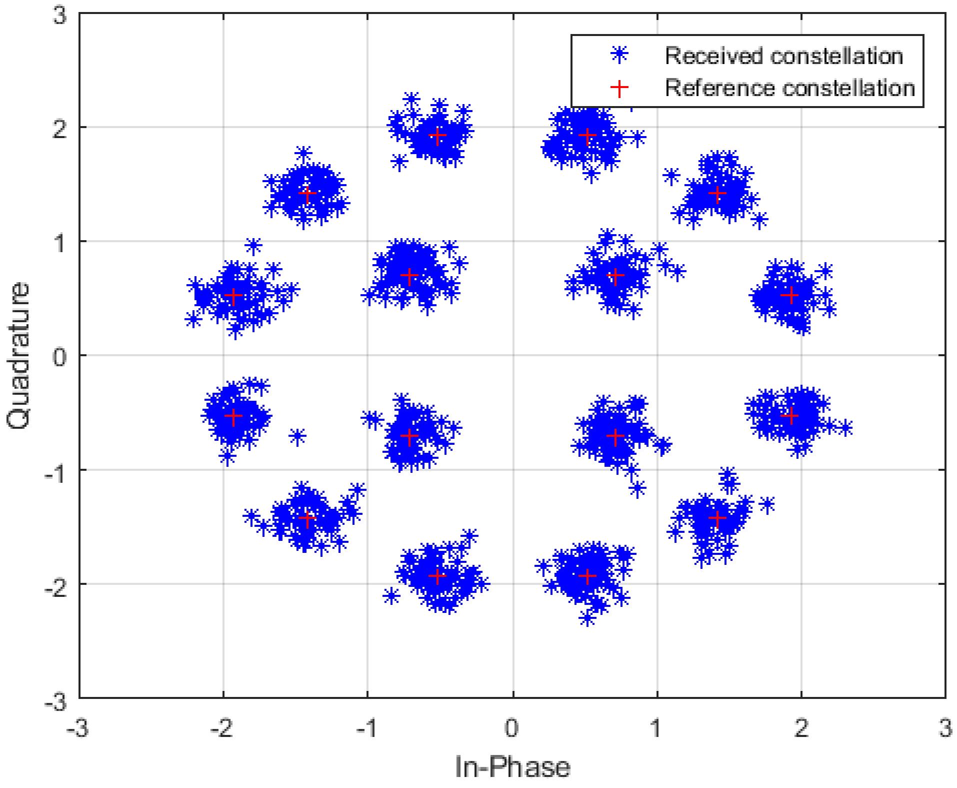
```
txSig = apskmod(x,M,radii);
```

Pass the modulated signal through a noisy channel.

```
snr = 20; % dB
rxSig = awgn(txSig,snr,'measured');
```

Plot the transmitted (reference) signal points and the noisy received signal points.

```
plot(rxSig,'b*')
hold on
grid
plot(txSig,'r+')
xlim([-3 3])
ylim([-3 3])
xlabel('In-Phase')
ylabel('Quadrature')
legend('Received constellation','Reference constellation')
```



Demodulate the received signal and compare to the input data.

```
z = apskdemod(rxSig,M,radii);
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate 64-APSK Custom Symbol Mapped Signal

Demodulate a 64-APSK signal with custom symbol mapping. Compute hard decision bit output and verify that the input matches the output.

Define vectors for modulation order and PSK ring radii. Generate 100 symbols of random bit input.

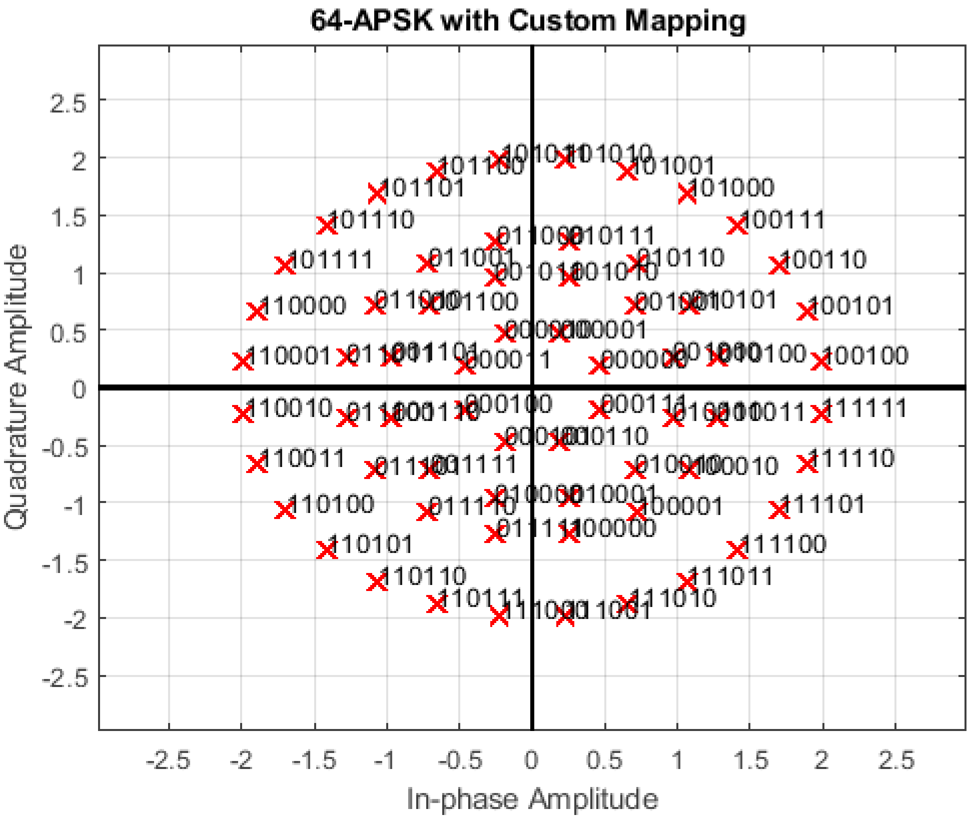
```
M = [8 12 16 28]; % 4-PSK circles
modOrder = sum(M);
radii = [0.5 1 1.3 2];
x = randi([0 1],100*log2(modOrder),1);
```

Create a custom symbol mapping vector of binary mapping.

```
cmap = 0:63;
```

Modulate the data and plot the constellation.

```
y = apskmod(x,M,radii,'SymbolMapping',cmap,'inputType','bit', ...
    'PlotConstellation',true);
```



Demodulate the received signal.

```
z = apskdemod(y,M,radii,'SymbolMapping',cmap,'OutputType','bit');
```

Verify that the demodulated signal is equal to the original data.

```
isequal(x,z)
```

```
ans = logical
   1
```

Soft Bit Demodulate 32-APSK Signal

Demodulate a 32-APSK signal and calculate soft bits.

Define vectors for modulation order and PSK ring radii. Generate 10000 symbols of random bit data.

```
M = [16 16];
modOrder = sum(M);
radii = [0.6 1.2];
numSym = 10000;
x = randi([0 1], numSym*log2(modOrder),1);
```

Generate a reference constellation. Create a constellation diagram object.

```
refAPSK = apskmod(0:modOrder-1,M,radii);
constDiagAPSK = comm.ConstellationDiagram('ReferenceConstellation',refAPSK, ...
  'Title','Received Symbols','XLimits',[-2 2],'YLimits',[-2 2]);
```

Modulate the data.

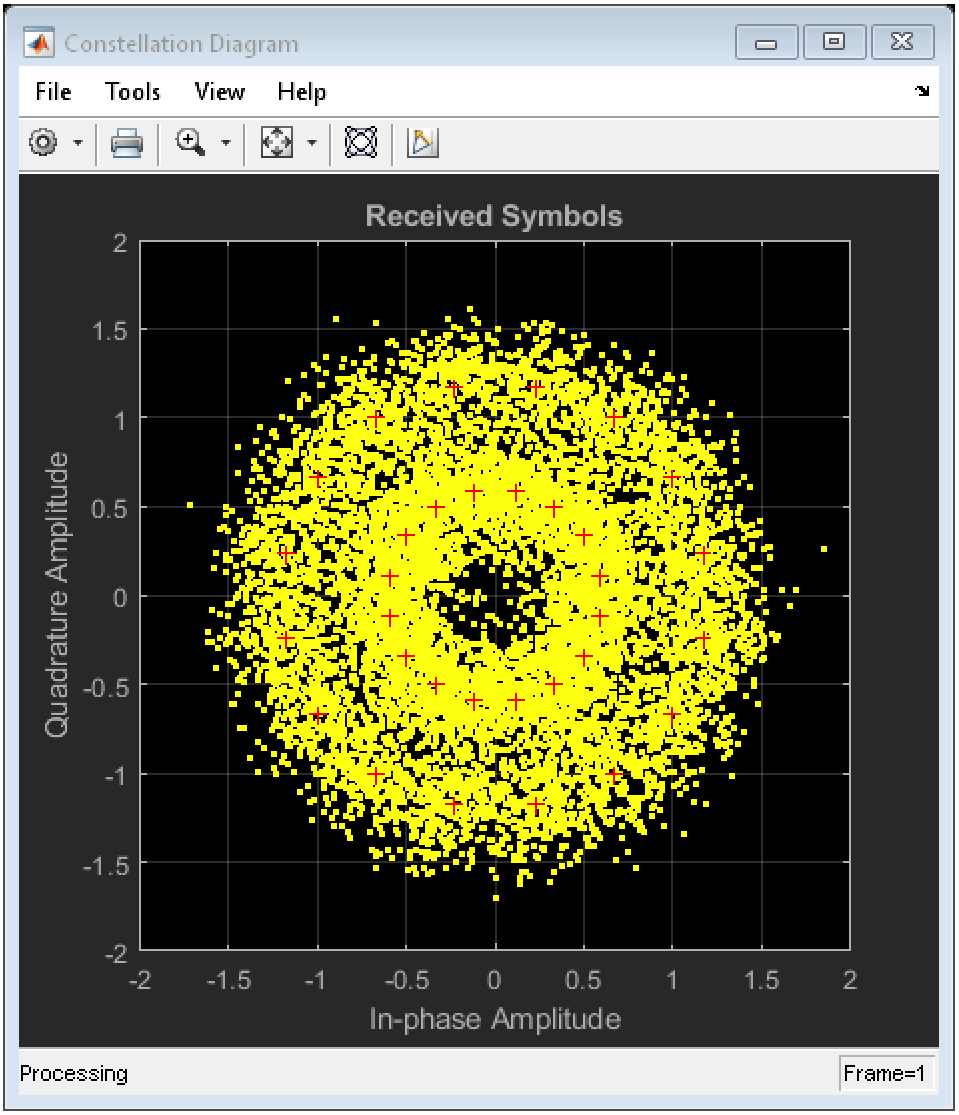
```
txSig = apskmod(x,M,radii,'InputType','bit');
sigPow = var(txSig);
```

Pass the signal through a noisy channel.

```
snr = 15;
rxSig = awgn(txSig,snr,sigPow,'linear');
```

Plot the reference and received constellation symbols.

```
constDiagAPSK(rxSig)
```



Demodulate the signal and compute soft bits.

```
z = apskdemod(rxSig,M,radii,'OutputType','approxllr', ...
    'NoiseVariance',sigPow/snr);
```

Input Arguments

y — APSK modulated signal

scalar | vector | matrix

APSK modulated signal, specified as a complex scalar, vector, or matrix. Each column is treated as an independent channel.

Data Types: double | single
Complex Number Support: Yes

M — Constellation points per PSK ring

vector

Constellation points per PSK ring, specified as a vector with more than one element. Vector elements indicate the number of constellation points in each PSK ring. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. Element values must be multiples of four and `sum(M)` must be a power of two. The modulation order is the total number of points in the signal constellation and equals the sum of the vector elements, `sum(M)`.

Example: `[4 12 16]` specifies a three PSK ring constellation with a modulation order of `sum(M) = 32`.

Data Types: `double`

`radii` — PSK ring radii

vector

PSK ring radii, specified as a vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. The elements must be positive and arranged in increasing order.

Example: `[0.5 1 2]` defines constellation PSK ring radii. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

Data Types: `double`

`phaseoffset` — PSK ring phase offsets

`[pi/M(1) pi/M(2) … pi/M(end)]` (default) | scalar | vector

Phase offset of each PSK ring in radians, specified as a scalar or vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, that corresponds to the outermost circle. The `phaseoffset` can be a scalar only if all the elements of `M` are the same value.

Example: `[pi/4 pi/12 pi/16]` defines three constellation PSK ring phase offsets. The inner ring has a phase offset of pi/4, the second ring has a phase offset of pi/12, and the outer ring has a phase offset of pi/16.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = apskdemod(x,M,radii,'OutputType','bit','OutputDataType','single');`

`SymbolMapping` — Symbol mapping

`'gray'` | `'contourwise-gray'` | integer vector

Symbol mapping, specified as the comma-separated pair consisting of `SymbolMapping` and one of the following:

- `'contourwise-gray'` — Uses Gray mapping along the contour in phase dimension.
- `'gray'` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all the values for `M` must be equal and all the values for `phaseoffset` must be equal. For a description of the Gray mapping used, see [2].
- integer vector — Use custom symbol mapping. Vector must consist of `sum(M)` unique elements with values from 0 to (`sum(M)-1`). The first element corresponds to the constellation point in the

first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

The default symbol mapping depends on M and phaseOffset. When all the elements of M and phaseOffset are equal, the default is 'gray'. For all other cases, the default is 'contourwise-gray'.

Data Types: double | char | string

OutputType — Output type

'integer' (default) | 'bit' | 'llr' | 'approxllr'

Output type, specified as the comma-separated pair consisting of 'OutputType' and 'integer', 'bit', 'llr', or 'approxllr'. For a description of the returned output, see z.

Data Types: char | string

OutputDataType — Output data type

'double' (default) | ...

Output data type, specified as the comma-separated pair consisting of OutputDataType and one of the indicated data types. Acceptable values for OutputDataType depend on the OutputType value.

OutputType Value	Acceptable OutputDataType Values
'integer'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', or 'uint32'
'bit'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', 'uint32', or 'logical'

Dependencies

This name-value pair argument applies only when OutputType is set to 'integer' or 'bit'.

Data Types: char | string

NoiseVariance — Noise variance

1 (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of NoiseVariance and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “APSK Soft Demodulation” on page 2-23 for algorithm selection considerations.

Dependencies

This name-value pair argument applies only when OutputType is set to 'llr' or 'approxllr'.

Data Types: double

PlotConstellation — Option to plot constellation

false (default) | true

Option to plot constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

z — Demodulated signal

scalar | vector | matrix

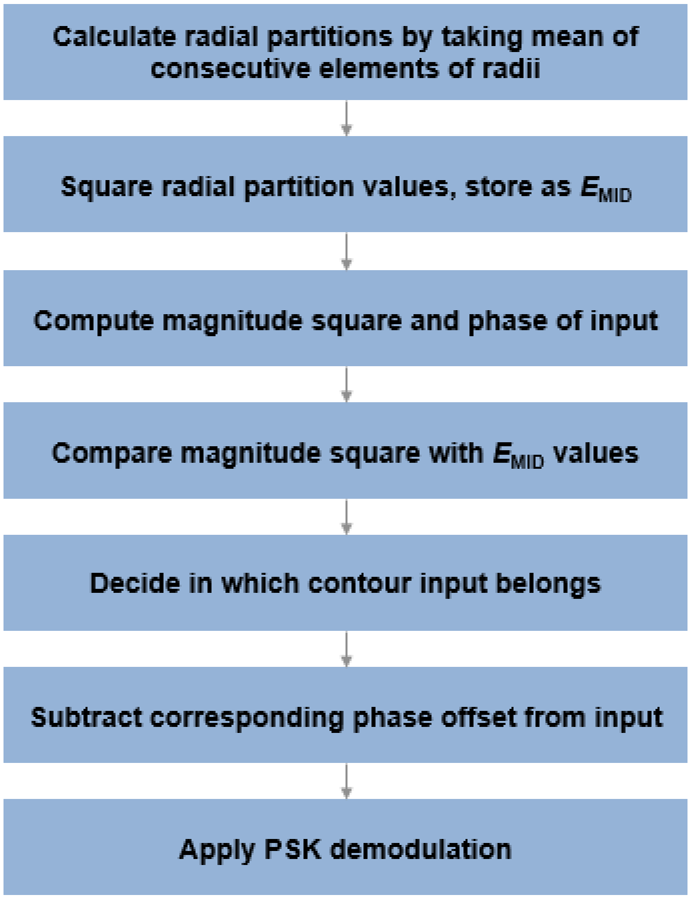
Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of z depend on the specified OutputType value.

OutputType Value	Return Value of apskdemod	Dimensions of z
'integer'	Demodulated integer values from 0 to (sum(M) – 1)	z has the same dimensions as input y.
'bit'	Demodulated bits	The number of rows in z is $\log_2$(sum(M)) times the number of rows in y. Each demodulated symbol is mapped to a group of $\log_2$(sum(M)) elements in a column, where the first element represents the MSB and the last element represents the LSB.
'llr'	Log-likelihood ratio value for each bit	
'approxllr'	Approximate log-likelihood ratio value for each bit	

More About

APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding as described in [1].


Calculate radial partitions by taking mean of consecutive elements of radii
Square radial partition values, store as E_{MID}
Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide in which contour input belongs
Subtract corresponding phase offset from input
Apply PSK demodulation

APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

References

[1] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskmod` | `dvbsapskdemod` | `genqamdemod` | `mil188qamdemod` | `pskdemod` | `qamdemod`

Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

apskmod

Amplitude phase shift keying (APSK) modulation

Syntax

```
y = apskmod(x,M,radii)
y = apskmod(x,M,radii,phaseoffset)
y = apskmod( ___ ,Name,Value)
```

Description

y = apskmod(x,M,radii) performs APSK modulation on the input data, x, based on the specified number of constellation points per PSK ring, M, and the radius of each PSK ring, radii. For a description of APSK modulation, see “Algorithms” on page 2-32.

Note apskmod specifically applies to multiple ring PSK constellations. For a single ring PSK constellation, use pskmod.

y = apskmod(x,M,radii,phaseoffset) specifies an initial phase offset for each PSK ring of the APSK modulated signal.

y = apskmod(___ ,Name,Value) specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, 'OutputDataType','double' specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply APSK Modulation

Modulate data using APSK with an unequal number of constellation points on each circle.

Define vectors for modulation order and PSK ring radii. Generate data for constellation points.

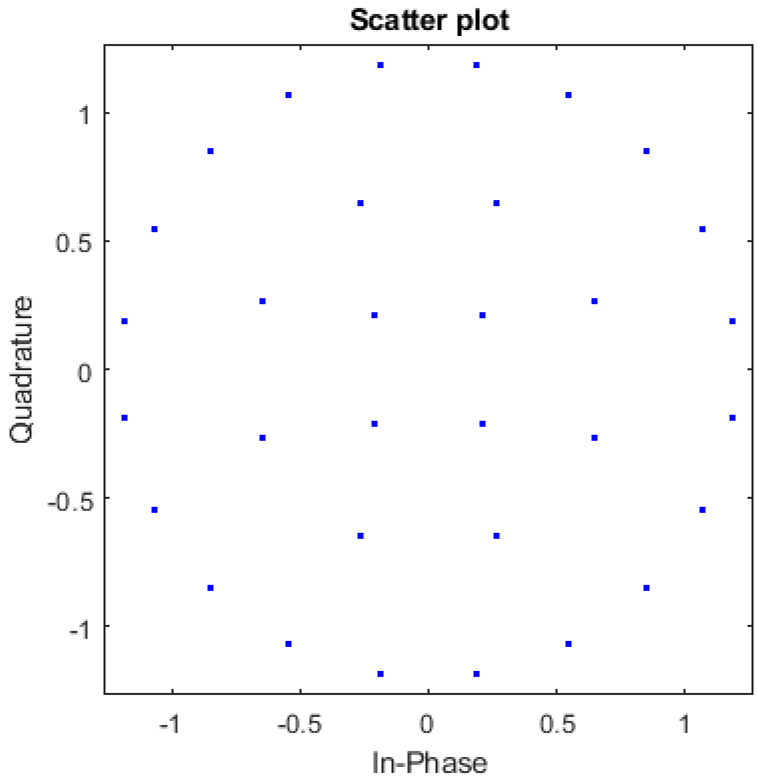
```
M = [4 8 20];
radii = [0.3 0.7 1.2];
modOrder = sum(M);
x = 0:modOrder-1;
```

Apply APSK modulation to the data.

```
y = apskmod(x,M,radii);
```

Plot the resulting constellation using a scatter plot.

```
scatterplot(y)
```



Apply APSK Modulation with Phase Offset

Modulate a random data sequence using APSK with zero phase offset for the inner circle and pi/6 phase offset for the outer circle.

Define vectors for modulation order, PSK ring radii, and PSK ring phase offset. Generate random data.

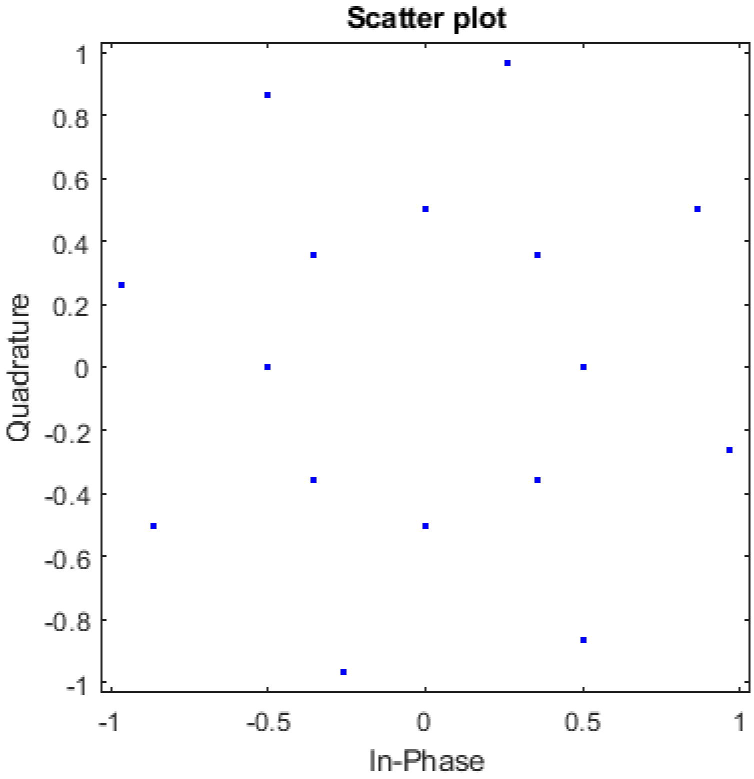
```
M = [8 8];
modOrder = sum(M);
radii = [0.5 1];
phOff = [0 pi/6];

x = randi([0 modOrder-1],100,1);
```

Apply APSK modulation to the data.

```
y = apskmod(x,M,radii,phOff);
```

Plot the resulting constellation using a scatter plot and observe the phase offset between the constellation circles.

```
scatterplot(y)
```



Apply APSK Modulation Modifying Symbol Ordering

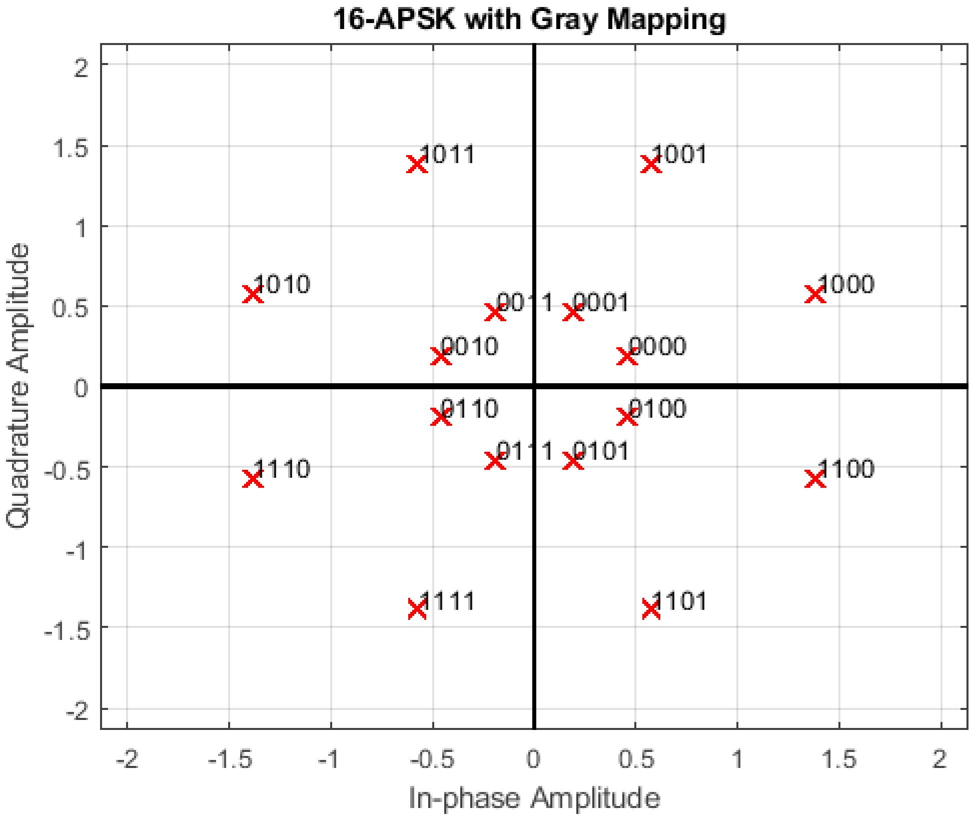
Plot APSK constellations for gray and custom symbol mappings.

Define vectors for modulation order and PSK ring radii. Generate bit data for constellation points.

```
M = [8 8];
modOrder = sum(M);
radii = [0.5 1.5];
x = 0:modOrder-1;
xBit = de2bi(x);
```

Apply APSK modulation to the data using the default phase offset. Since element values for `M` are equal and element values for phase offset are equal, the symbol mapping defaults to `'gray'`. Binary input is used to highlight the Gray nature of the constellation mapping. Plot the constellation.

```
y = apskmod(xBit,M,radii,'PlotConstellation',true,'InputType','bit');
```

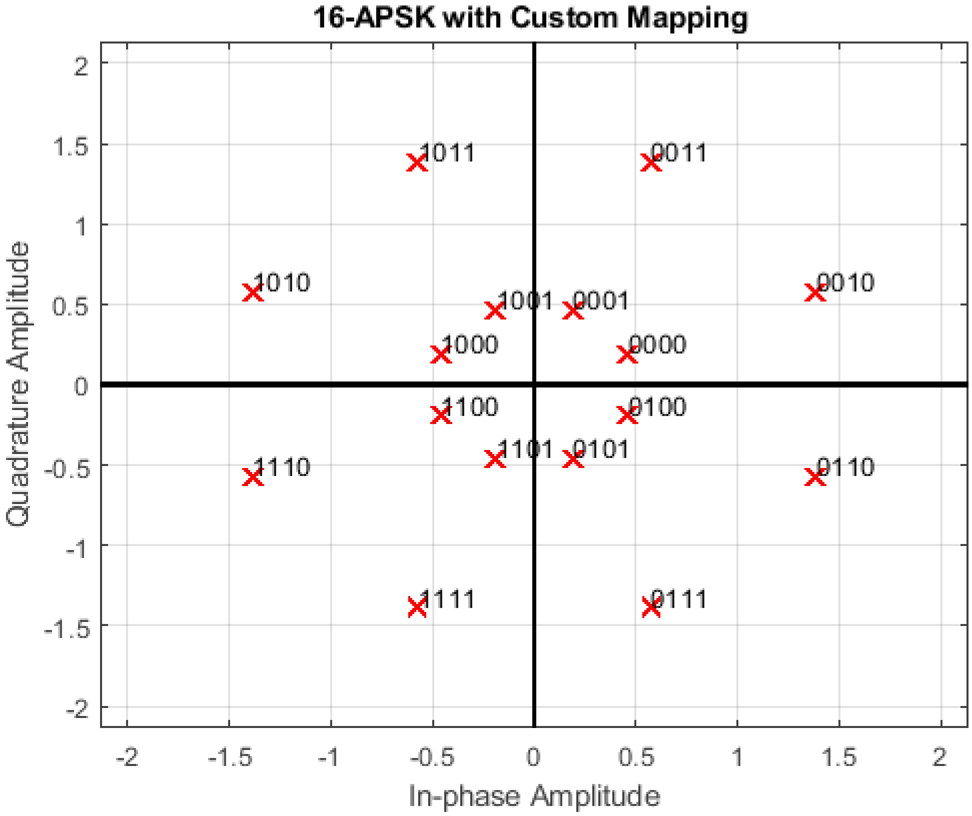


Create a custom symbol mapping vector. This custom mapping happens to be another Gray mapping.

```
cmap = [0;1;9;8;12;13;5;4;2;3;11;10;14;15;7;6];
```

Apply APSK modulation with a custom symbol mapping. Plot the constellation. Binary input is used to highlight that the custom mapping defines different Gray symbol mapping.

```
z = apskmod(xBit,M,radii,'SymbolMapping',cmap,'PlotConstellation',true,'InputType','bit');
```



Apply APSK Modulation to Input Bits

Modulate a random bit sequence using APSK and output data type `single`. Pass the signal through a noisy channel and display the constellation diagram.

Define vectors for modulation order and PSK ring radii. Generate random binary data.

```
M = [8 12 20 24];
radii = [0.8 1.2 2 2.5];
bitsPerSym = log2(sum(M));

x = randi([0 1],2000*bitsPerSym,1);
```

Apply APSK modulation to the data and use a name-value pair to output as data type `single`.

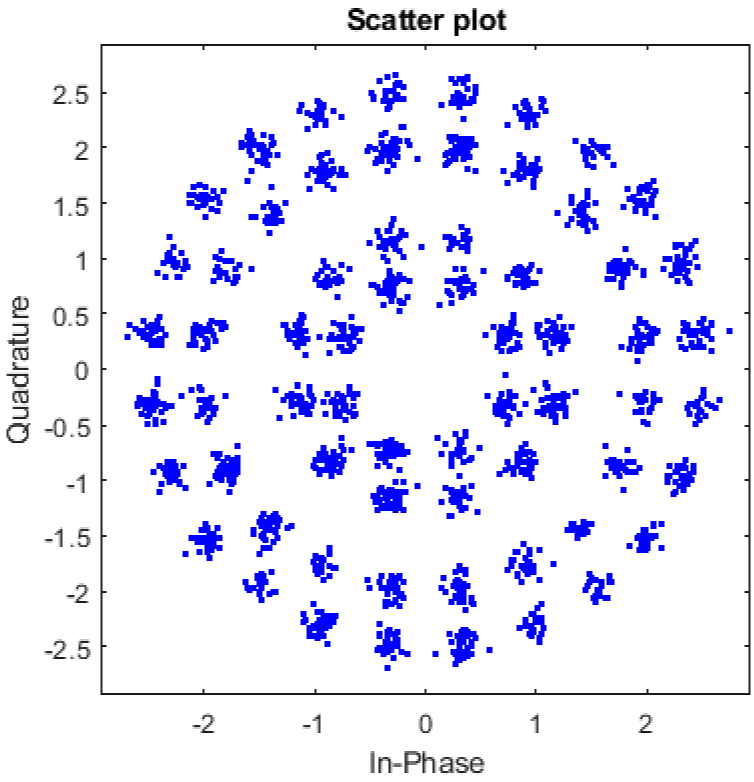
```
y = apskmod(x,M,radii,'InputType','bit','OutputDataType','single');
```

Pass through an AWGN channel with a 25 dB SNR.

```
yrec = awgn(y,25,'measured');
```

Plot the received constellation as a scatter plot.

```
scatterplot(yrec)
```



Input Arguments

x — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers in the range [0, (`sum(M)-1`)].

Note To process the input signal as binary elements, set the '`InputType`' name-value pair to `'bit'`. For binary inputs, the number of rows must be an integer multiple of $\log_2(\text{sum}(M))$. Groups of $\log_2(\text{sum}(M))$ bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`M` — Constellation points per PSK ring

vector

Constellation points per PSK ring, specified as a vector with more than one element. Each vector element indicates the number of constellation points in its corresponding PSK ring. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. Element values must be multiples of four and `sum(M)` must be a power of two. The modulation order is the total number of points in the signal constellation and equals the sum of the vector elements, `sum(M)`.

Example: `[4 12 16]` specifies a three PSK ring constellation with a modulation order of `sum(M) =` 32.

Data Types: `double`

`radii` — Radius per PSK ring

vector

Radius per PSK ring, specified as a vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. The elements must be positive and arranged in increasing order.

Example: `[0.5 1 2]` defines radii for three constellation PSK rings. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

Data Types: `double`

`phaseoffset` — Phase offset per PSK ring

`[pi/M(1) pi/M(2) … pi/M(end)]` (default) | scalar | vector

Phase offset per PSK ring in radians, specified as a scalar or vector with the same length as `M`. The first element corresponds to the innermost circle, and so on, until the last element, which corresponds to the outermost circle. The `phaseoffset` can be a scalar only if all the elements of `M` are the same value.

Example: `[pi/4 pi/12 pi/16]` defines three constellation PSK ring phase offsets. The inner ring has a phase offset of pi/4, the second ring has a phase offset of pi/12, and the outer ring has a phase offset of pi/16.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = apskmod(x,M,radii,'InputType','bit','OutputDataType','single');`

`SymbolMapping` — Symbol mapping

`'gray'` | `'contourwise-gray'` | integer vector

Symbol mapping, specified as the comma-separated pair consisting of `'SymbolMapping'` and one of the following:

- `'contourwise-gray'` — Uses Gray mapping along the contour in the phase dimension for each PSK ring.
- `'gray'` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all the values for `M` must be equal and all the values for `phaseoffset` must be equal. For a description of the Gray mapping used, see [2].
- integer vector — Use custom symbol mapping. Vector must consist of `sum(M)` unique elements with values in the range [0, (`sum(M)-1`]. The first element corresponds to the constellation point in the first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

The default symbol mapping depends on M and phaseOffset. When all the elements of M are equal and all the elements of phaseOffset are equal, the default is 'gray'. For all other cases, the default is 'contourwise-gray'.

Data Types: double | char | string

InputType — Input type
'integer' (default) | 'bit'

Input type, specified as the comma-separated pair consisting of 'InputType' and either 'integer' or 'bit'. To use 'integer', the input signal must consist of integers in the range [0, (sum(*M*) – 1)]. To use 'bit', the input signal must contain binary values and the number of rows must be an integer multiple of $\log_2$(sum(M)).

Data Types: char | string

OutputDataType — Output data type
'double' (default) | 'single'

Output data type, specified as the comma-separated pair consisting of 'OutputDataType' and either 'double' or 'single'.

Data Types: char | string

PlotConstellation — Plot reference constellation
false (default) | true

Plot reference constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the reference constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

y — APSK modulated signal
scalar | vector | matrix

APSK modulated signal, returned as a complex scalar, vector, or matrix. The dimensions of y depend on the specified 'InputType' value.

InputType	Dimensions of y
'integer'	y has the same dimensions as input x.
'bit'	The number of rows in y equals the number of rows in x divided by $\log_2$(sum(M)).

Algorithms

The function implements a pure APSK constellation.

A pure M-APSK constellation is composed of N_C concentric rings or contours, each with uniformly spaced PSK points. The M-APSK constellation set is

$$\chi = \begin{cases} R_1 \exp\left(j\left(\frac{2\pi}{M_1} i + \theta_1\right)\right), & i = 0, \ldots, M_1 - 1, \\ R_2 \exp\left(j\left(\frac{2\pi}{M_2} i + \theta_2\right)\right), & i = 0, \ldots, M_2 - 1, \\ \vdots & \vdots \\ R_{N_C} \exp\left(j\left(\frac{2\pi}{M_{N_C}} i + \theta_{\mathrm{Nc}}\right)\right), & i = 0, \ldots, M_{N_C} - 1, \end{cases}$$

where

- The modulation order is equal to the sum of all M_l for l = 1, 2, ... , N_C.
- N_C is the number of concentric rings. $N_C \geq 2$.
- M_l is the number of constellation points in the lth ring.
- R_l is the radius of the lth ring.
- θ_l is the phase offset of the lth ring.
- $j = \sqrt{-1}$

References

[1] Corazza, Giovanni E. *Digital Satellite Communications*. New York: Springer Science Business Media, LLC, 2007.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskmod` | `mil188qammod` | `pskmod` | `qammod`

Objects

`comm.GeneralQAMModulator` | `comm.PSKModulator`

Introduced in R2018a

arithdeco

Decode binary code by arithmetic decoding

Syntax

```
dseq = arithdeco(code,counts,len)
```

Description

`dseq = arithdeco(code,counts,len)` decodes the binary arithmetic code in `code` to recover the corresponding sequence of `len` symbols. The input `counts`specifies the statistics of the source by listing the number of times each symbol of the source alphabet occurs in a test data set. `code` must be an output of the `arithenco` function.

Examples

Decode Sequence Using Arithmetic Code

Using a source with a two-symbol alphabet, produce a test data set in which 99% of the symbols are 1s. Encode 1000 symbols from this source to produce a code vector with less than 1000 elements. The actual number of elements in the encoded sequence varies depending on the particular random sequence.

Specify for symbol 1 from the source alphabet to occur 99 times in the test data set.

```
counts = [99 1];
```

Generate a random sequence of length 1000.

```
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]);
```

Encode the random sequence. Then, decode the encoded sequence.

```
code = arithenco(seq,counts);
dseq = arithdeco(code,counts,length(seq));
```

Verify that the decoded sequence matches the original random sequence.

```
isequal(seq,dseq)
```

```
ans = logical
   1
```

Input Arguments

code — Binary arithmetic code
nonnegative binary row vector

Binary arithmetic code, specified as a nonnegative binary row vector. This value must be a binary arithmetic code produced by the `arithenco` function.

Data Types: `double`

`counts` — Statistics of symbols
positive numeric vector

Statistics of symbols, specified as a positive numeric vector. This input specifies the number of times each symbol of the source alphabet occurs in a test data set.

Data Types: `double`

`len` — Length of sequence
positive scalar

Length of the sequence to decode, specified as a positive scalar.

Data Types: `double`

Output Arguments

`dseq` — Decoded arithmetic code
positive numeric row vector

Decoded arithmetic code with a sequence of `len` source symbols, specified as a positive numeric row vector.

Algorithms

The `arithdeco` function uses the algorithm described in [1] .

References

[1] Sayood, Khalid. *Introduction to Data Compression*. 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

See Also

Functions
`arithenco`

Topics
"Arithmetic Coding"

Introduced before R2006a

arithenco

Encode sequence of symbols by arithmetic encoding

Syntax

```
code = arithenco(seq,counts)
```

Description

`code = arithenco(seq,counts)` generates the binary arithmetic code corresponding to the sequence of symbols specified in `seq`. The input `counts` specifies the statistics of the source by listing the number of times each symbol of the source alphabet occurs in a test data set.

Examples

Encode Data Sequence with Arithmetic Code

Using a source with a two-symbol alphabet, produce a test data set in which 99% of the symbols are 1s. Encode 1000 symbols from this source to produce a code vector with less than 1000 elements. The actual number of elements in the encoded sequence varies depending on the particular random sequence.

Specify for symbol 1 from the source alphabet to occur 99 times in the test data set.

```
counts = [99 1]
```

```
counts = 1×2

    99     1
```

Generate a random sequence of length 1000.

```
len = 1000;
seq = randsrc(1,len,[1 2; .99 .01]);
```

Encode the random sequence and display the encoded length.

```
code = arithenco(seq,counts);
s = size(code)
```

```
s = 1×2

     1    57
```

Input Arguments

seq — Sequence of symbols
positive numeric row vector

Sequence of symbols, specified as a positive numeric row vector. This input specifies the random sequence for the function to encode.

Data Types: `double`

`counts` — Statistics of symbols

positive numeric row vector

Statistics of symbols, specified as a positive numeric row vector. This input specifies the number of times each symbol of the source alphabet occurs in a test data set.

Data Types: `double`

Output Arguments

code — Generated binary arithmetic code

nonnegative binary row vector

Generated binary arithmetic code corresponding to the sequence of source symbols, returned as a nonnegative binary row vector.

Algorithms

The `arithenco` function uses the algorithm described in [1].

References

[1] Sayood, Khalid. *Introduction to Data Compression*. 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

See Also

Functions
`arithdeco`

Topics
"Arithmetic Coding"

Introduced before R2006a

awgn

Add white Gaussian noise to signal

Syntax

```
out = awgn(in,snr)
out = awgn(in,snr,signalpower)

out = awgn(in,snr,signalpower,randobject)
out = awgn(in,snr,signalpower,seed)
out = awgn( ___ ,powertype)
```

Description

`out = awgn(in,snr)` adds white Gaussian noise to the vector signal `in`. This syntax assumes that the power of `in` is 0 dBW.

`out = awgn(in,snr,signalpower)` accepts an input signal power value in dBW. To have the function measure the power of `in` before adding noise, specify `signalpower` as `'measured'`.

`out = awgn(in,snr,signalpower,randobject)` accepts input combinations from prior syntaxes and a random number stream object to generate normal random noise samples. For information about producing repeatable noise samples, see “Tips” on page 2-43.

`out = awgn(in,snr,signalpower,seed)` specifies a seed value for initializing the normal random number generator that is used when adding white Gaussian noise to the input signal. For information about producing repeatable noise samples, see “Tips” on page 2-43.

`out = awgn( ___ ,powertype)` specifies the signal and noise power type as `'dB'` or `'linear'` in addition to the input arguments in any of the previous syntaxes.

For the relationships between SNR and other measures of the relative power of the noise, such as E_s/N_0, and E_b/N_0, see “AWGN Channel Noise Level”.

Examples

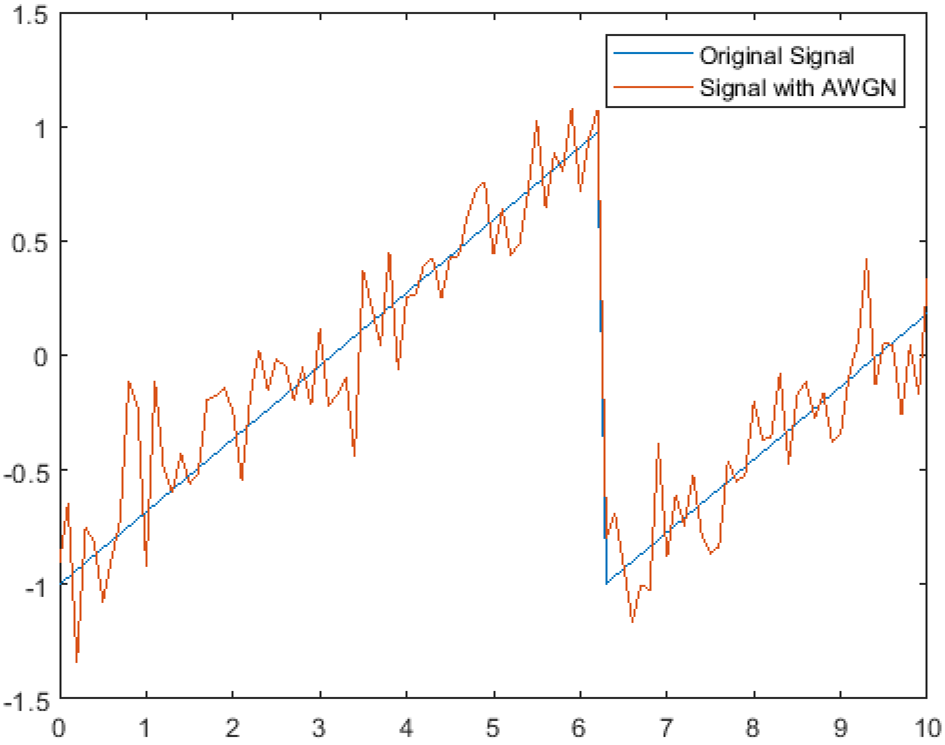
Add AWGN to Sawtooth Signal

Create a sawtooth wave.

```
t = (0:0.1:10)';
x = sawtooth(t);
```

Apply white Gaussian noise and plot the results.

```
y = awgn(x,10,'measured');
plot(t,[x y])
legend('Original Signal','Signal with AWGN')
```



General QAM Modulation in AWGN Channel

Transmit and receive data using a nonrectangular 16-ary constellation in the presence of Gaussian noise. Show the scatter plot of the noisy constellation and estimate the symbol error rate (SER) for two different signal-to-noise ratios.

Create a 16-QAM constellation based on the V.29 standard for telephone-line modems.

```
c = [-5 -5i 5 5i -3 -3-3i -3i 3-3i 3 3+3i 3i -3+3i -1 -1i 1 1i];
M = length(c);
```

Generate random symbols.

```
data = randi([0 M-1],2000,1);
```

Modulate the data by using the `genqammod` function. General QAM modulation is necessary because the custom constellation is not rectangular.

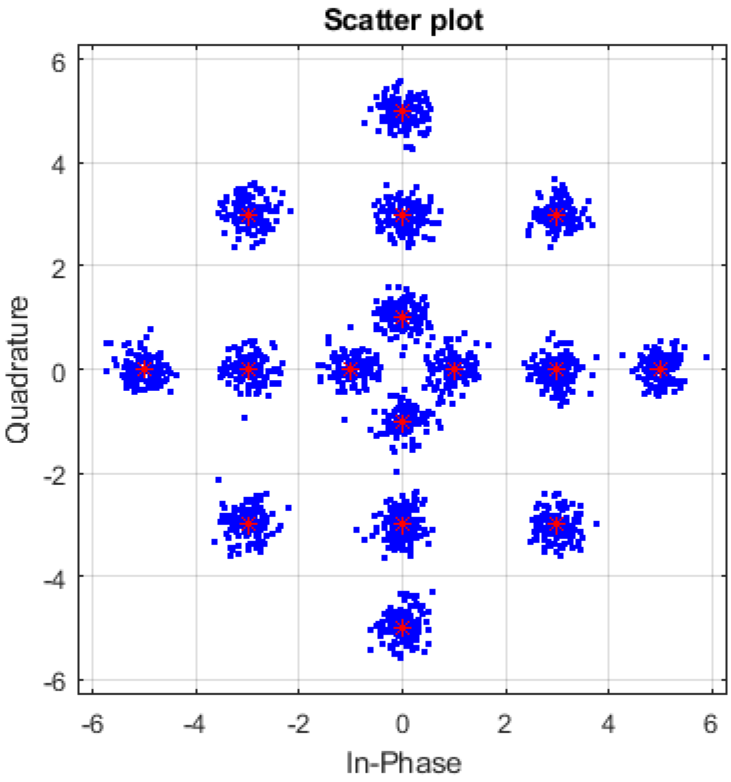
```
modData = genqammod(data,c);
```

Pass the signal through an AWGN channel having a 20 dB signal-to-noise ratio (SNR).

```
rxSig = awgn(modData,20,'measured');
```

Display a scatter plot of the received signal and the reference constellation, `c`.

```
h = scatterplot(rxSig);
hold on
scatterplot(c,[],[],'r*',h)
grid
hold off
```



Demodulate the received signal by using the `genqamdemod` function. Determine the number of symbol errors and the symbol error ratio.

```
demodData = genqamdemod(rxSig,c);
[numErrors,ser] = symerr(data,demodData)
```

```
numErrors = 1
```

```
ser = 5.0000e-04
```

Repeat the transmission and demodulation process with an AWGN channel having a 10 dB SNR. Determine the symbol error rate for the reduced SNR. As expected, the performance degrades when the SNR is decreased.

```
rxSig = awgn(modData,10,'measured');
demodData = genqamdemod(rxSig,c);
[numErrors,ser] = symerr(data,demodData)
```

```
numErrors = 462
```

```
ser = 0.2310
```

Repeatable AWGN with RandStream

Generate white Gaussian noise addition results using a `RandStream` object and the `reset` object function.

Specify the power of X to be `0` dBW, add noise to produce an SNR of `10` dB, and utilize a local random stream.

```
S = RandStream('mt19937ar','Seed',5489);
sigin = sqrt(2)*sin(0:pi/8:6*pi);
sigout1 = awgn(sigin,10,0,S);
```

Add AWGN to `sigin`. Use `isequal` to compare `sigout1` to `sigout2`. The outputs are not equal when the random stream was not reset.

```
sigout2 = awgn(sigin,10,0,S);
isequal(sigout1,sigout2)
```

```
ans = logical
   0
```

Reset the random stream object, returning the object to its state prior to adding AWGN to `sigout1`. Add AWGN to `sigin` and compare `sigout1` to `sigout3`. The outputs are equal after the random stream was reset.

```
reset(S);
sigout3 = awgn(sigin,10,0,S);
isequal(sigout1,sigout3)
```

```
ans = logical
   1
```

Input Arguments

in — Input signal

scalar | vector | array

Input signal, specified as a scalar, vector, or array. The power of the input signal is assumed to be 0 dBW.

Data Types: `double`
Complex Number Support: Yes

`snr` — Signal-to-noise ratio

scalar

Signal-to-noise ratio in dB, specified as a scalar.

Note When the noise is added, this function applies the same `snr` to all elements of the full input signal. Array input signals do not have a notion of independent channels. To consider multiple channels independently, see `comm.AWGNChannel`.

Data Types: `double`

`signalpower` — Signal power

scalar | `'measured'`

Signal power, specified as a scalar or `'measured'`.

- When `signalpower` is a scalar, the value is used as the signal level of `in` to determine the appropriate noise level based on the value of `snr`.
- When `signalpower` is `'measured'`, the signal level of `in` is computed to determine the appropriate noise level based on the value of `snr`.

> **Note** When you specify `'measured'`, this function computes the signal power using all elements of the full input signal. When the power is computed, array input signals do not have a notion of independent channels.

Data Types: `double`

`randobject` — Random number stream object

`RandStream` object

Random number stream object, specified as a `RandStream` object. The state of the random stream object determines the sequence of numbers produced by the `randn` function. Configure the random stream object using the `reset (RandStream)` function and its properties.

`wgn` generates normal random noise samples using `randn`. The `randn` function uses one or more uniform values from the `RandStream` object to generate each normal value.

For information about producing repeatable noise samples, see “Tips” on page 2-43.

`seed` — Random number generator seed

scalar

Random number generator seed value, specified as a scalar.

Data Types: `double`

`powertype` — Signal power unit

`'dB'` (default) | `'linear'`

Signal power unit, specified as `'dB'` or `'linear'`

- When `powertype` is `'dB'`, the `snr` is measured in dB and `signalpower` is measured in dBW.
- When `powertype` is `'linear'`, the `snr` is measured as a ratio and `signalpower` is measured in watts.

For the relationships between SNR and other measures of the relative power of the noise, such as E_s/N_0, and E_b/N_0, see “AWGN Channel Noise Level”.

Output Arguments

`out` — Output signal

scalar | vector | array

Output signal, returned as a scalar, vector, or array. The returned output signal is the input signal with white Gaussian noise added to it.

Tips

- To generate repeatable white Gaussian noise samples, use one of these tips:
  - Provide a static `seed` value as an input to `awgn`.
  - Use the `reset (RandStream)` function on the `randobject` before passing it as an input to `awgn`.
  - Provide `randobject` in a known state as an input to `awgn`. For more information, see `RandStream`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation supported, except for syntaxes that include a `RandStream` object.

See Also

Functions

`RandStream` | `bsc` | `randn` | `wgn`

Objects

`comm.AWGNChannel`

Topics

"AWGN Channel Noise Level"

Introduced before R2006a

bchdec

BCH decoder

Syntax

```
decoded = bchdec(code,N,K)
decoded = bchdec(code,N,K,paritypos)
[decoded,cnumerr] = bchdec( ___ )
[decoded,cnumerr,ccode] = bchdec( ___ )
```

Description

`decoded = bchdec(code,N,K)` attempts to decode the received signal in `code` using an (`N`,`K`) BCH decoder with the narrow-sense generator polynomial. Parity symbols are at the end and the leftmost symbol is the most significant symbol.

In the `decoded` Galois field array, each row represents the attempt at decoding the corresponding row in `code`.

`decoded = bchdec(code,N,K,paritypos)` specifies in `paritypos` whether the parity symbols in `code` were appended or prepended to the message in the coding operation.

`[decoded,cnumerr] = bchdec( ___ )` returns a column vector, `cnumerr`, where each element is the number of corrected errors in the corresponding row of `code`. You can return `cnumerr` with either of the preceding syntaxes.

`[decoded,cnumerr,ccode] = bchdec( ___ )` returns `ccode`, the corrected version of `code`.

Examples

Results of Error Correction

BCH-decode an input that has more errors per codeword than the error correcting capability of the BCH decoder. Decode a BCH coded message with two errors per codeword using a single-error correcting BCH decoder. View the effects of the error mismatch on the output codeword.

Check the number of errors per codeword a [63,57] BCH decoder is capable of correcting.

```
n = 63;
k = 57;
t = bchnumerr(n,k)
```

```
t = 1
```

The [63,57] BCH decoder is capable of correcting one error per codeword.

Create a random stream and use it to generate a GF array. Encode the message.

```
s = RandStream('swb2712','Seed',9973);
msg = gf(randi(s,[0 1],1,k));
code = bchenc(msg,n,k);
```

Add two errors per codeword and decode the errored code.

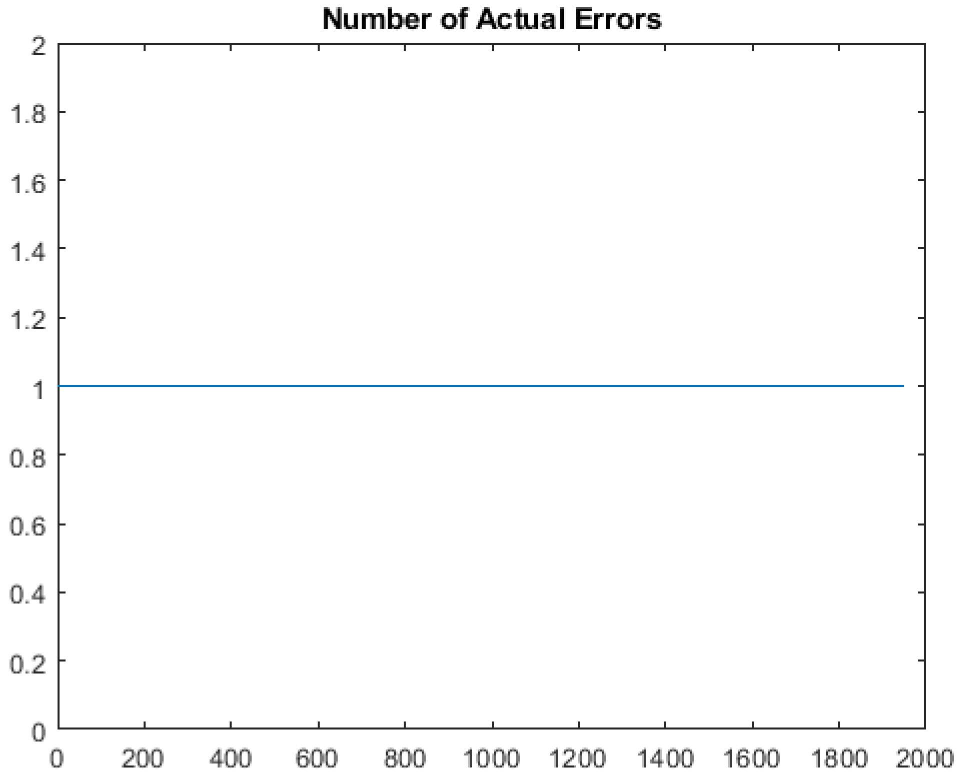
```
cnumerr2 = zeros(nchoosek(n,2),1);
nErrs = zeros(nchoosek(n,2),1);
cnumerrIdx = 1;
for idx1 = 1 : n-1
    %sprintf('idx1 for 2 errors = %d', idx1)
    for idx2 = idx1+1 : n
        errors = zeros(1,n);
        errors(idx1) = 1;
        errors(idx2) = 1;
        erroredCode = code + gf(errors);
        [decoded2, cnumerr2(cnumerrIdx)] ...
          = bchdec(erroredCode,n,k);
```

Encode the decoded message. Check that the re-encoded message differs from the errored message in only one bit.

```
        if cnumerr2(cnumerrIdx) == 1
            code2 = bchenc(decoded2,n,k);
            nErrs(cnumerrIdx) = biterr(double(erroredCode.x), ...
              double(code2.x));
        end
        cnumerrIdx = cnumerrIdx + 1;
    end
end
```

Plot the computed number of errors, based on the difference between the doubly-errored code and the re-encoded version of the initial decoding.

```
plot(nErrs)
title('Number of Actual Errors')
```



All inputs with two errors were decoded to a codeword that differs in exactly one bit from the re-encoded version.

Decode Received BCH Codeword in Noisy Channel

Set the BCH parameters for a Galois array of GF(2).

```
M = 4;
n = 2^M-1;   % Codeword length
k = 5;       % Message length
nwords = 10; % Number of words to encode
```

Create a message.

```
msgTx = gf(randi([0 1],nwords,k));
```

Find the error-correction capability.

```
t = bchnumerr(n,k)
```

```
t = 3
```

Encode the message.

```
enc = bchenc(msgTx,n,k);
```

Corrupt up to t bits in each codeword.

```
noisycode = enc + randerr(nwords,n,1:t);
```

Decode the noisy code.

```
msgRx = bchdec(noisycode,n,k);
```

Validate that the message was properly decoded.

```
isequal(msgTx,msgRx)
```

```
ans = logical
   1
```

Increase the number of possible errors, and generate another noisy codeword.

```
t2 = t + 1;
noisycode2 = enc + randerr(nwords,n,1:t2);
```

Decode the new received codeword.

```
[msgRx2,numerr] = bchdec(noisycode2,n,k);
```

Determine if the message was properly decoded by examining the number of corrected errors, `numerr`. Entries of `-1` correspond to decoding failures, which occur when the codeword has more errors than can be corrected for the specified `[n,k]` pair.

```
numerr
```

```
numerr = 10×1

     1
     2
    -1
     2
     3
     1
    -1
     4
     2
     3
```

Two of the ten transmitted codewords were not correctly received.

Input Arguments

code — Encoded message
Galois field array

Encoded message, specified as a Galois field array of symbols over GF(2). Each `N`-element row of `code` represents a corrupted systematic codeword.

For more information, see “Creating a Galois field array”.

N — Codeword length

integer

Codeword length, specified as an integer of the form N = 2^M–1, where M is an integer from 3 to 16. See “Tips” on page 2-49 for information about valid N values, valid (N,K) pairs, and error correcting capabilities for a given BCH code.

Example: 15 for *M*=4

K — Message length

integer

Message length, specified as an integer. N and K must produce a narrow-sense BCH code.

Example: 5 specifies a Galois field array with five elements.

paritypos — Parity position

'end' (default) | 'beginning'

Parity position, specified as 'end' or 'beginning'. Parity symbols are at the end or beginning of each word in the output Galois field array. If paritypos is 'beginning', then a decoding failure causes bchdec to remove N-K symbols from the beginning rather than the end of the row.

Output Arguments

decoded — Decoded message

Galois field array of symbols over GF(2)

Decoded message, returned as a Galois field array of symbols over GF(2). Each row represents the attempt at decoding the corresponding row in code. A decoding failure occurs if bchdec detects more than *T* errors in a row of code, where *T* is the number of errors per codeword that the decoder is capable of correcting. When a decoding failure occurs, bchdec forms the corresponding row of decoded by removing N-K symbols from the end of the row of code. For more information, see “Error Correcting Capability” on page 2-48.

cnumerr — Number of corrected errors

column vector

Number of corrected errors in the corresponding row of code, returned as a column vector. A value of –1 in cnumerr indicates a decoding failure in that row in code.

ccode — Corrected version of code

Galois field array

Corrected version of code, returned as a Galois field array. ccode has the same format as the input code. If a decoding failure occurs in a certain row of code, the corresponding row in ccode contains that row unchanged.

More About

Error Correcting Capability

BCH decoders correct up to a specified number of errors per codeword based on the (N,K) pair used to BCH encode that message. The error correcting capability, *T*, of a given (N,K) pair is returned by bchnumerr. See “Tips” on page 2-49 for information about valid N values, valid (N,K) pairs, and error correcting capabilities for a given BCH code.

If the coded message contains more errors per codeword than the decoder is capable of correcting, the decoder is unlikely to decode to the correct codeword. For example, when a single-error-correcting BCH decoder (*T*=1) is given an input with two errors per codeword, it decodes it to a valid codeword but not the correct codeword. When a double-error-correcting BCH decoder (*T*=2) is given an input with three errors per codeword, the decoder sometimes decodes to an invalid codeword. The cnumerr and ccode output provide feedback to analyze the correctness of the decoded message.

Tips

- To generate the list of valid (N,K) pairs along with the corresponding values of the error-correction capability, run bchnumerr(N).
- Valid values for N = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of N is 65,535.

Algorithms

bchdec uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” on page 2-49.

References

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

[2] Berlekamp, Elwyn R. *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

See Also

Functions
bchenc | bchgenpoly | bchnumerr

Objects
comm.BCHDecoder

Topics
“Block Codes”

Introduced before R2006a

bchenc

BCH encoder

Syntax

```
code = bchenc(msg,N,K)
code = bchenc(msg,N,K,paritypos)
```

Description

`code = bchenc(msg,N,K)` encodes the input message using an (`N`,`K`) BCH encoder that uses a narrow-sense generator polynomial. For a description of Bose–Chaudhuri–Hocquenghem (BCH) coding, see [1].

`code = bchenc(msg,N,K,paritypos)` appends or prepends the parity symbols to the encoded input message to form the output.

Examples

Decode Received BCH Codeword in Noisy Channel

Set the BCH parameters for a Galois array of GF(2).

```
M = 4;
n = 2^M-1;   % Codeword length
k = 5;       % Message length
nwords = 10; % Number of words to encode
```

Create a message.

```
msgTx = gf(randi([0 1],nwords,k));
```

Find the error-correction capability.

```
t = bchnumerr(n,k)
```

```
t = 3
```

Encode the message.

```
enc = bchenc(msgTx,n,k);
```

Corrupt up to `t` bits in each codeword.

```
noisycode = enc + randerr(nwords,n,1:t);
```

Decode the noisy code.

```
msgRx = bchdec(noisycode,n,k);
```

Validate that the message was properly decoded.

```
isequal(msgTx,msgRx)
```

```
ans = logical
   1
```

Increase the number of possible errors, and generate another noisy codeword.

```
t2 = t + 1;
noisycode2 = enc + randerr(nwords,n,1:t2);
```

Decode the new received codeword.

```
[msgRx2,numerr] = bchdec(noisycode2,n,k);
```

Determine if the message was properly decoded by examining the number of corrected errors, numerr. Entries of -1 correspond to decoding failures, which occur when the codeword has more errors than can be corrected for the specified [n,k] pair.

```
numerr
```

```
numerr = 10×1

     1
     2
    -1
     2
     3
     1
    -1
     4
     2
     3
```

Two of the ten transmitted codewords were not correctly received.

Input Arguments

msg — Message to encode

Galois field array of symbols over GF(2)

Message to encode, specified as a Galois field array of symbols over GF(2). Each K-element row of msg represents a message word, where the leftmost symbol is the most significant symbol.

For more information, see “Creating a Galois field array”.

Example: msgTx = gf(randi([0 1],10,5)), where msgTx is a 10-by-5 gf array.

N — Codeword length

integer

Codeword length, specified as an integer of the form N = 2^M–1, where *M* is an integer from 3 through 16. For more information, see “Tips” on page 2-52.

Example: 15 for *M*=4

K — Message length

integer

Message length, specified as an integer. N and K must produce a narrow-sense BCH code.

Example: 5 specifies a Galois array with five elements

paritypos — Parity position

'end' (default) | 'beginning'

Parity position, specified as 'end' or 'beginning'. Parity symbols are at the end or beginning of each word in the output Galois array.

Output Arguments

code — Encoded message

Galois field array

Encoded message, returned as a Galois field array. Parity symbols are at the end or beginning of each word in the output Galois array. To specify the position of the parity symbols, use the paritypos argument.

Tips

- To generate the list of valid (N,K) pairs along with the corresponding values of the error-correction capability, run bchnumerr(N).
- Valid values for N = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of N is 65,535.

References

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

See Also

Functions

bchdec | bchgenpoly | bchnumerr | gf

Objects

comm.BCHEncoder

Topics

“Block Codes”
“Galois Field Computations”
“How Integers Correspond to Galois Field Elements”

Introduced before R2006a

bchgenpoly

Generator polynomial of BCH code

Syntax

```
genpoly = bchgenpoly(n,k)
genpoly = bchgenpoly(n,k,prim_poly)
genpoly = bchgenpoly(n,k,prim_poly,outputFormat)
[genpoly,t] = bchgenpoly(...)
```

Description

`genpoly = bchgenpoly(n,k)` returns the narrow-sense generator polynomial of a BCH code with codeword length `n` and message length `k`. The codeword length `n` must have the form 2^m-1 for some integer m between 3 and 16. 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 LCM[m_1(x), m_2(x), ..., m_2t(x)], where:

- LCM represents the least common multiple,
- m_i(x) represents the minimum polynomial corresponding to α^i, α is a root of the default primitive polynomial for the field GF(`n+1`),
- and t represents the error-correcting capability of the code.

Note Although the `bchgenpoly` function performs intermediate computations in GF(`n+1`), the final polynomial has binary coefficients. The output from `bchgenpoly` is a Galois vector in GF(2) rather than in GF(`n+1`).

`genpoly = bchgenpoly(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 Alpha as a root. `prim_poly` is either a polynomial character vector or an integer whose binary representation indicates the coefficients of the primitive polynomial in order of descending powers. To use the default primitive polynomial for GF(`n+1`), set `prim_poly` to `[]`.

`genpoly = bchgenpoly(n,k,prim_poly,outputFormat)` is the same as the previous syntax, except that `outputFormat` specifies output data type. The value of `outputFormat` can be ‘`gf`' or '`double`' corresponding to Galois field and double data types respectively. The default value of `outputFormat` is '`gf`'.

`[genpoly,t] = bchgenpoly(...)` returns `t`, the error-correction capability of the code.

Examples

Create a BCH Generator Polynomial

Create two BCH generator polynomials based on different primitive polynomials.

Set the codeword and message lengths, `n` and `k`.

```
n = 15;
k = 11;
```

Create the generator polynomial and return the error correction capability, `t`.

```
[genpoly,t] = bchgenpoly(15,11)
```

```
genpoly = GF(2) array.

Array elements =

   1   0   0   1   1

t = 1
```

Create a generator polynomial for a (15,11) BCH code using a different primitive polynomial expressed as a character vector. Note that `genpoly2` differs from `genpoly`, which uses the default primitive.

```
genpoly2 = bchgenpoly(15,11,'D^4 + D^3 + 1')
```

```
genpoly2 = GF(2) array.

Array elements =

   1   1   0   0   1
```

Limitations

The maximum allowable value of `n` is 65535.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`bchdec` | `bchenc` | `bchnumerr`

Topics

“Block Codes”

Introduced before R2006a

bchnumerr

Number of correctable errors for BCH code

Syntax

```
T = bchnumerr(N)
T = bchnumerr(N, K)
```

Description

`T = bchnumerr(N)` returns all the possible combinations of message length, `K`, and number of correctable errors, `T`, for a BCH code of codeword length, `N`.

`T = bchnumerr(N, K)` returns the number of correctable errors, `T`, for an `(N, K)` BCH code.

Examples

Determine Message Length Combinations for BCH Code

Calculate the possible message length combinations for a BCH code word length of `15`.

```
T = bchnumerr(15)
```

```
T = 3×3

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

Compute the Correctable Errors for BCH Code

Calculate the number of correctable errors for BCH code `15,11`

```
T = bchnumerr(15,11)
```

```
T = 1
```

Input Arguments

N — Codeword length
integer scalar

Codeword length, specified as an integer scalar. `N` must have the form 2^m-1 for some integer, m, between 3 and 16.

Example: 15

Data Types: `double`

K — Message length

integer scalar

Message length, specified as an integer scalar. `N` and `K` must produce a narrow-sense BCH code.

Example: 11

Data Types: `double`

Output Arguments

T — Number of correctable errors

scalar or matrix

Number of correctable errors, returned as a scalar or matrix value.

`bchnumerr(N)` returns a matrix with three columns. The first column lists `N`, the second column lists `K`, and the third column lists `T`.

`bchnumerr(N,K)` returns a scalar, which represents the number of correctable errors for the BCH code.

See Also

`bchdec` | `bchenc`

Topics

"Block Codes"
"BCH Codes"

Introduced before R2006a

berawgn

BER and SER for uncoded AWGN channels

Syntax

```
ber = berawgn(EbNo,modtype,M)

ber = berawgn(EbNo,'psk',M,dataenc)
ber = berawgn(EbNo,'oqpsk',dataenc)

ber = berawgn(EbNo,'fsk',M,coherence)
ber = berawgn(EbNo,'fsk',M,coherence,rho)

ber = berawgn(EbNo,'msk',precoding)
ber = berawgn(EbNo,'msk',precoding,coherence)

ber = berawgn(EbNo,'cpfsk',M,modindex,kmin)

[ber,ser] = berawgn( ___ )
```

Description

The berawgn function returns the bit error rate (BER) and symbol error rate (SER) in an additive white Gaussian noise (AWGN) channel for uncoded data using various modulation schemes. The first input argument, `EbNo`, is the ratio of bit energy to noise power spectral density (E_b/N_0) and its units are in dB. Values in the output `ber` and `ser` vectors correspond to the theoretical error rate at the specified E_b/N_0 levels for a Gray-coded signal constellation. For more information, see Analytical Expressions Used in berawgn.

`ber = berawgn(EbNo,modtype,M)` returns the BER at the specified E_b/N_0 levels for the modulation type and modulation order specified by `modtype` and `M`, respectively.

`ber = berawgn(EbNo,'psk',M,dataenc)` specifies the data encoding type as differential or nondifferential for PSK modulation.

`ber = berawgn(EbNo,'oqpsk',dataenc)` specifies the data encoding type as differential or nondifferential for OQPSK modulation.

`ber = berawgn(EbNo,'fsk',M,coherence)` specifies the receiver technique as coherent or noncoherent for FSK modulation.

`ber = berawgn(EbNo,'fsk',M,coherence,rho)` additionally specifies the complex correlation coefficient of the FSK-modulated signal.

`ber = berawgn(EbNo,'msk',precoding)` specifies whether precoding is applied for MSK modulation.

`ber = berawgn(EbNo,'msk',precoding,coherence)` additionally specifies the receiver technique as coherent or noncoherent for MSK modulation.

`ber = berawgn(EbNo,'cpfsk',M,modindex,kmin)` specifies the modulation index, `modindex`, and the number of paths having the minimum distance, `kmin`, for CPFSK modulation.

`[ber,ser] = berawgn( ___ )` returns the BER and symbol error rate (SER) using any input argument combination from previous syntaxes.

Examples

Return Theoretical BER Data for AWGN Channels

Return theoretical bit error rate data for several modulation schemes in an AWGN channel.

Create a vector of E_b/N_0 values and specify the modulation order.

```
EbNo = (0:10)';
M = 4; % Modulation order
```

Return theoretical BER data for QPSK modulation.

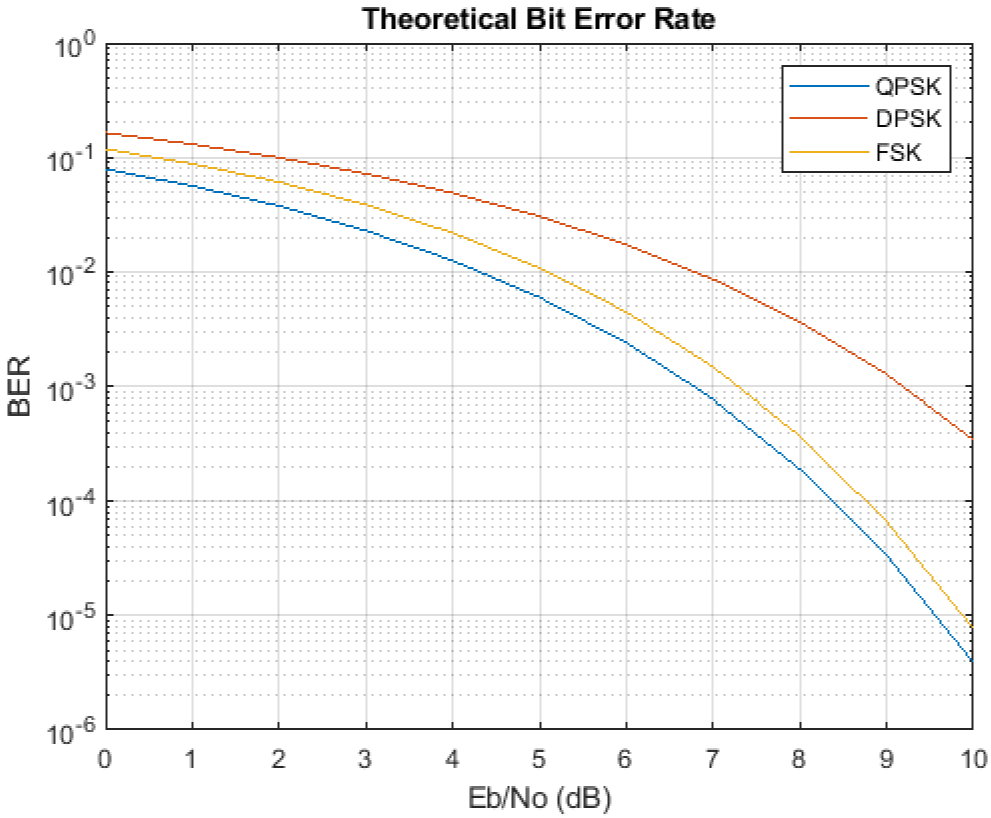
```
berQ = berawgn(EbNo,'psk',M,'nondiff');
```

Return equivalent data for DPSK and FSK modulations.

```
berD = berawgn(EbNo,'dpsk',M);
berF = berawgn(EbNo,'fsk',M,'coherent');
```

Plot the results.

```
semilogy(EbNo,[berQ berD berF])
xlabel('Eb/No (dB)')
ylabel('BER')
legend('QPSK','DPSK','FSK')
title("Theoretical Bit Error Rate")
grid
```



Input Arguments

EbNo — Energy per bit to noise power spectral density ratio

scalar | vector

Energy per bit to noise power spectral density ratio in dB, specified as a scalar or vector.

Data Types: `single` | `double`

`modtype` — Modulation type

`'psk'` | `'oqpsk'` | `'dpsk'` | ...

Modulation type, specified as one of these options.

`modtype` Value	Modulation Scheme	Dependencies
`'psk'`	Phase shift keying (PSK)	When you set the input `dataenc` to `'diff'`, modulation order `M` must be 2 or 4.
`'oqpsk'`	Offset quadrature phase shift keying (OQPSK)	None
`'dpsk'`	Differential phase shift keying (DPSK)	None

modtype Value	Modulation Scheme	Dependencies
'pam'	Pulse amplitude modulation (PAM)	None
'qam'	Quadrature amplitude modulation (QAM)	The modulation order M must be at least 4. • When $k = log_2$M is odd, the symbols lie in a rectangular constellation of size M = $I \times J$, where $I = 2^{\frac{k-1}{2}}$ and $J = 2^{\frac{k+1}{2}}$. • When k is even, the symbols lie in a square constellation of size$2^{\frac{k}{2}} \times 2^{\frac{k}{2}}$.
'fsk'	Frequency-shift keying (FSK)	When you set the input coherence to 'noncoherent', modulation order M must be in the range [2, 64].
'msk'	Minimum-shift keying (MSK)	None
'cpfsk'	Continuous-phase frequency-shift keying (CPFSK)	None

Data Types: char | string

M — Modulation order
2^k

Modulation order, specified as an integer equal to 2^k, where k is a positive integer.

Example: 4 or 2^2

Data Types: single | double

dataenc — Data encoding type
'diff' | 'nondiff'

Data encoding type, specified as one of these values.

- 'diff' — For differential data encoding
- 'nondiff' — For nondifferential data encoding

Dependencies

To enable this argument, set the modtype argument to 'psk' or 'oqpsk'.

Data Types: char | string

coherence — Coherent detection type
'conherent' | 'noncoherent'

Coherent detection type, specified as one of these values.

- 'conherent' — For coherent detection
- 'noncoherent' — For noncoherent detection

Dependencies

To enable this argument, set the modtype argument to 'fsk' or 'msk'.

Data Types: char | string

rho — Complex correlation coefficient

complex scalar

Complex correlation coefficient, specified as a complex scalar. For more information about the complex correlation coefficient and how to compute it for nonorthogonal binary frequency-shift keying (BFSK) modulation, see “Nonorthogonal 2-FSK with Coherent Detection”.

Dependencies

To enable this argument, set the modtype argument to 'fsk' and the M argument to 2.

Data Types: single | double
Complex Number Support: Yes

precoding — Enable precoding

'off' | 'on'

Enable precoding, specified as one of these values.

- 'off' — For conventional MSK
- 'on' — For precoded MSK

Dependencies

To enable this argument, set the modtype argument to 'msk'.

Data Types: char | string

modindex — Modulation index

positive integer

Modulation index, specified as a positive integer.

Dependencies

To enable this argument, set the modtype argument to 'cpfsk'.

Data Types: single | double

kmin — Number of paths having minimum distance

positive integer

Number of paths having the minimum distance, specified as a positive integer. If the number of paths is unknown, specify a value of 1.

Dependencies

To enable this argument, set the modtype argument to 'cpfsk'.

Data Types: `single` | `double`

Output Arguments

`ber` — Bit error rate

scalar | vector

Bit error rate (BER) for uncoded data over an AWGN channel, returned as a scalar or vector. The BER is computed for each E_b/N_0 setting specified by input `EbNo` according to the modulation type specified by input `modtype` and related dependencies.

Data Types: `double`

`ser` — Symbol error rate

scalar | vector

Symbol error rate (SER) for uncoded data over an AWGN channel, returned as a scalar or vector. The SER is computed for each E_b/N_0 setting specified by input `EbNo` according to the modulation type specified by input `modtype` and related dependencies.

Data Types: `double`

Limitations

The numerical accuracy of the output returned by this function is limited by approximations related to the numerical implementation of the expressions to roughly two significant digits.

Alternatives

As an alternative to the `berawgn` function, use the **BER Analyzer** app and configure settings in the **Theoretical** tab.

References

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

[2] Cho, K., and D. Yoon. “On the General BER Expression of One- and Two-Dimensional Amplitude Modulations.” *IEEE Trans. Commun.* 50, no. 7, (2002): 1074-1080.

[3] Lee, P. J. “Computation of the Bit Error Rate of Coherent M-ary PSK with Gray Code Bit Mapping.” *IEEE Trans. Commun.* COM-34, no. 5, (1986): 488-491.

[4] Proakis, John G. *Digital Communications* 4th ed. New York: McGraw Hill, 2001.

[5] Simon, M. K, S. M. Hinedi, and W. C. Lindsey. *Digital Communication Techniques – Signal Design and Detection*. Prentice-Hall, 1995.

[6] Simon, M. K. “On the Bit-Error Probability of Differentially Encoded QPSK and Offset QPSK in the Presence of Carrier Synchronization.” *IEEE Trans. Commun.*. 54, (2006): 806-812.

[7] Lindsey, W. C., and M. K. Simon. *Telecommunication Systems Engineering*. Englewood Cliffs, N.J: Prentice-Hall, 1973.

See Also

Apps
BER Analyzer

Functions
`bercoding` | `berfading` | `bersync` | `bertool`

Topics
"Theoretical Results"
Analytical Expressions Used in berawgn

Introduced before R2006a

bercoding

Bit error rate (BER) for coded AWGN channels

Syntax

```
berub = bercoding(EbNo,'conv',decision,coderate,dspec)
berub = bercoding(EbNo,'block','hard',n,k,dmin)
berub = bercoding(EbNo,'block','soft',n,k,dmin)
berapprox = bercoding(EbNo,'Hamming','hard',n)
berub = bercoding(EbNo,'Golay','hard',24)
berapprox = bercoding(EbNo,'RS','hard',n,k)
berapprox = bercoding(...,modulation)
```

Description

`berub = bercoding(EbNo,'conv',`*`decision`*`,coderate,dspec)` returns an upper bound or approximation on the BER of a binary convolutional code with coherent phase shift keying (PSK) modulation over an additive white Gaussian noise (AWGN) channel. `EbNo` is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, `berub` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. To specify hard-decision decoding, set *`decision`* to `'hard'`; to specify soft-decision decoding, set *`decision`* to `'soft'`. The convolutional code has code rate equal to `coderate`. The `dspec` input is a structure that contains information about the code's distance spectrum:

- `dspec.dfree` is the minimum free distance of the code.
- `dspec.weight` is the weight spectrum of the code.

To find distance spectra for some sample codes, use the `distspec` function or see [5] and [3].

Note The results for binary PSK and quadrature PSK modulation are the same. This function does not support M-ary PSK when M is other than 2 or 4.

`berub = bercoding(EbNo,'block','hard',n,k,dmin)` returns an upper bound on the BER of an [`n`,`k`] binary block code with hard-decision decoding and coherent BPSK or QPSK modulation. `dmin` is the minimum distance of the code.

`berub = bercoding(EbNo,'block','soft',n,k,dmin)` returns an upper bound on the BER of an [`n`,`k`] binary block code with soft-decision decoding and coherent BPSK or QPSK modulation. `dmin` is the minimum distance of the code.

`berapprox = bercoding(EbNo,'Hamming','hard',n)` returns an approximation of the BER of a Hamming code using hard-decision decoding and coherent BPSK modulation. (For a Hamming code, if n is known, then k can be computed directly from n.)

`berub = bercoding(EbNo,'Golay','hard',24)` returns an upper bound of the BER of a Golay code using hard-decision decoding and coherent BPSK modulation. Support for Golay currently is only for n=24. In accordance with [3], the Golay coding upper bound assumes only the correction of

3-error patterns. Even though it is theoretically possible to correct approximately 19% of 4-error patterns, most decoders in practice do not have this capability.

`berapprox = bercoding(EbNo,'RS','hard',n,k)` returns an approximation of the BER of (n,k) Reed-Solomon code using hard-decision decoding and coherent BPSK modulation.

`berapprox = bercoding(...,`*`modulation`*`)` returns an approximation of the BER for coded AWGN channels when you specify a *`modulation`* type. See the `berawgn` function for a listing of the supported modulation types.

Examples

Upper Bound on Theoretical BER for a Block Code

Find an upper bound on the theoretical BER of a (23,12) block code.

Set the example parameters.

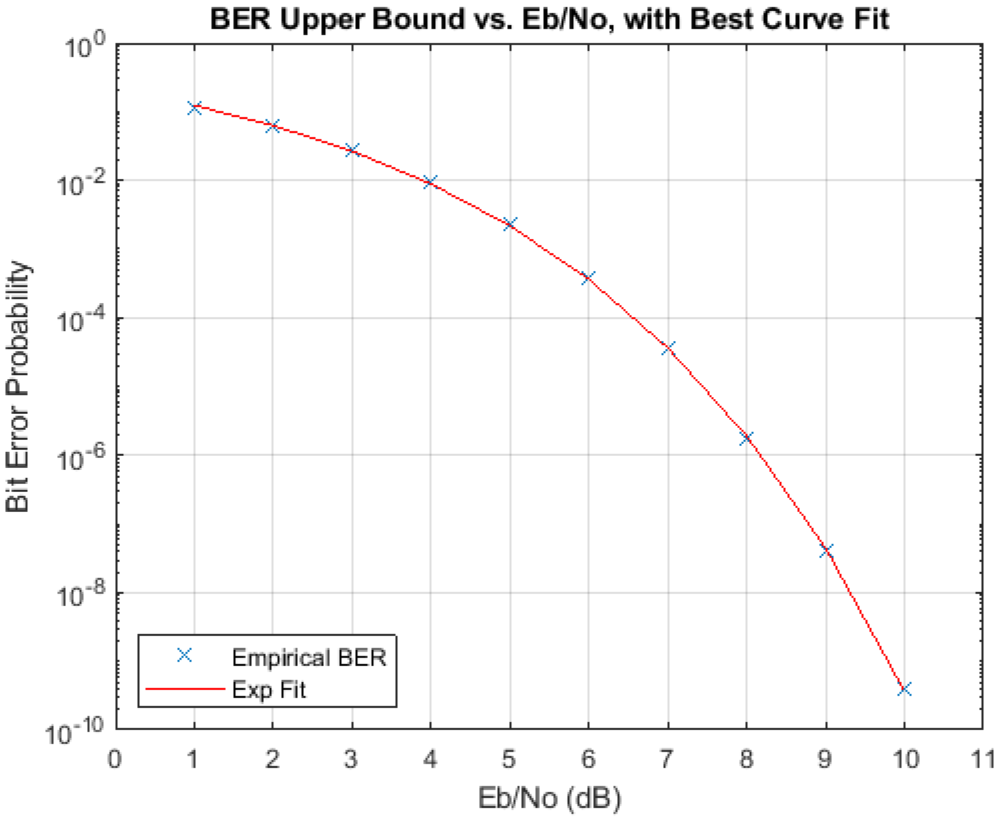
```
n = 23;             % Codeword length
k = 12;             % Message length
dmin = 7;           % Minimum distance
EbNo = 1:10;        % Eb/No range (dB)
```

Estimate the BER.

```
berBlk = bercoding(EbNo,'block','hard',n,k,dmin);
```

Plot the estimated BER.

```
berfit(EbNo,berBlk)
ylabel('Bit Error Probability')
title('BER Upper Bound vs. Eb/No, with Best Curve Fit')
```



Estimate Coded BER Performance of 16-QAM in AWGN

Estimate the performance of a 16-QAM channel in AWGN when encoded with a (15,11) Reed-Solomon code using hard-decision decoding.

Set the input Eb/No range and determine the uncoded BER for 16-QAM.

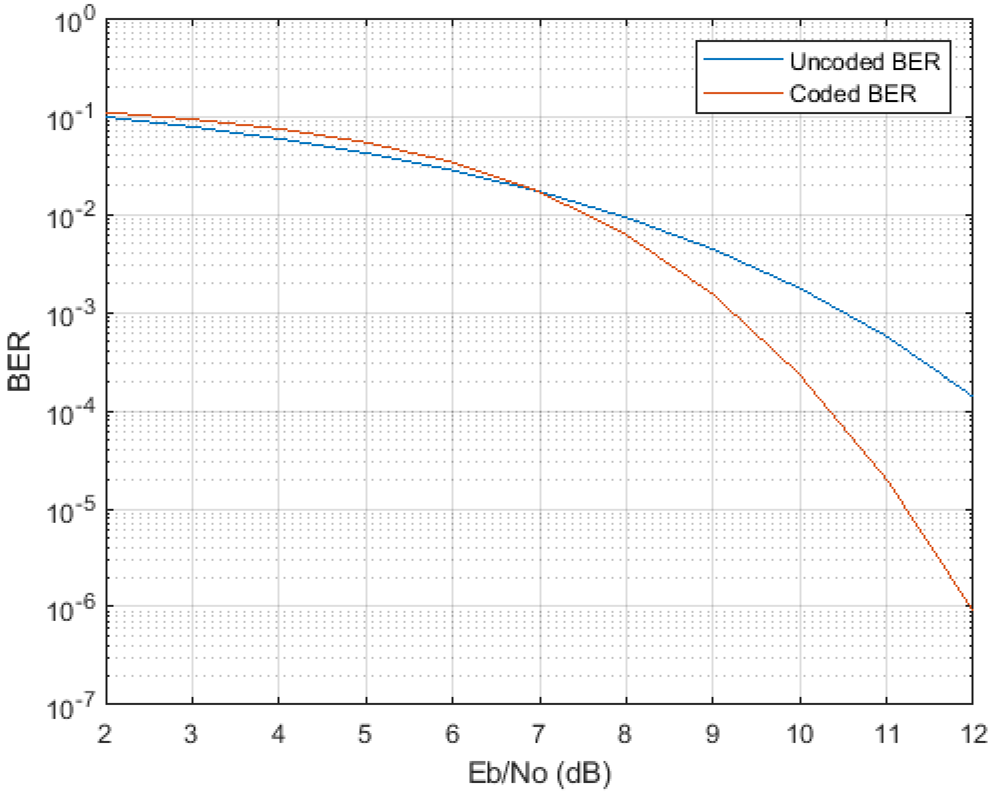
```
ebno = (2:12)';
uncodedBER = berawgn(ebno,'qam',16);
```

Estimate the coded BER for 16-QAM channel with a (15,11) Reed-Solomon code using hard decision decoding.

```
codedBER = bercoding(ebno,'RS','hard',15,11,'qam',16);
```

Plot the estimated BER curves.

```
semilogy(ebno,[uncodedBER codedBER])
grid
legend('Uncoded BER','Coded BER')
xlabel('Eb/No (dB)')
ylabel('BER')
```



Limitations

The numerical accuracy of this function's output is limited by

- Approximations in the analysis leading to the closed-form expressions that the function uses
- Approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

Alternatives

As an alternative to the `bercoding` function, invoke the BERTool GUI (`bertool`) and use the **Theoretical** tab.

References

[1] Proakis, J. G., *Digital Communications*, 4th ed., New York, McGraw-Hill, 2001.

[2] Frenger, P., P. Orten, and T. Ottosson, “Convolutional Codes with Optimum Distance Spectrum,” *IEEE Communications Letters*, Vol. 3, No. 11, Nov. 1999, pp. 317–319.

[3] Odenwalder, J. P., *Error Control Coding Handbook*, Final Report, LINKABIT Corporation, San Diego, CA, 1976.

[4] Sklar, B., *Digital Communications*, 2nd ed., Prentice Hall, 2001.

[5] Ziemer, R. E., and R. L., Peterson, *Introduction to Digital Communication*, 2nd ed., Prentice Hall, 2001.

See Also

`berawgn` | `berfading` | `bersync` | `bertool` | `distspec`

Topics

"Theoretical Performance Results"
Analytical Expressions Used in bercoding and BERTool

Introduced before R2006a

berconfint

Bit error rate (BER) and confidence interval of Monte Carlo simulation

Syntax

```
[ber,interval] = berconfint(nerrs,ntrials)
[ber,interval] = berconfint(nerrs,ntrials,level)
```

Description

`[ber,interval] = berconfint(nerrs,ntrials)` returns the error probability estimate `ber` and the 95% confidence interval `interval` for a Monte Carlo simulation of `ntrials` trials with `nerrs` errors. `interval` is a two-element vector that lists the endpoints of the interval. If the errors and trials are measured in bits, the error probability is the bit error rate (BER); if the errors and trials are measured in symbols, the error probability is the symbol error rate (SER).

`[ber,interval] = berconfint(nerrs,ntrials,level)` specifies the confidence level as a real number between 0 and 1.

Examples

If a simulation of a communication system results in 100 bit errors in 10^6 trials, the BER (bit error rate) for that simulation is the quotient 10^{-4}. The command below finds the 95% confidence interval for the BER of the system.

```
nerrs = 100; % Number of bit errors in simulation
ntrials = 10^6; % Number of trials in simulation
level = 0.95; % Confidence level
[ber,interval] = berconfint(nerrs,ntrials,level)
```

The output below shows that, with 95% confidence, the BER for the system is between 0.0000814 and 0.0001216.

```
ber =

  1.0000e-004


interval =

  1.0e-003 *

    0.0814    0.1216
```

For an example that uses the output of `berconfint` to plot error bars on a BER plot, see “Curve Fitting An Error Rate Plot”

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

Introduced before R2006a

berfading

Bit error rate (BER) for Rayleigh and Rician fading channels

Syntax

```
ber = berfading(EbNo,'pam',M,divorder)
ber = berfading(EbNo,'qam',M,divorder)
ber = berfading(EbNo,'psk',M,divorder)
ber = berfading(EbNo,'depsk',M,divorder)
ber = berfading(EbNo,'oqpsk',divorder)
ber = berfading(EbNo,'dpsk',M,divorder)
ber = berfading(EbNo,'fsk',M,divorder,coherence)
ber = berfading(EbNo,'fsk',2,divorder,coherence,rho)
ber = berfading(EbNo,...,K)
ber = berfading(EbNo,'psk',2,1,K,phaserr)
[BER,SER] = berfading(EbNo, ...)
```

Description

For All Syntaxes

The first input argument, `EbNo`, is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels.

Most syntaxes also have an `M` input that specifies the alphabet size for the modulation. `M` must have the form 2^k for some positive integer `k`.

`berfading` uses expressions that assume Gray coding. If you use binary coding, the results may differ.

For cases where diversity is used, the E_b/N_0 on each diversity branch is `EbNo/divorder`, where `divorder` is the diversity order (the number of diversity branches) and is a positive integer.

For Specific Syntaxes

`ber = berfading(EbNo,'pam',M,divorder)` returns the BER for PAM over an uncoded Rayleigh fading channel with coherent demodulation.

`ber = berfading(EbNo,'qam',M,divorder)` returns the BER for QAM over an uncoded Rayleigh fading channel with coherent demodulation. The alphabet size, `M`, must be at least 4. When $k = \log_2 M$ is odd, a rectangular constellation of size $M = I \times J$ is used, where $I = 2^{\frac{k-1}{2}}$ and $J = 2^{\frac{k+1}{2}}$.

`ber = berfading(EbNo,'psk',M,divorder)` returns the BER for coherently detected PSK over an uncoded Rayleigh fading channel.

`ber = berfading(EbNo,'depsk',M,divorder)` returns the BER for coherently detected PSK with differential data encoding over an uncoded Rayleigh fading channel. Only $M = 2$ is currently supported.

`ber = berfading(EbNo,'oqpsk',divorder)` returns the BER of coherently detected offset-QPSK over an uncoded Rayleigh fading channel.

`ber = berfading(EbNo,'dpsk',M,divorder)` returns the BER for DPSK over an uncoded Rayleigh fading channel. For DPSK, it is assumed that the fading is slow enough that two consecutive symbols are affected by the same fading coefficient.

`ber = berfading(EbNo,'fsk',M,divorder,coherence)` returns the BER for orthogonal FSK over an uncoded Rayleigh fading channel. `coherence` should be `'coherent'` for coherent detection, or `'noncoherent'` for noncoherent detection.

`ber = berfading(EbNo,'fsk',2,divorder,coherence,rho)` returns the BER for binary nonorthogonal FSK over an uncoded Rayleigh fading channel. `rho` is the complex correlation coefficient. See “Nonorthogonal 2-FSK with Coherent Detection” for the definition of the complex correlation coefficient and how to compute it for nonorthogonal BFSK.

`ber = berfading(EbNo,...,K)` returns the BER over an uncoded Rician fading channel, where `K` is the ratio of specular to diffuse energy in linear scale. For the case of `'fsk'`, `rho` must be specified before `K`.

`ber = berfading(EbNo,'psk',2,1,K,phaserr)` returns the BER of BPSK over an uncoded Rician fading channel with imperfect phase synchronization. `phaserr` is the standard deviation of the reference carrier phase error in radians.

`[BER,SER] = berfading(EbNo, ...)` returns both the BER and SER.

Examples

Estimate BER Performance of 16-QAM in Fading

Generate a vector of Eb/No values to evaluate.

```
EbNo = 8:2:20;
```

Initialize the BER results vector.

```
ber = zeros(length(EbNo),20);
```

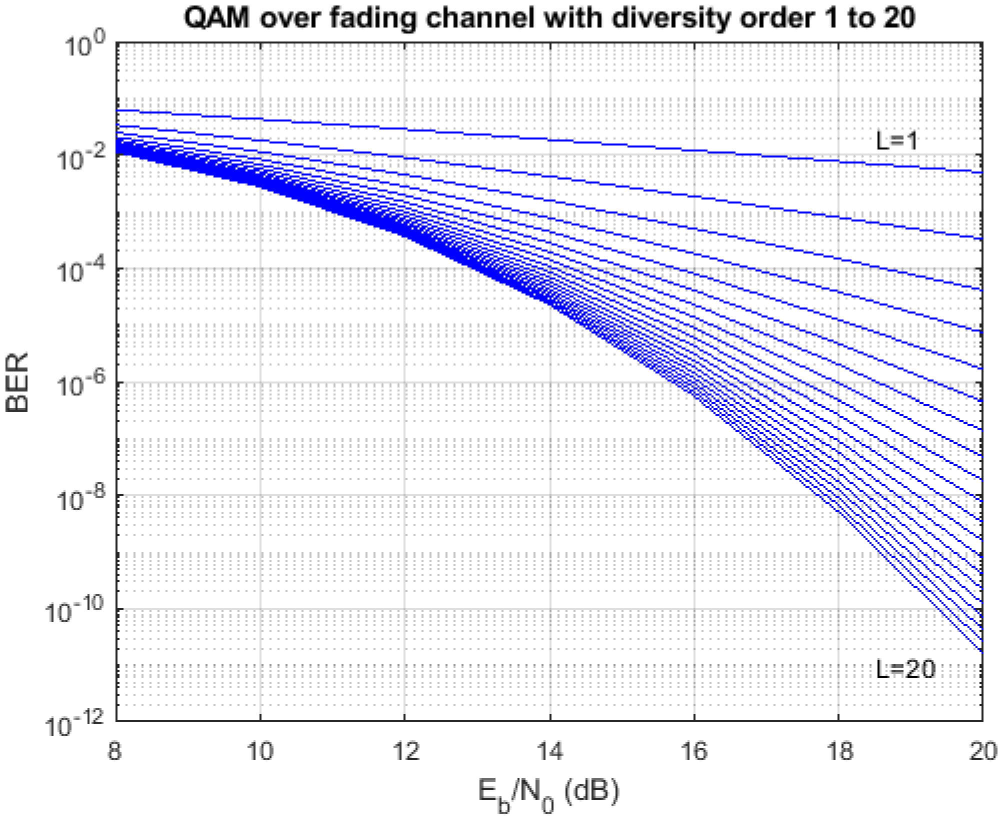
Generate BER vs. Eb/No curves for 16-QAM in a fading channel. Vary the diversity order from 1 to 20.

```
for L = 1:20
    ber(:,L) = berfading(EbNo,'qam',16,L);
end
```

Plot the results.

```
semilogy(EbNo,ber,'b')
text(18.5, 0.02, sprintf('L=%d',1))
text(18.5, 1e-11, sprintf('L=%d',20))
title('QAM over fading channel with diversity order 1 to 20')
xlabel('E_b/N_0 (dB)')
```

```
ylabel('BER')
grid on
```



Limitations

The numerical accuracy of this function's output is limited by approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

Alternatives

As an alternative to the `berfading` function, invoke the BERTool GUI (`bertool`), and use the **Theoretical** tab.

References

[1] Proakis, John G., *Digital Communications*, 4th ed., New York, McGraw-Hill, 2001.

[2] Modestino, James W., and Mui, Shou Y., *Convolutional code performance in the Rician fading channel*, IEEE Trans. Commun., 1976.

[3] Cho, K., and Yoon, D., “On the general BER expression of one- and two-dimensional amplitude modulations”, *IEEE Trans. Commun.*, Vol. 50, Number 7, pp. 1074-1080, 2002.

[4] Lee, P. J., “Computation of the bit error rate of coherent M-ary PSK with Gray code bit mapping”, *IEEE Trans. Commun.*, Vol. COM-34, Number 5, pp. 488-491, 1986.

[5] Lindsey, W. C., “Error probabilities for Rician fading multichannel reception of binary and N-ary signals”, *IEEE Trans. Inform. Theory*, Vol. IT-10, pp. 339-350, 1964.

[6] Simon, M. K , Hinedi, S. M., and Lindsey, W. C., *Digital Communication Techniques – Signal Design and Detection*, Prentice-Hall, 1995.

[7] Simon, M. K., and Alouini, M. S., *Digital Communication over Fading Channels – A Unified Approach to Performance Analysis*, 1st ed., Wiley, 2000.

[8] Simon, M. K , “On the bit-error probability of differentially encoded QPSK and offset QPSK in the presence of carrier synchronization”, *IEEE Trans. Commun.*, Vol. 54, pp. 806-812, 2006.

See Also

`berawgn` | `bercoding` | `bersync` | `bertool`

Topics

“Theoretical Performance Results”
Analytical Expressions Used in berfading

Introduced before R2006a

berfit

Fit curve to nonsmooth empirical bit error rate (BER) data

Syntax

```
fitber = berfit(empEbNo,empber)
fitber = berfit(empEbNo,empber,fitEbNo)
fitber = berfit(empEbNo,empber,fitEbNo,options)
fitber = berfit(empEbNo,empber,fitEbNo,options,fittype)
[fitber,fitprops] = berfit(...)
berfit(...)
berfit(empEbNo,empber,fitEbNo,options,'all')
```

Description

`fitber = berfit(empEbNo,empber)` fits a curve to the empirical BER data in the vector `empber` and returns a vector of fitted bit error rate (BER) points. The values in `empber` and `fitber` correspond to the E_b/N_0 values, in dB, given by `empEbNo`. The vector `empEbNo` must be in ascending order and must have at least four elements.

Note The `berfit` function is intended for curve fitting or interpolation, *not* extrapolation. Extrapolating BER data beyond an order of magnitude below the smallest empirical BER value is inherently unreliable.

`fitber = berfit(empEbNo,empber,fitEbNo)` fits a curve to the empirical BER data in the vector `empber` corresponding to the E_b/N_0 values, in dB, given by `empEbNo`. The function then evaluates the curve at the E_b/N_0 values, in dB, given by `fitEbNo` and returns the fitted BER points. The length of `fitEbNo` must equal or exceed that of `empEbNo`.

`fitber = berfit(empEbNo,empber,fitEbNo,options)` uses the structure `options` to override the default options used for optimization. These options are the ones used by the `fminsearch` function. You can create the `options` structure using the `optimset` function. Particularly relevant fields are described in the table below.

Field	Description
`options.Display`	Level of display: `'off'` (default) displays no output; `'iter'` displays output at each iteration; `'final'` displays only the final output; `'notify'` displays output only if the function does not converge.
`options.MaxFunEvals`	Maximum number of function evaluations before optimization ceases. The default is 10^4.
`options.MaxIter`	Maximum number of iterations before optimization ceases. The default is 10^4.

Field	Description
`options.TolFun`	Termination tolerance on the closed-form function used to generate the fit. The default is 10^{-4}.
`options.TolX`	Termination tolerance on the coefficient values of the closed-form function used to generate the fit. The default is 10^{-4}.

`fitber = berfit(empEbNo,empber,fitEbNo,options,`*`fittype`*`)` specifies which closed-form function `berfit` uses to fit the empirical data, from the possible fits listed in "Algorithms" on page 2-80 below. *`fittype`* can be `'exp'`, `'exp+const'`, `'polyRatio'`, or `'doubleExp+const'`. To avoid overriding default optimization options, use `options = []`.

`[fitber,fitprops] = berfit(...)` returns the MATLAB structure `fitprops`, which describes the results of the curve fit. Its fields are described in the table below.

Field	Description
`fitprops.fitType`	The closed-form function type used to generate the fit: `'exp'`, `'exp+const'`, `'polyRatio'`, or `'doubleExp+const'`.
`fitprops.coeffs`	The coefficients used to generate the fit. If the function cannot find a valid fit, `fitprops.coeffs` is an empty vector.
`fitprops.sumSqErr`	The sum squared error between the log of the fitted BER points and the log of the empirical BER points.
`fitprops.exitState`	The exit condition of `berfit`: `'The curve fit converged to a solution.'`, `'The maximum number of function evaluations was exceeded.'`, or `'No desirable fit was found'`.
`fitprops.funcCount`	The number of function evaluations used in minimizing the sum squared error function.
`fitprops.iterations`	The number of iterations taken in minimizing the sum squared error function. This is not necessarily equal to the number of function evaluations.

`berfit(...)` plots the empirical and fitted BER data.

`berfit(empEbNo,empber,fitEbNo,options,'all')` plots the empirical and fitted BER data from all the possible fits, listed in the "Algorithms" on page 2-80 below, that return a valid fit. To avoid overriding default options, use `options = []`.

Note A valid fit must be

- real-valued
- monotonically decreasing
- greater than or equal to 0 and less than or equal to 1

If a fit does not confirm to this criteria, it is rejected.

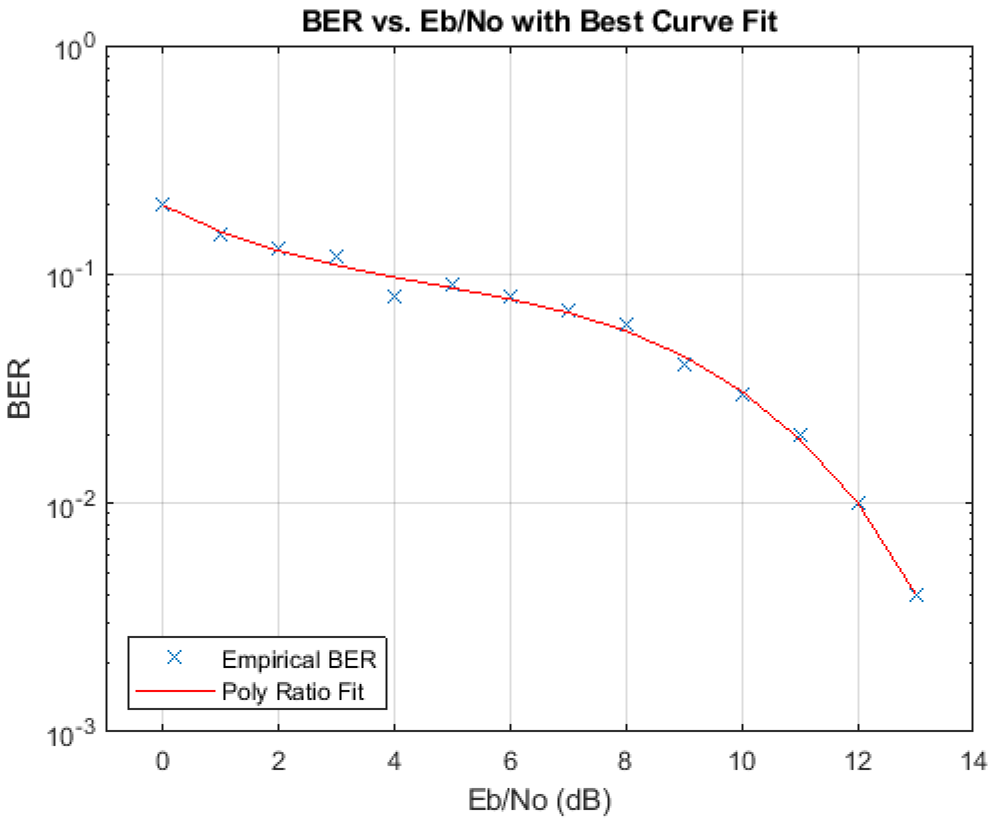
Examples

Bit Error Rate Curve Fitting

These examples illustrate the syntax of the `berfit` function, but they use hard-coded or theoretical BER data for simplicity. For an example that uses empirical BER data from a simulation, see “Curve Fitting An Error Rate Plot”.

Best fit for a sample set of data

```
EbN0 = 0:13;
berdata = [.2 .15 .13 .12 .08 .09 .08 .07 .06 .04 .03 .02 .01 .004];
berfit(EbN0,berdata);
```

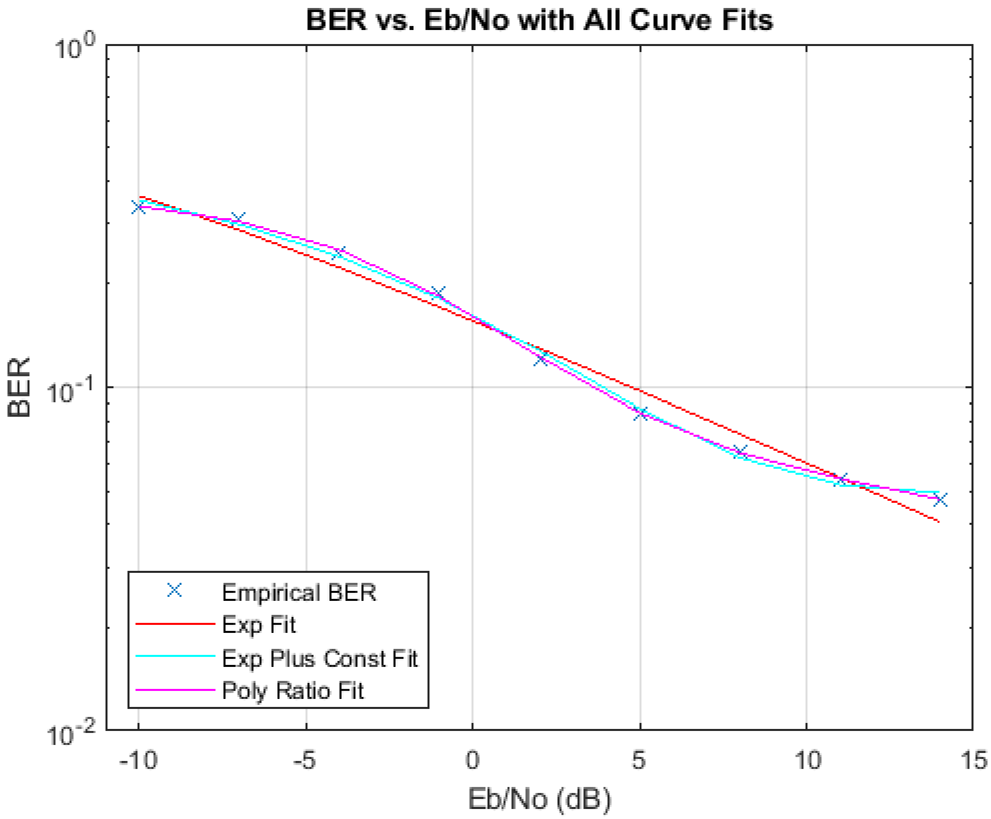


Plot the best fit. The curve connects the points created by evaluating the fit expression at the values in EbN0. To make the curve look smoother, use a syntax like `berfit(EbN0,berdata,[0:0.2:13])`. This alternative syntax uses more points when plotting the curve, but it does not change the fit expression.

Fit for a BER curve with an error floor

We generate the empirical BER array by simulating a channel with a null (ch = [0.5 0.47]) with BPSK modulation and linear MMSE equalizer at the receiver. We run the berfit with the 'all' option. The 'doubleExp+const' fit does not provide a valid fit, and the 'exp' fit type does not work well for this data. The 'exp+const' and 'polyRatio' fits closely match the simulated data.

```
EbN0 = -10:3:15;
empBER = [0.3361 0.3076 0.2470 0.1878 0.1212 0.0845 0.0650 0.0540 0.0474];
figure; berfit(EbN0, empBER, [], [], 'all');
```



Use of the options input structure as well as the fitprops output structure

The 'notify' value for the display level causes the function to produce output when one of the attempted fits does not converge. The exitState field of the output structure also indicates which fit converges and which fit does not.

```
M = 8; EbN0 = 3:10;
berdata = berfading(EbN0,'psk',M,2); % Compute theoretical BER.
noisydata = berdata.*[.93 .92 1 .59 .08 .15 .01 .01];
% Say when fit fails to converge.
options = optimset('display','notify');

disp('*** Trying exponential fit.') % Poor fit

*** Trying exponential fit.

[fitber1,fitprops1] = berfit(EbN0,noisydata,EbN0,...
   options,'exp')
```

```
Exiting: Maximum number of function evaluations has been exceeded
         - increase MaxFunEvals option.
         Current function value: 2.749919

fitber1 = 1×8

    0.1247    0.0727    0.0376    0.0168    0.0064    0.0020    0.0005    0.0001


fitprops1 = struct with fields:
       fitType: 'exp'
        coeffs: [4x1 double]
      sumSqErr: 2.7499
     exitState: 'The maximum number of function evaluations has been exceeded'
     funcCount: 10001
    iterations: 6193
```

```
disp('*** Trying polynomial ratio fit.') % Good fit
```

```
*** Trying polynomial ratio fit.
```

```
[fitber2,fitprops2] = berfit(EbN0,noisydata,EbN0,...
   options,'polyRatio')
```

```
fitber2 = 1×8

    0.1701    0.0874    0.0407    0.0169    0.0060    0.0016    0.0003    0.0001


fitprops2 = struct with fields:
       fitType: 'polyRatio'
        coeffs: [6x1 double]
      sumSqErr: 2.3880
     exitState: 'The curve fit converged to a solution'
     funcCount: 554
    iterations: 331
```

Algorithms

The `berfit` function fits the BER data using unconstrained nonlinear optimization via the `fminsearch` function. The closed-form functions that `berfit` considers are listed in the table below, where *x* is the E_b/N_0 in linear terms (*not* dB) and *f* is the estimated BER. These functions were empirically found to provide close fits in a wide variety of situations, including exponentially decaying BERs, linearly varying BERs, and BER curves with error rate floors.

Value of *fittype*	Functional Expression
'exp'	$f(x) = a_1 \exp\left[\frac{-(x - a_2)^{a_3}}{a_4}\right]$
'exp+const'	$f(x) = a_1 \exp\left[\frac{-(x - a_2)^{a_3}}{a_4}\right] + a_5$

Value of *fittype*	Functional Expression
'polyRatio'	$f(x) = \frac{a_1x^2 + a_2x + a_3}{x^3 + a_4x^2 + a_5x + a_6}$
'doubleExp+const'	$a_1\exp\left[\frac{-(x-a_2)^{a_3}}{a_4}\right] + a_5\exp\left[\frac{-(x-a_6)^{a_7}}{a_8}\right] + a_9$

The sum squared error function that `fminsearch` attempts to minimize is

$$F = \sum[\log(\text{empirical BER}) - \log(\text{fitted BER})]^2$$

where the fitted BER points are the values in `fitber` and the sum is over the E_b/N_0 points given in `empEbNo`. It is important to use the log of the BER values rather than the BER values themselves so that the high-BER regions do not dominate the objective function inappropriately.

References

For a general description of unconstrained nonlinear optimization, see the following work.

[1] Chapra, Steven C., and Raymond P. Canale, *Numerical Methods for Engineers*, Fourth Edition, New York, McGraw-Hill, 2002.

See Also

Introduced before R2006a

bersync

Bit error rate (BER) for imperfect synchronization

Syntax

```
ber = bersync(EbNo,timerr,'timing')
ber = bersync(EbNo,phaserr,'carrier')
```

Description

`ber = bersync(EbNo,timerr,'timing')` returns the BER of uncoded coherent binary phase shift keying (BPSK) modulation over an additive white Gaussian noise (AWGN) channel with imperfect timing. The normalized timing error is assumed to have a Gaussian distribution. `EbNo` is the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels. `timerr` is the standard deviation of the timing error, normalized to the symbol interval. `timerr` must be between 0 and 0.5.

`ber = bersync(EbNo,phaserr,'carrier')` returns the BER of uncoded BPSK modulation over an AWGN channel with a noisy phase reference. The phase error is assumed to have a Gaussian distribution. `phaserr` is the standard deviation of the error in the reference carrier phase, in radians.

Examples

Calculate Bit Error Rate (BER) for Imperfect Synchronization

The code below computes the BER of coherent BPSK modulation over an AWGN channel with imperfect timing. The example varies both `EbNo` and `timerr`. (When `timerr` assumes the final value of zero, the `bersync` command produces the same result as `berawgn(EbNo,'psk',2)`.)

```
EbNo = [4 8 12];
timerr = [0.2 0.07 0];
ber = zeros(length(timerr),length(EbNo));
for ii = 1:length(timerr)
    ber(ii,:) = bersync(EbNo,timerr(ii),'timerr');
end
```

Display result using scientific notation.

```
format short e; ber
```

```
ber = 3×3

   5.2073e-02   2.0536e-02   1.1160e-02
   1.8948e-02   7.9757e-04   4.9008e-06
   1.2501e-02   1.9091e-04   9.0060e-09
```

Switch back to default notation format.

```
format;
```

Limitations

The numerical accuracy of this function's output is limited by

- Approximations in the analysis leading to the closed-form expressions that the function uses
- Approximations related to the numerical implementation of the expressions

You can generally rely on the first couple of significant digits of the function's output.

Limitations Related to Extreme Values of Input Arguments

Inherent limitations in numerical precision force the function to assume perfect synchronization if the value of `timerr` or `phaserr` is very small. The table below indicates how the function behaves under these conditions.

Condition	Behavior of Function
`timerr < eps`	`bersync(EbNo,timerr,'timing')` defined as `berawgn(EbNo,'psk',2)`
`phaserr < eps`	`bersync(EbNo,phaserr,'carrier')` defined as `berawgn(EbNo,'psk',2)`

Algorithms

This function uses formulas from [3].

When the last input is `'timing'`, the function computes

$$\frac{1}{4\pi\sigma}\int_{-\infty}^{\infty}\exp(-\frac{\xi^2}{2\sigma^2})\int_{\sqrt{2R}(1-2|\xi|)}^{\infty}\exp(-\frac{x^2}{2})dxd\xi+\frac{1}{2\sqrt{2\pi}}\int_{\sqrt{2R}}^{\infty}\exp(-\frac{x^2}{2})dx$$

where σ is the `timerr` input and R is the value of `EbNo` converted from dB to a linear scale.

When the last input is `'carrier'`, the function computes

$$\frac{1}{\pi\sigma}\int_{0}^{\infty}\exp(-\frac{\phi^2}{2\sigma^2})\int_{\sqrt{2R}\cos\phi}^{\infty}\exp(-\frac{y^2}{2})dyd\phi$$

where σ is the `phaserr` input and R is the value of `EbNo` converted from dB to a linear scale.

Alternatives

As an alternative to the `bersync` function, invoke the BERTool GUI (`bertool`) and use the **Theoretical** tab.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

[2] Sklar, Bernard, *Digital Communications: Fundamentals and Applications*, Second Edition, Upper Saddle River, NJ, Prentice-Hall, 2001.

[3] Stiffler, J. J., *Theory of Synchronous Communications*, Englewood Cliffs, NJ, Prentice-Hall, 1971.

See Also

`berawgn` | `bercoding` | `berfading` | `bertool`

Topics

"Theoretical Results"

Introduced before R2006a

bertool

Open bit error rate analysis GUI (BERTool)

Syntax

```
bertool
```

Description

`bertool` launches the Bit Error Rate Analysis Tool (BERTool). The BERTool application enables you to analyze the bit error rate (BER) performance of communications systems. BERTool computes the BER as a function of signal-to-noise ratio. It analyzes performance either with Monte-Carlo simulations of MATLAB functions and Simulink models or with theoretical closed-form expressions for selected types of communication systems. See “BERTool” to learn more.

Introduced before R2006a

bi2de

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 decimal integer.

`d = bi2de(b,flg)` converts a binary row vector to a decimal integer, where `flg` determines the position of the most significant digit.

`d = bi2de(b,p)` converts a base-`p` row vector `b` to a decimal integer.

`d = bi2de(b,p,flg)` converts a base-`p` row vector to a decimal integer, where `flg` determines the position of the most significant digit.

Examples

Convert Binary Numbers to Decimals

Generate a matrix that contains binary representations of five random numbers between 0 and 15. Convert the binary numbers to decimal integers.

```
b = randi([0 1],5,4);
d = bi2de(b)
```

```
d = 5×1

     1
     5
    14
    11
    15
```

Convert Binary to Decimal

This example shows how to convert binary numbers to decimal integers. It highlights the difference between right- and left- most significant digit positioning.

```
b1 = [0 1 0 1 1];
b2 = [1 1 1 0];
```

Convert the two binary arrays to decimal by using the bi2de function. Assign the most significant digit is the leftmost element. The output of converting b1 corresponds to $0(2^4) + 1(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = 11$, and b2 corresponds to $1(2^3) + 1(2^2) + 1(2^1) + 0(2^0) = 14$.

```
d1 = bi2de(b1,'left-msb')
```

```
d1 = 11
```

```
d2 = bi2de(b2,'left-msb')
```

```
d2 = 14
```

Assign the most significant digit is the rightmost element. The output of converting b1 corresponds to $0(2^0) + 1(2^1) + 0(2^2) + 1(2^3) + 1(2^4) = 26$, and b2 corresponds to $1(2^0) + 1(2^1) + 1(2^2) + 0(2^3) = 7$.

```
d1 = bi2de(b1,'right-msb')
```

```
d1 = 26
```

```
d2 = bi2de(b2,'right-msb')
```

```
d2 = 7
```

Convert Octal Number to Decimal

Convert an octal (base-8) number to its decimal equivalent.

Assign the most significant digit to the leftmost position. The output corresponds to $4(8^3) + 2(8^2) + 7(8^1) + 1(8^0) = 2233$.

```
d = bi2de([4 2 7 1],8,'left-msb')
```

```
d = 2233
```

Assign the most significant digit to the rightmost position. The output corresponds to $4(8^0) + 2(8^1) + 7(8^2) + 1(8^3) = 980$.

```
d = bi2de([4 2 7 1],8,'right-msb')
```

```
d = 980
```

Input Arguments

b — Binary input

row vector | matrix

Binary input, specified as a row vector or matrix of positive integer or logical values.

Note b must represent an integer less than or equal to 2^{52}.

Data Types: double | single | logical | integer | fi

`flg` — MSB flag

`'right-msb'` (default) | `'left-msb'`

MSB flag, specified as `'right-msb'` or `'left-msb'`.

- `'right-msb'` –– Indicates the right (or last) column of the binary input, b, as the most significant bit (or highest-order digit).
- `'left-msb'` –– Indicates the left (or first) column of the binary input, b, as the most significant bit (or highest-order digit).

Data Types: `char` | `string`

p — Base

2 (default) | positive integer scalar

Base of the input `b`, specified as an integer greater than or equal to `2`.

Data Types: `double` | `single`

Output Arguments

d — Decimal output

nonnegative integer | vector

Decimal output, returned as an nonnegative integer or row vector. If `b` is a matrix, each row represents a base-`p` number. In this case, the output `d` is a column vector in which each element is the decimal representation of the corresponding row of `b`.

If the input data type is

- An integer data type and the value of `d` can be contained in the same integer data type as the input, the output data type uses the same data type as the input. Otherwise, the output data type is chosen to be big enough to contain the decimal output.
- double or logical data type, the output data type is double.
- single data type, the output data type is single.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`de2bi`

Introduced before R2006a

bin2gray

(To be removed) Convert positive integers into corresponding Gray-encoded integers

Note will be removed in a future release. Use the appropriate modulation object or function to remap constellation points instead. For more information, see “Compatibility Considerations”.

Syntax

```
y = bin2gray(x,modulation,M)
[y,map] = bin2gray(x,modulation,M)
```

Description

`y = bin2gray(x,modulation,M)` generates a Gray-encoded vector or matrix output `y` with the same dimensions as its input parameter `x`. `x` can be a scalar, vector, matrix, or 3-D array. `modulation` is the modulation type and must be `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`. `M` is the modulation order and must be an integer power of 2.

Note If you are converting binary-coded data to Gray-coded data and modulating the result immediately afterwards, you should use the appropriate modulation object or function with the `'Gray'` option, instead of `bin2gray`.

`[y,map] = bin2gray(x,modulation,M)` generates a Gray-encoded output `y` with its respective Gray-encoded constellation map, `map`.

You can use map output to label a Gray-encoded constellation. The map output gives the Gray-encoded labels for the corresponding modulation.

Examples

Binary to Gray Symbol Mapping

This example shows how to use the `bin2gray` and `gray2bin` functions to map integer inputs from a natural binary order symbol mapping to a Gray-coded signal constellation and vice versa, assuming 16-QAM modulation. In addition, a visual representation of the difference between Gray-coded and binary-coded symbol mappings is shown.

Convert Binary to Gray

Create a complete vector of 16-QAM integers.

```
M = 16;
x = (0:M-1);
```

Convert the input vector from a natural binary order to a Gray-encoded vector using `bin2gray`.

```
[y,mapy] = bin2gray(x,'qam',M);
```

Convert Gray to Binary

Convert the Gray-encoded symbols, y, back to a binary ordering using gray2bin.

```
z = gray2bin(y,'qam',M);
```

Verify that the original data, x, and the final output vector, z, are identical.

```
isequal(x,z)
```

```
ans = logical
   1
```

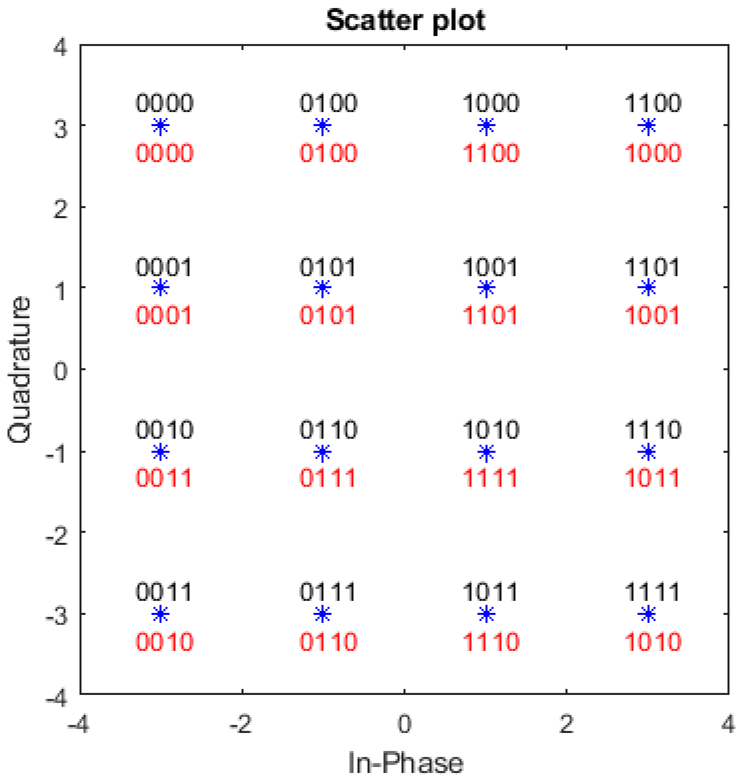
Show Symbol Mappings

To create a constellation plot showing the different symbol mappings, use the qammod function to find the complex symbol values.

```
sym = qammod(x,M);
```

Plot the constellation symbols and label them using the Gray (y) and binary (z) output vectors. The binary representation of the Gray-coded symbols is shown in black while the binary representation of the naturally ordered symbols is shown in red. Set the axes scaling so that all points are displayed.

```
scatterplot(sym,1,0,'b*');
for k = 1:16
    text(real(sym(k))-0.3,imag(sym(k))+0.3,...
        dec2base(mapy(k),2,4));

    text(real(sym(k))-0.3,imag(sym(k))-0.3,...
        dec2base(z(k),2,4),'Color',[1 0 0]);
end
axis([-4 4 -4 4])
```



Input Arguments

x — Binary-encoded data

vector | matrix

Input binary-encoded data, specified as a vector or matrix.

Data Types: `double`

`modulation` — Modulation type

`'qam'` | `'pam'` | `'fsk'` | `'dpsk'` | `'psk'`

Modulation type, specified as, `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`

`M` — Modulation order

scalar

Modulation order, specified as an integer power of 2.

Data Types: `double`

Output Arguments

y — Gray-encoded data

vector | matrix

Gray-encoded data with the same size and dimensions input x.

map — Map of labels
vector

Map output to label a Gray-encoded constellation, specified as a vector with a length the size of the modulation order, M. The map gives the Gray-encoded labels for the corresponding modulation.

Compatibility Considerations

bin2gray will be removed
Not recommended starting in R2020a

bin2gray will be removed in a future release. Use the appropriate modulation object or function instead. If your workflow uses bin2gray or gray2bin with any of the modulations schemes in this table, follow the appropriate example.

Modulation	Old Functionality	Use This Instead

QAM (qammod and qamdemod)	x = randi([0 63],1,100);
	y = bin2gray(x,'qam',64);
	z = qammod(y,64,'bin');
	
	x = 2*(randn(100,1)+1j*randn(100,1));
	y = qamdemod(x,64,'bin');
	z = gray2bin(y,'qam',64);
		x = randi([0 63],1,100);
		z = qammod(x,64,'gray');
		
		x = 2*(randn(100,1)+1j*randn(100,1));
		z = qamdemod(x,64,'gray')

PAM (pammod and pamdemod)	x = randi([0 63],1,100);
	y = gray2bin(x,'pam',64);
	z = pammod(y,64,pi/4,'bin');
	
	x = 2*(randn(100,1)+1j*randn(100,1));
	y = pamdemod(x,64,pi/4,'bin');
	z = bin2gray(y,'pam',64);
		x = randi([0 63],1,100);
		z = pammod(x,64,pi/4,'gray');
		
		x = 2*(randn(100,1)+1j*randn(100,1));
		z = pamdemod(x,64,pi/4,'gray')

FSK (fskmod and fskdemod)	x = randi([0 63],1,100);
	y = gray2bin(x,'fsk',64);
	z = fskmod(y,64,1,256,256,'cont','bin')
	
	x = 2*(randn(512,1)+1j*randn(512,1));
	y = fskdemod(x,64,1,256,256,'bin');
	z = bin2gray(y,'fsk',64)
		x = randi([0 63],1,100);
		z = fskmod(x,64,1,256,256,'cont','gray')
		x = 2*(randn(512,1)+1j*randn(512,1));
		z = fskdemod(x,64,1,256,256,'gray');

DPSK (dpskmod and dpskdemod)	x = randi([0 63],1,100);
	y = gray2bin(x,'dpsk',64);
	z = dpskmod(y,64,pi/4,'bin');
	
	x = 2*(randn(100,1)+1j*randn(100,1));
	y = dpskdemod(x,64,pi/4,'bin');
	z = bin2gray(y,'dpsk',64);
		x = randi([0 63],1,100);
		z = dpskmod(x,64,pi/4,'gray');
		
		x=2*(randn(100,1)+1j*randn(100,1));
		z = dpskdemod(x,64,pi/4,'gray');

PSK (pskmod and pskdemod)	x=randi([0 63],1,100);
	y=gray2bin(x,'psk',64);
	z=pskmod(y,64,0,'bin');
	
	x = 2*(randn(100,1)+1j*randn(100,1));
	y = pskdemod(x,64,0,'bin');
	z = bin2gray(y,'psk',64);
		x=randi([0 63],1,100);
		z=pskmod(x,64,0,'gray');
		
		x = 2*(randn(100,1)+1j*randn(100,1));
		z = pskdemod(x,64,0,'gray');

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dpskmod` | `fskmod` | `pammod` | `pskmod` | `qammod`

Topics

Gray Encoding a Modulated Signal

Introduced before R2006a

biterr

Number of bit errors and bit error rate (BER)

Syntax

```
[number,ratio] = biterr(x,y)
[number,ratio] = biterr(x,y,k)
[number,ratio] = biterr(x,y,k,flag)
[number,ratio,individual] = biterr( ___ )
```

Description

`[number,ratio] = biterr(x,y)` compares the unsigned binary representation of elements in `x` to those in `y`. The function returns `number`, the number of bits that differ in the comparison, and `ratio`, the ratio of `number` to the total number of bits. The function determines the order in which it compares `x` and `y` based on their sizes. For more details, see Algorithms on page 2-98 section.

`[number,ratio] = biterr(x,y,k)` also specifies `k`, the maximum number of bits for each element in `x` and `y`. If the unsigned binary representation of any element in `x` or `y` is more than `k` digits, the function errors.

`[number,ratio] = biterr(x,y,k,flag)` specifies a `flag` to override default settings for how the function compares the elements and computes the outputs. For more information, see Algorithms on page 2-99 section.

`[number,ratio,individual] = biterr( ___ )` returns the binary comparison result of `x` and `y` as matrix `individual`. You can specify any of the input argument combination from the previous syntaxes.

Examples

Bit Error Rate Computation

Create two binary matrices.

```
x = [0 0; 0 0; 0 0; 0 0]
```

```
x = 4×2

     0     0
     0     0
     0     0
     0     0
```

```
y = [0 0; 0 0; 0 0; 1 1]
```

```
y = 4×2

     0     0
     0     0
     0     0
     1     1
```

Determine the number of bit errors.

```
numerrs = biterr(x,y)
```

```
numerrs = 2
```

Compute the number of column-wise errors .

```
numerrs = biterr(x,y,[],'column-wise')
```

```
numerrs = 1×2

     1     1
```

Compute the number of row-wise errors.

```
numerrs = biterr(x,y,[],'row-wise')
```

```
numerrs = 4×1

     0
     0
     0
     2
```

Compute the number of overall errors. Behavior is the same as the default behaviour.

```
numerrs = biterr(x,y,[],'overall')
```

```
numerrs = 2
```

Estimate Bit Error Rate for 64-QAM in AWGN

Demodulate a noisy 64-QAM signal and estimate the bit error rate (BER) for a range of Eb/No values. Compare the BER estimate to theoretical values.

Set the simulation parameters.

```
M = 64;                 % Modulation order
k = log2(M);            % Bits per symbol
EbNoVec = (5:15)';      % Eb/No values (dB)
numSymPerFrame = 100;   % Number of QAM symbols per frame
```

Initialize the results vector.

```
berEst = zeros(size(EbNoVec));
```

The main processing loop executes these steps.

- Generate binary data and convert to 64-ary symbols.
- QAM-modulate the data symbols.
- Pass the modulated signal through an AWGN channel.
- Demodulate the received signal.
- Convert the demodulated symbols into binary data.
- Calculate the number of bit errors.

The `while` loop continues to process data until either 200 errors are encountered or 1e7 bits are transmitted.

```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k);
    % Reset the error and bit counters
    numErrs = 0;
    numBits = 0;

    while numErrs < 200 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame,k);
        dataSym = bi2de(dataIn);

        % QAM modulate using 'Gray' symbol mapping
        txSig = qammod(dataSym,M);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal
        rxSym = qamdemod(rxSig,M);
        % Convert received symbols to bits
        dataOut = de2bi(rxSym,k);

        % Calculate the number of bit errors
        nErrors = biterr(dataIn,dataOut);

        % Increment the error and bit counters
        numErrs = numErrs + nErrors;
        numBits = numBits + numSymPerFrame*k;
    end

    % Estimate the BER
    berEst(n) = numErrs/numBits;
end
```

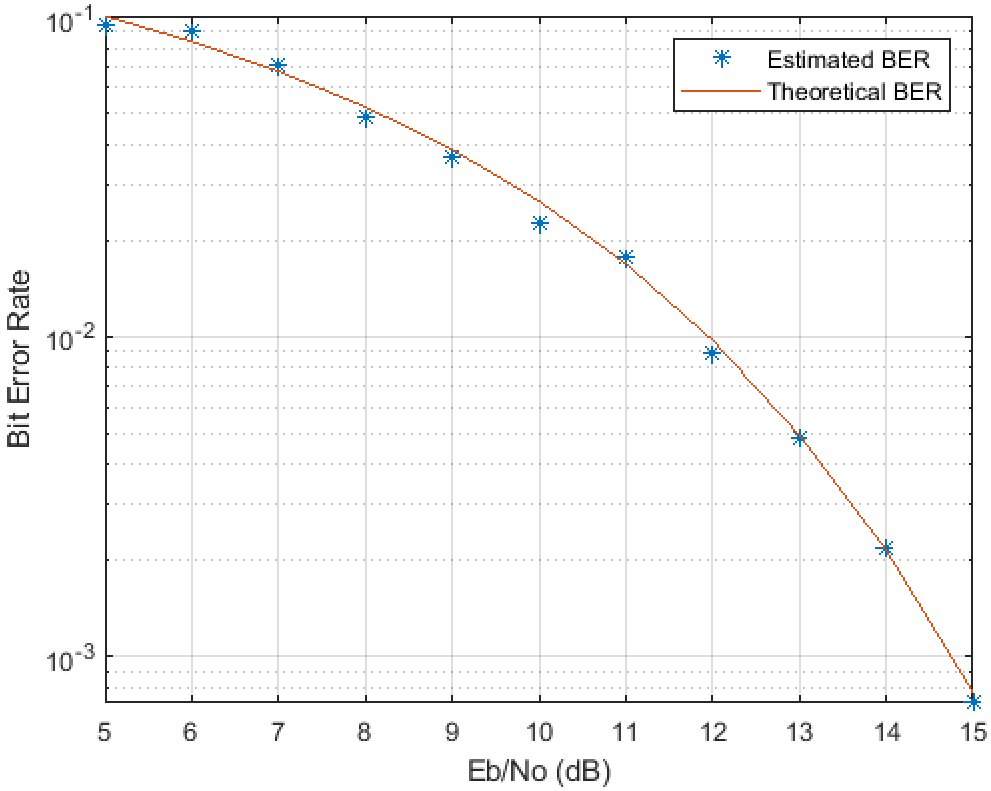
Determine the theoretical BER curve by using the `berawgn` function.

```
berTheory = berawgn(EbNoVec,'qam',M);
```

Plot the estimated and theoretical BER data. The estimated BER data points are well aligned with the theoretical curve.

```
semilogy(EbNoVec,berEst,'*')
hold on
semilogy(EbNoVec,berTheory)
grid
```

```
legend('Estimated BER','Theoretical BER')
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



Input Arguments

x,y — Inputs to be compared (as separate arguments)

vector | matrix

Inputs to be compared, specified as separate arguments, as a vector or matrix of nonnegative integer elements. The function converts each element of x and y to its unsigned binary representation for comparison.

Data Types: single | double | int8 | int16 | int32 | int64 | uint8 | uint16 | uint32 | uint64 | logical

k — Maximum number of bits for input elements

positive integer

Maximum number of bits for input elements of x and y, specified as a positive integer. If the number of bits required for binary representation of any element in x or y is greater than k, the function errors.

If you do not set k, the function sets it as the number of bits in the binary representation of the largest element in x and y.

Data Types: `single` | `double`

`flag` — Flag to override default settings
`'overall'` | `'row-wise'` | `'column-wise'`

Flag to override default settings of the function, specified as `'overall'`, `'row-wise'`, or `'column-wise'`. Flag specifies how the function compares elements in inputs `x,y` and computes the output. For more details, see the Algorithms on page 2-99 section.

Data Types: `string` | `char`

Output Arguments

`number` — Number of bit errors
nonnegative integer | integer vector

Number of bit errors, returned as a nonnegative integer or integer vector.

Data Types: `single` | `double`

`ratio` — Bit error rate
scalar

Bit error rate, returned as a scalar. `ratio` is the number of bit errors, `number`, to the total number of bits used in the binary representation. The total number of bits is `k` times the number of entries in the smaller of the inputs `x,y`.

`individual` — Binary comparison result of each input element
matrix

Binary comparison result of each input element in `x` and `y`, returned as a matrix whose dimensions are those of the larger of `x` and `y`. Each element specifies the number of bits by which the elements in the pair differ.

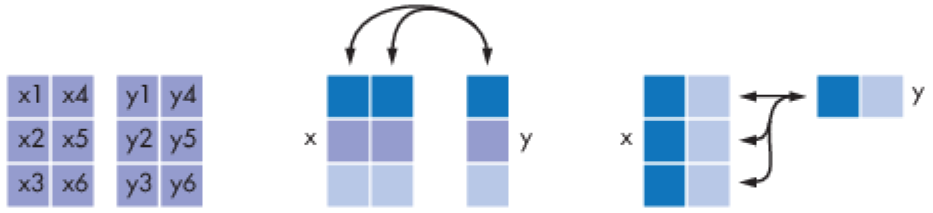
Data Types: `single` | `double`

Algorithms

Comparing Inputs Based on Sizes

The function uses the sizes of `x` and `y` to determine the order in which it compares their elements.

- If inputs are matrices of the same dimensions, then the function compares the inputs element by element. `number` is a nonnegative integer in this case. For example, see case (a) in the figure.
- If one input is a matrix and the other input is a column vector, then the function compares each column of the matrix element by element with the column vector. The number of rows in the matrix must be equal to the length of the column vector. In other words, if the matrix has dimensions *m*-by-*n*, then the column vector must have dimensions *m*-by-1. For example, see case (b) in the figure.
- If one input is a matrix and the other input is a row vector, then the function compares each row of the matrix element by element with the row vector. The number of columns in the matrix must be equal to the length of the row vector. In other words, if the matrix has dimensions *m*-by-*n*, then the row vector must have dimensions 1-by-*n*. For example, see case (c) in the figure.



(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

Comparing Inputs Based on Flag

This table describes how the output is computed based on the different values of `flag`. `x` is considered as a matrix in this table and the size of `y` is varied.

Size of `y`	`flag` Value	Type of Comparison	`number` Value	Total Number of Bits
Matrix	`'overall'` (default)	Element by element	Total number of bit errors	`k` times the number of elements in `y`
	`'row-wise'`	*m*th row of `x` to *m*th row of `y`	Column vector whose elements represent the bit errors in each row	`k` times the number of elements in `y`
	`'column-wise'`	*m*th column of `x` to *m*th column of `y`	Row vector whose elements represent the bit errors in each column	`k` times the number of elements in `y`
Row vector	`'overall'`	`y` to each row of `x`	Total number of bit errors	`k` times the number of elements of `x`
	`'row-wise'`(default)	`y` to each row of `x`	Column vector whose elements represent the bit errors in each row of `x`	`k` times the size of `y`
Column vector	`'overall'`	`y` to each column of `x`	Total number of bit errors	`k` times the number of elements of `x`
	`'column-wise'` (default)	`y` to each column of `x`	Row vector whose elements represent bit errors in each column of `x`	`k` times the size of `y`

See Also

`alignsignals` | `finddelay` | `symerr`

Introduced before R2006a

bsc

Binary symmetric channel

Syntax

```
ndata = bsc(data,probability)
ndata = bsc(data,probability,streamhandle)
ndata = bsc(data,probability,seed)
[ndata,err] = bsc( ___ )
```

Description

`ndata = bsc(data,probability)` passes the binary input signal `data` through a binary symmetric channel having the specified error probability. The channel introduces a bit error and processes each element of the input `data` independently. `data` must be an array of binary numbers or a Galois array in GF(2). `probability` must be a scalar from 0 to 1.

`ndata = bsc(data,probability,streamhandle)` accepts a random stream handle to generate uniform noise samples by using `rand`. Providing a random stream handle or using the `reset (RandStream)` function on the default random stream object enables you to generate repeatable noise samples. For more information, see `RandStream`.

`ndata = bsc(data,probability,seed)` accepts a seed value, for initializing the uniform random number generator, `rand`. If you want to generate repeatable noise samples, then either reset the random stream input before calling `bsc` or use the same seed input.

`[ndata,err] = bsc( ___ )` returns an array containing the channel errors, using any of the preceding syntaxes.

Examples

Add Bit Errors to Bit Stream

Using the `bsc` function, introduce bit errors in the bits in a random matrix with probability 0.15.

```
z = randi([0 1],100,100); % Random matrix
nz = bsc(z,.15); % Binary symmetric channel
[numerrs, pcterrs] = biterr(z,nz) % Number and percentage of errors
```

```
numerrs = 1509

pcterrs = 0.1509
```

The output below is typical. For relatively small sets of data, the percentage of bit errors is not exactly 15% in most trials. If the size of the matrix `z` is large, the bit error percentage will be closer to the exact probability you specify.

Check for Errors After Decoding

Using the bsc function, introduce bit errors in the bits in a random matrix with probability 0.01. Use Viterbi decoder to decode message data.

Define trellis for Viterbi decoder. Generate and encode message data.

```
trel = poly2trellis([4 3],[4 5 17;7 4 2]);
msg = ones(10000,1);
```

Create objects for convolutional encoder, Viterbi decoder, and error rate calculator.

```
hEnc = comm.ConvolutionalEncoder(trel);
hVitDec = comm.ViterbiDecoder(trel, 'InputFormat','hard', 'TracebackDepth',...
    2, 'TerminationMethod', 'Truncated');
hErrorCalc = comm.ErrorRate;
```

Encode the message data. Introduce bit errors. Display the total number of errors.

```
code = hEnc(msg);
[ncode,err] = bsc(code,.01);
numchanerrs = sum(sum(err))
```

```
numchanerrs = 158
```

Decode the data and check the number of errors after decoding.

```
dcode = hVitDec(ncode);
berVec = hErrorCalc(msg, dcode);
ber = berVec(1)
```

```
ber = 0.0049
```

```
numsyserrs = berVec(2)
```

```
numsyserrs = 49
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation supported, except for syntaxes that include a RandStream object.

See Also

Functions

RandStream | awgn | gf | rand

Topics

“Design a Rate 2/3 Feedforward Encoder Using Simulink”

Introduced before R2006a

cdma2000ForwardReferenceChannels

Define cdma2000 forward reference channel

Syntax

```
cfg = cdma2000ForwardReferenceChannels(wv)
cfg = cdma2000ForwardReferenceChannels(wv,numchips)
cfg = cdma2000ForwardReferenceChannels(BSTM-RC,numchips,P,M)
cfg = cdma2000ForwardReferenceChannels(traffic,numchips,F-SCH-SPEC)
```

Description

`cfg = cdma2000ForwardReferenceChannels(wv)` returns a structure, `cfg`, that defines the cdma2000® forward link parameters given the input waveform identifier, `wv`. To generate a forward link reference channel waveform, pass this structure to the `cdma2000ForwardWaveformGenerator` function.

For all syntaxes, `cdma2000ForwardReferenceChannels` creates a configuration structure that is compliant with the cdma2000 physical layer specification [1].

`cfg = cdma2000ForwardReferenceChannels(wv,numchips)` specifies the number of chips to generate.

`cfg = cdma2000ForwardReferenceChannels(BSTM-RC,numchips,P,M)` returns the data structure for the `BSTM-RC` waveform identifiers, given the total traffic channel power, `P`, and the number of traffic channels, `M`. For more information on base station testing, see Table 6.5.2-1 of [2].

`cfg = cdma2000ForwardReferenceChannels(traffic,numchips,F-SCH-SPEC)` returns the data structure for the specified traffic channel, `traffic`, and the forward supplemental channel (F-SCH) and frame length combination, `F-SCH-SPEC`. If omitted, `F-SCH-SPEC` has a default value of the lowest F-SCH data rate allowable for a 20 ms frame length, given the radio configuration specified by `traffic`.

Examples

Generate Waveform for RC2 Forward Traffic Channels

Create a parameter structure, `config`, for all forward traffic channels (F-FCH and F-SCCH) that are supported by radio configuration 2.

```
config = cdma2000ForwardReferenceChannels('ALL-RC2')
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
```

```
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 1000
                  FPICH: [1x1 struct]
                 FAPICH: [1x1 struct]
                FTDPICH: [1x1 struct]
               FATDPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FSYNC: [1x1 struct]
                  FBCCH: [1x1 struct]
                  FCACH: [1x1 struct]
                  FCCCH: [1x1 struct]
                 FCPCCH: [1x1 struct]
                  FQPCH: [1x1 struct]
                   FFCH: [1x1 struct]
                  FOCNS: [1x1 struct]
                  FSCCH: [1x1 struct]
```

Examine the fields for the Forward Fundamental Channel (F-FCH). The data rate is 14,400 bps and the frame length is 20 ms.

```
config.FFCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC2'
              DataRate: 14400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 7
             EnableQOF: 'Off'
    PowerControlEnable: 'Off'
```

Generate the complex waveform using the corresponding waveform generator function.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

A waveform composed of the channels specified by each substructure of `config` is generated by `cdma2000ForwardWaveformGenerator`.

Generate CDMA200 Waveform Containing Sync Channel Message

Create a reference channel, specify the sync channel message as the data source, add the `SyncMessage` structure to the `FSYNC` substructure. Generate the waveform using this reference channel configuration.

Create a reference channel for testing a base station using radio configuration 3.

```
config = cdma2000ForwardReferenceChannels('BSTM-RC3');
```

Adjust the Forward Sync Channel (F-SYNC) settings. Set a relative channel power of 0.0 dB and specify the sync channel message as the data source.

```
config.FSYNC.Power = 0.0;
config.FSYNC.DataSource = 'SyncMessage';
```

Define the sync channel message structure (for P_REV = 6 (IS-2000-0)) and add it to the `config.FSYNC` substructure. Display the `FSYNC` structure.

```
sm = struct();
sm.P_REV = 6;                          % Protocol revision field
sm.MIN_P_REV = 6;                      % Minimum protocol revision field
sm.SID = hex2dec('14B');               % System identifier field
sm.NID = 1;                            % Network identification field
sm.PILOT_PN = 0;                       % Pilot PN offset field
sm.LC_STATE = hex2dec('20000000000');  % Long code state field
sm.SYS_TIME = hex2dec('36AE0924C');    % System time field
sm.LP_SEC = 0;                         % Leap second field
sm.LTM_OFF = 0;                        % Local time offset field
sm.DAYLT = 0;                          % Daylight saving time indicator field
sm.PRAT = 0;                           % Paging channel data rate field
sm.CDMA_FREQ = hex2dec('2F6');         % CDMA frequency field
sm.EXT_CDMA_FREQ = hex2dec('2F6');     % Extended CDMA frequency field

config.FSYNC.SyncMessage = sm;

config.FSYNC
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    EnableCoding: 'On'
      DataSource: 'SyncMessage'
     SyncMessage: [1x1 struct]
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate F-CCCH Waveform

Create a structure for a 2000-chip forward common control channel (F-CCCH). Specify a 38,400 bps data rate, a 5 ms frame length, and an accompanying broadcast control channel (F-BCCH) with a 9600 bps data rate.

```
config   = cdma2000ForwardReferenceChannels('CONTROL-38400-5-9600',2000)
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
```

```
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2000
                  FPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FCCCH: [1x1 struct]
                  FBCCH: [1x1 struct]
```

Verify that the F-CCCH and F-BCCH data rates are 38,400 bps and 9600 bps, respectively.

```
config.FCCCH.DataRate
```

```
ans = 38400
```

```
config.FBCCH.DataRate
```

```
ans = 9600
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate Waveform for Base Station Testing

Create a reference channel for testing a base station using radio configuration 3. Specify the number of chips, the total power allocated to the individual channels, and the number of traffic channels. The FFCH substructure is a structure array whose dimensions are set by the number of traffic channels.

```
config = cdma2000ForwardReferenceChannels('BSTM-RC3',1000,-3,4)
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 1000
                  FPICH: [1x1 struct]
                  FSYNC: [1x1 struct]
                   FPCH: [1x1 struct]
                   FFCH: [1x4 struct]
```

Verify that the length of the FFCH substructure corresponds to the number of specified traffic channels, 4.

```
length(config.FFCH)
```

```
ans = 4
```

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Generate F-SCH Waveform

Create a traffic channel using radio configuration 7 composed of a 614,400 bps forward supplemental channel (F-SCH) having a 20 ms frame length. Set the number of chips to 5000.

```
config = cdma2000ForwardReferenceChannels('TRAFFIC-RC7-4800', ...
    5000,'F-SCH-614400-20')
```

```
config = struct with fields:
          SpreadingRate: 'SR3'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
           LongCodeState: 0
      PowerNormalization: 'Off'
       OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
        EnableModulation: 'Off'
     ModulationFrequency: 0
               NumChips: 5000
                  FPICH: [1x1 struct]
                   FFCH: [1x1 struct]
                   FSCH: [1x1 struct]
```

This channel uses spreading rate 3, `'SR3'`, which has a 3.75 MHz bandwidth.

Generate the forward link waveform.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector. The input typically identifies the channel type, radio configuration, data rate, and frame length. To specify `wv`, connect the substrings with hyphens, for example, `'CONTROL-19200-10-4800'`.

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
wv	'FPICH-ONLY'				Generates a waveform containing a pilot channel only.
	'CONTROL'	9600	20	4800 | 9600 | 19200	Character vector representing the forward common control channel (F-CCCH) data rate in bps, the frame length in ms, and the forward broadcast control channel (F-BCCH) data rate in bps. Specify `'CONTROL-9600-20-9600'` to create a structure variable, `wv`, with a 9600 bps F-CCCH data rate, a 20 ms frame length, and a 9600 bps F-BCCH data rate.
		19200	10 | 20		
		38400	5 | 10 | 20		
	'TRAFFIC'	RC1	1200 | 2400 | 4800 | 9600	N/A	Character vector representing the radio configuration and the forward fundamental channel (F-FCH) data rate in bps. Specify `'TRAFFIC-RC9-14400'` to create a channel with radio
		RC2 | RC5 | RC8 | RC9	1800 | 3600 | 7200 | 14400		

Parameter Field	Values				Description
	Substring 1	Substring 2	Substring 3	Substring 4	
		RC3 | RC4 | RC6 | RC7	1500 | 2700 | 4800 | 9600		configuration 9 having a 14400 bps F-FCH data rate.
	'BSTM'	RC1 | RC2 | RC3 | RC4 | RC5 | RC6 | RC7 | RC8 | RC9	N/A	N/A	Models for testing the base station transmitter. Specify 'BSTM-RC1' to create a structure for base station testing with radio configuration 1.
	'ALL'	RC1 | RC2 | RC3 | RC4 | RC5 | RC6 | RC7 | RC8 | RC9	N/A	N/A	Returns all channels that are supported for the specified radio configuration. Specify 'ALL-RC4' to create a structure containing all traffic channels for radio configuration 4.

Example: 'CONTROL-9600-20-9600'

Example: 'TRAFFIC-RC9-7200'

Example: 'ALL-RC5'

Data Types: char

numchips — Number of chips
1000 (default) | positive integer scalar

Number of chips, specified as a positive integer.

Example: 1024

Data Types: double

BSTM-RC — BSTM reference channel type
'BSTM-RC1' | 'BSTM-RC2' | 'BSTM-RC3' | 'BSTM-RC4' | 'BSTM-RC5' | 'BSTM-RC6' | 'BSTM-RC7' | 'BSTM-RC8' | 'BSTM-RC9'

BSTM reference channel type, specified as a character vector. For more information, see Table 6.5.2-1 of [2].

Example: 'BSTM-RC8'

Data Types: char

P — Power budget allocated to traffic channels

0 (default) | real scalar

Power budget allocated to all traffic channels, specified in decibels as a real scalar.

Example: 5

Data Types: double

M — Number of traffic channels

6 (default) | positive integer scalar

Number of traffic channels, specified as a positive integer.

Example: 8

Data Types: double

traffic — Traffic configuration

character vector

Traffic channel configuration, specified as a character vector. The table shows the supported traffic channel configurations.

Radio Configuration	Traffic Channel Configuration			
1	'TRAFFIC-RC1-1200'	'TRAFFIC-RC1-2400'	'TRAFFIC-RC1-4800'	'TRAFFIC-RC1-9600'
2	'TRAFFIC-RC2-1800'	'TRAFFIC-RC2-3600'	'TRAFFIC-RC2-7200'	'TRAFFIC-RC2-14400'
3	'TRAFFIC-RC3-1500'	'TRAFFIC-RC3-2700'	'TRAFFIC-RC3-4800'	'TRAFFIC-RC3-9600'
4	'TRAFFIC-RC4-1500'	'TRAFFIC-RC4-2700'	'TRAFFIC-RC4-4800'	'TRAFFIC-RC4-9600'
5	'TRAFFIC-RC5-1800'	'TRAFFIC-RC5-3600'	'TRAFFIC-RC5-7200'	'TRAFFIC-RC5-14400'
6	'TRAFFIC-RC6-1500'	'TRAFFIC-RC6-2700'	'TRAFFIC-RC6-4800'	'TRAFFIC-RC6-9600'
7	'TRAFFIC-RC7-1500'	'TRAFFIC-RC7-2700'	'TRAFFIC-RC7-4800'	'TRAFFIC-RC7-9600'
8	'TRAFFIC-RC8-1800'	'TRAFFIC-RC8-3600'	'TRAFFIC-RC8-7200'	'TRAFFIC-RC8-14400'
9	'TRAFFIC-RC9-1800'	'TRAFFIC-RC9-3600'	'TRAFFIC-RC9-7200'	'TRAFFIC-RC9-14400'

Example: 'TRAFFIC-RC6-4800' is a traffic channel that uses radio configuration 6 with a 4800 bps data rate.

Data Types: char

`F-SCH-SPEC` — Forward supplemental channel data rate and frame length

character vector

Forward supplemental channel data rate and frame length, specified as a character vector. The supported data rate and frame length combinations are summarized in the table.

Radio Configuration	Frame Length		
	20 ms	40 ms	80 ms
3 | **4** | **6** | **7**	'F-SCH-1500-20' | 'F-SCH-2700-20' | 'F-SCH-4800-20' | 'F-SCH-9600-20' | 'F-SCH-19200-20' | 'F-SCH-38400-20' | 'F-SCH-76800-20' | 'F-SCH-153600-20'	'F-SCH-1350-40' | 'F-SCH-2400-40' | 'F-SCH-4800-40' | 'F-SCH-9600-40' | 'F-SCH-19200-40' | 'F-SCH-38400-40' | 'F-SCH-76800-40'	'F-SCH-1200-80' | 'F-SCH-2400-80' | 'F-SCH-4800-80' | 'F-SCH-9600-80' | 'F-SCH-19200-80' | 'F-SCH-38400-80'
4 | **6** | **7**	'F-SCH-307200-20'	'F-SCH-153600-40'	'F-SCH-76800-80'
7	'F-SCH-614400-20'	'F-SCH-307200-40'	'F-SCH-153600-80'
5 | **8** | **9**	'F-SCH-1800-20' | 'F-SCH-3600-20' | 'F-SCH-7200-20' | 'F-SCH-14400-20' | 'F-SCH-28800-20' | 'F-SCH-57600-20' | 'F-SCH-115200-20' | 'F-SCH-230400-20'	'F-SCH-1800-40' | 'F-SCH-3600-40' | 'F-SCH-7200-40' | 'F-SCH-14400-40' | 'F-SCH-28800-40' | 'F-SCH-57600-40' | 'F-SCH-115200-40'	'F-SCH-1800-80' | 'F-SCH-3600-80' | 'F-SCH-7200-80' | 'F-SCH-14400-80' | 'F-SCH-28800-80' | 'F-SCH-57600-80'
8 | **9**	'F-SCH-460800-20'	'F-SCH-230400-40'	'F-SCH-115200-80'
9	'F-SCH-1036800-20'	'F-SCH-518400-40'	'F-SCH-259200-80'

For more data rate information for the cdma2000 forward links, see tables 3.1.3.1.3-2 and 3.1.3.1.3-4 of [1].

Example: `'F-SCH-460800-20'` is a supplemental channel with a 460,800 bps data rate and a 20 ms frame length.

Data Types: `char`

Output Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
SpreadingRate	'SR1' | 'SR3'	Spreading rate of the waveform. SR1 corresponds to a 1.2288 Mcps carrier. SR3 corresponds to a 3.6864 Mcps carrier. SR3 supports direct sequence spreading only.
Diversity	'NTD' | 'OTD' | 'STS'	Transmit diversity type (applicable only for SR1), where NTD is no transmit diversity, OTD is orthogonal transmit diversity, and STS is space time spreading
QOF	'QOF1' | 'QOF2' | 'QOF3'	Quasi-orthogonal function type
PNOffset	Nonnegative scalar integer	PN offset of the base station
LongCodeState	Positive scalar integer	Initial long code state
PowerNormalization	'Off' | 'NormalizeTo0dB' | 'NoiseFillTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients, used only when the FilterType field is set to 'Custom'
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
FPICH	Structure	See **FPICH Substructure**. Optional.
FAPICH	Structure	See **FAPICH Substructure**. Optional.
FTDPICH	Structure	See **FTDPICH Substructure**. Optional.
FATDPICH	Structure	See **FATDPICH Substructure**. Optional.
FSYNC	Structure	See **FSYNC Substructure**. Optional.
FPCH	Structure	See **FPCH Substructure**. Optional.
FCCCH	Structure	See **FCCCH Substructure**. Optional.
FCACH	Structure	See **FCACH Substructure**. Optional.
FQPCH	Structure	See **FQPCH Substructure**. Optional.
FCPCCH	Structure	See **FCPCCH Substructure**. Optional.
FBCCH	Structure	See **FBCCH Substructure**. Optional.
FFCH	Structure	See **FFCH Substructure**. Optional.
FDCCH	Structure	See **FDCCH Substructure**. Optional.
FSCCH	Structure	See **FSCCH Substructure**. Optional.

Parameter Field	Values	Description
FSCH	Structure	See **FSCH Substructure**. Optional.
FOCNS	Structure	See **FOCNS Substructure**. Optional.

FPICH Substructure

Include the FPICH substructure in the cfg structure to configure the forward pilot channel (F-PICH). The FPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FAPICH Substructure

Include the FAPICH substructure in the cfg structure to configure the forward auxiliary pilot channel (F-APICH). The FAPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256 | 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FTDPICH Substructure

Include the FTDPICH substructure in the cfg structure to configure the forward transmit diversity pilot Channel (F-TDPICH). The FTDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FATDPICH

Include the FATDPICH substructure in the cfg structure to configure the forward auxiliary transmit diversity pilot channel (F-ATDPICH). The FATDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256, 512	Walsh code length

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FSYNC Substructure

Include the FSYNC substructure in the cfg structure to configure the forward sync channel (F-SYNC). The FSYNC substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or 'SyncMessage'. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or a 'SyncMessage' character vector.
SyncMessage	Structure	See **SyncMessage Substructure**. Optional.

SyncMessage Substructure

If the DataSource field of the FSYNC substructure is set to 'SyncMessage', add the SyncMessage substructure to the cfg.FSYNC substructure to configure the sync channel message. The SyncMessage substructure contains these fields.

Parameter Field	Typical Value	Description
P_REV	6	Protocol revision
MIN_P_REV	6	Minimum protocol revision
SID	hex2dec('14B')	System identifier
NID	1	Network identification
PILOT_PN	0	Pilot PN offset
LC_STATE	hex2dec('20000000000')	Long code state
SYS_TIME	hex2dec('36AE0924C')	System time
LP_SEC	0	Leap second
LTM_OFF	0	Local time offset
DAYLT	0	Daylight saving time indicator
PRAT	0	Paging channel data rate
CDMA_FREQ	hex2dec('2F6')	CDMA frequency
EXT_CDMA_FREQ	hex2dec('2F6')	Extended CDMA frequency

FPCH Substructure

Include the FPCH substructure in the cfg substructure to configure the forward paging channel (F-PCH). The FPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600	Data rate (bps)
LongCodeMask	Positive scalar integer	Long code identifier
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ 7	Walsh code number
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or a paging message character vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'. Paging message options include 'PagingMessage1', 'PagingMessage2', and 'PagingMessage3'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or one of three paging messages. For a description of paging message contents see footnote 1.

[1] When the DataSource enumeration specifies one of the paging message options, simulated paging message data is used as input to the F-PCH physical channel:

- 'PagingMessage1' — Streams a 7680 bit sequence (800 ms at fullrate) of paging message contents onto the channel that includes the General Page Message, the CDMA Channel List Message, the Extended System Parameter Message, the Extended Neighbor List Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a nonsequential pattern.
- 'PagingMessage2' — Streams a 2304 bit sequence (240 ms at fullrate) of paging message contents onto the channel that includes a truncated version of the full 'PagingMessage1' content.
- 'PagingMessage3' — Streams an 864 bit sequence (90 ms at fullrate) of paging message contents onto the channel that includes the Neighbor List Message, the CDMA Channel List Message, the General Page Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a sequential pattern.

For more information on the F-PCH contents, refer to 3GPP2 C.S0004, Table 3.1.2.3.1.1.2–1.

FCCCH Substructure

Include the FCCCH substructure in the cfg structure to configure the forward common control channel (F-CCCH). The FCCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On'|'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCACH Substructure

Include the FCACH substructure in the cfg structure to configure the forward common assignment channel (F-CACH). The FCACH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FQPCH Substructure

Include the FQPCH substructure in the cfg structure to configure the forward quick paging channel (F-QPCH). The FQPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	2400 | 4800	Data rate (bps)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCPCCH Substructure

Include the FCPCCH substructure in the cfg structure to configure the forward common power control channel (F-CPCCH). The FCPCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 63	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FBCCH Substructure

Include the FBCCH substructure in the cfg structure to configure the forward broadcast control channel (F-BCCH). The FBCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
DataRate	4800 | 9600 | 19200	Data rate (bps)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 127	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FFCH Substructure

Include the FFCH substructure in the cfg structure to configure the forward fundamental traffic channel (F-FCH). The FFCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' through 'RC9'	Radio configuration channel
DataRate	1200 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel

Parameter Field	Values	Description
PowerControlPower	Real scalar	Power control subchannel power (relative to F-FCH)
PowerControlDataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

FDCCH Substructure

Include the FDCCH substructure in the cfg structure to configure the forward dedicated control channel (F-DCCH). The FDCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' through 'RC9'	Radio configuration channel
DataRate	9600 | 14400	Data rate (bps)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCCH Substructure

Include the FSCCH substructure in the cfg structure to configure the forward supplemental code channel (F-SCCH). The FSCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' | 'RC2'	Radio configuration channel

Parameter Field	Values	Description
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCH Substructure

Include the `FSCH` substructure in the `cfg` structure to configure the forward supplemental channel (F-SCH). The `FSCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`RadioConfiguration`	`'RC3'` | `'RC4'` | `'RC5'` | `'RC6'` | `'RC7'` | `'RC8'` | `'RC9'`	Radio configuration channel
`DataRate`	`1200` | `1350` | `1500` | `1800` | `2400` | `2700` | `3600` | `4800` | `7200` | `9600` | `14400` | `19200` | `28800` | `38400` | `57600` | `76800` | `115200` | `153600` | `230400` | `307200`	Data rate (bps)
`FrameLength`	`20` | `40` | `80`	Frame length (ms)
`CodingType`	`'Conv'` | `'Turbo'`	Channel coding type, convolutional or turbo
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableQOF`	`'On'` | `'Off'`	Enable QOF spreading
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FOCNS Substructure

Include the FOCNS substructure in the cfg structure to configure orthogonal channel noise source information. The FOCNS substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*.

[2] 3GPP2 C.S0010–C v2.0. “Recommended Minimum Performance Standards for cdma2000 Spread Spectrum Base Stations.” *3rd Generation Partnership Project 2*.

[3] 3GPP2 C.S0004–F v1.0. “Signaling Link Access Control (LAC) Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*.

See Also

cdma2000ForwardWaveformGenerator | cdma2000ReverseReferenceChannels

Introduced in R2015b

cdma2000ForwardWaveformGenerator

Generate cdma2000 forward link waveform

Syntax

```
[waveform1,waveform2] = cdma2000ForwardWaveformGenerator(cfg)
```

Description

`[waveform1,waveform2] = cdma2000ForwardWaveformGenerator(cfg)` returns the cdma2000 forward link baseband primary waveform, `waveform1`, and the forward link diversity waveform, `waveform2`, as defined by the parameter definition structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties the function uses to generate a cdma2000 waveform. You can generate `cfg` by using the `cdma2000ForwardReferenceChannels` function. The top-level parameters of `cfg` are `SpreadingRate`, `Diversity`, `QOF`, `PNOffset`, `LongCodeState`, `PowerNormalization`, `CustomFilterCoefficients`, `OversamplingRatio`, `FilterType`, `InvertQ`, `EnableModulation`, `ModulationFrequency`, and `NumChips`. To enable specific channels, add their associated substructures, for example, the forward paging channel, `FPCH`.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all combinations of spreading rate, radio configuration, frame length, and data rate are supported. To ensure that the input argument is valid, use the `cdma2000ForwardReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate Waveform for RC2 Forward Traffic Channels

Create a parameter structure, `config`, for all forward traffic channels (F-FCH and F-SCCH) that are supported by radio configuration 2.

```
config = cdma2000ForwardReferenceChannels('ALL-RC2')
```

```
config = struct with fields:
          SpreadingRate: 'SR1'
              Diversity: 'NTD'
                    QOF: 'QOF1'
               PNOffset: 0
          LongCodeState: 0
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
```

```
               NumChips: 1000
                  FPICH: [1x1 struct]
                 FAPICH: [1x1 struct]
                FTDPICH: [1x1 struct]
               FATDPICH: [1x1 struct]
                   FPCH: [1x1 struct]
                  FSYNC: [1x1 struct]
                  FBCCH: [1x1 struct]
                  FCACH: [1x1 struct]
                  FCCCH: [1x1 struct]
                 FCPCCH: [1x1 struct]
                  FQPCH: [1x1 struct]
                   FFCH: [1x1 struct]
                  FOCNS: [1x1 struct]
                  FSCCH: [1x1 struct]
```

Examine the fields for the Forward Fundamental Channel (F-FCH). The data rate is 14,400 bps and the frame length is 20 ms.

```
config.FFCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC2'
              DataRate: 14400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 7
             EnableQOF: 'Off'
    PowerControlEnable: 'Off'
```

Generate the complex waveform using the corresponding waveform generator function.

```
waveform = cdma2000ForwardWaveformGenerator(config);
```

A waveform composed of the channels specified by each substructure of `config` is generated by `cdma2000ForwardWaveformGenerator`.

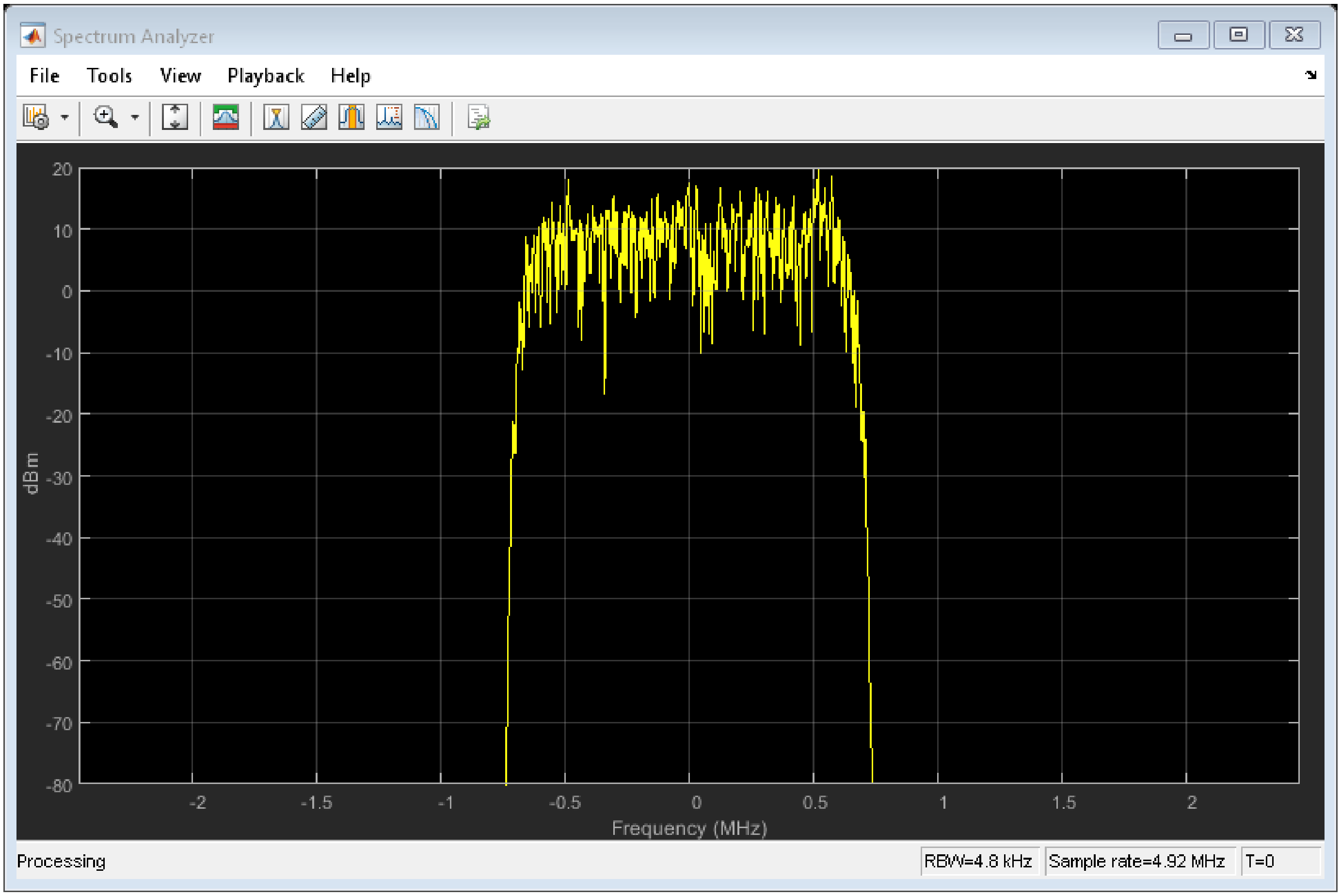
Generate Forward Traffic Channel for RC4

Configure a cdma2000 forward link supporting a 307.2 kbps forward supplemental channel (F-SCH) using radio configuration 4.

```
config = cdma2000ForwardReferenceChannels('TRAFFIC-RC4-4800',5000, ...
    'F-SCH-307200-20');
```

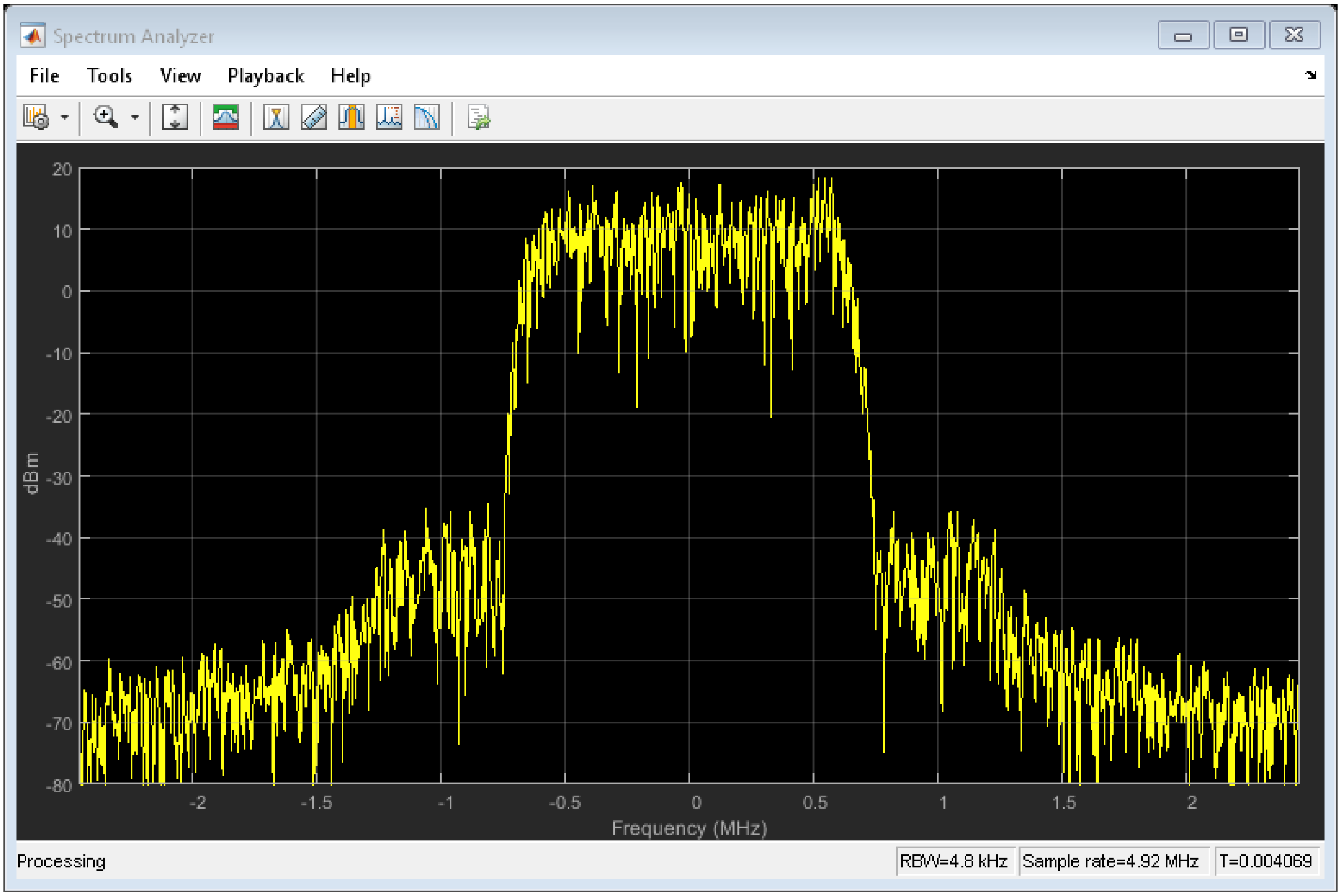
Generate the waveform and plot its spectrum. The sample rate is equal to the product of the chip rate and the oversampling ratio. RC4 uses spreading rate 1, which is equivalent to a 1.2288 Mcps chip rate.

```
wv = cdma2000ForwardWaveformGenerator(config);
fs = 1.2288e6 * config.OversamplingRatio;

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Change the filter type to `'cdma2000Short'` and plot the spectrum.

```
config.FilterType = 'cdma2000Short';
wv = cdma2000ForwardWaveformGenerator(config);
step(sa,wv)
```



The 'cdma2000Short' filter does not provide as much out-of-band attenuation as does the 'cdma2000Long' filter.

Generate cdma2000 Waveform with Two Forward Supplemental Channels

Create a parameter structure that specifies a forward traffic channel. Use it to generate a forward channel waveform.

Create a parameter structure specifying a traffic channel consisting of a 4800 bps fundamental channel, 5000 chips, and a 614.4 kbps supplemental channel (F-SCH) having a 20 ms frame duration.

```
cfg = cdma2000ForwardReferenceChannels('TRAFFIC-RC7-4800', ...
    5000,'F-SCH-614400-20');
```

Based on the first F-SCH, create a second F-SCH.

```
cfg(2).FSCH = cfg.FSCH;
```

Set the data rate of the second F-SCH to 38.4 kbps. Set the frame duration to 40 ms.

```
cfg(2).FSCH.DataRate = 38400;
cfg(2).FSCH.FrameLength = 40;
cfg.FSCH
```

```
ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC7'
              DataRate: 614400
           FrameLength: 20
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 2
             EnableQOF: 'Off'
            CodingType: 'conv'


ans = struct with fields:
                Enable: 'On'
                 Power: 0
    RadioConfiguration: 'RC7'
              DataRate: 38400
           FrameLength: 40
          LongCodeMask: 0
          EnableCoding: 'On'
            DataSource: {'PN9'  [1]}
             WalshCode: 2
             EnableQOF: 'Off'
            CodingType: 'conv'
```

Set the Walsh code of the second F-SCH so that it is not identical to the Walsh code of the first F-SCH.

```
cfg(2).FSCH.WalshCode = 3;
```

Generate the forward link waveform.

```
wv = cdma2000ForwardWaveformGenerator(cfg);
```

Input Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
`SpreadingRate`	`'SR1'` | `'SR3'`	Spreading rate of the waveform. `SR1` corresponds to a 1.2288 Mcps carrier. `SR3` corresponds to a 3.6864 Mcps carrier. SR3 supports direct sequence spreading only.

Parameter Field	Values	Description
Diversity	'NTD' | 'OTD' | 'STS'	Transmit diversity type (applicable only for SR1), where NTD is no transmit diversity, OTD is orthogonal transmit diversity, and STS is space time spreading
QOF	'QOF1' | 'QOF2' | 'QOF3'	Quasi-orthogonal function type
PNOffset	Nonnegative scalar integer	PN offset of the base station
LongCodeState	Positive scalar integer	Initial long code state
PowerNormalization	'Off' | 'NormalizeTo0dB' | 'NoiseFillTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients, used only when the FilterType field is set to 'Custom'
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
FPICH	Structure	See **FPICH Substructure**. Optional.
FAPICH	Structure	See **FAPICH Substructure**. Optional.
FTDPICH	Structure	See **FTDPICH Substructure**. Optional.
FATDPICH	Structure	See **FATDPICH Substructure**. Optional.
FSYNC	Structure	See **FSYNC Substructure**. Optional.
FPCH	Structure	See **FPCH Substructure**. Optional.
FCCCH	Structure	See **FCCCH Substructure**. Optional.
FCACH	Structure	See **FCACH Substructure**. Optional.
FQPCH	Structure	See **FQPCH Substructure**. Optional.
FCPCCH	Structure	See **FCPCCH Substructure**. Optional.
FBCCH	Structure	See **FBCCH Substructure**. Optional.
FFCH	Structure	See **FFCH Substructure**. Optional.
FDCCH	Structure	See **FDCCH Substructure**. Optional.
FSCCH	Structure	See **FSCCH Substructure**. Optional.
FSCH	Structure	See **FSCH Substructure**. Optional.
FOCNS	Structure	See **FOCNS Substructure**. Optional.

FPICH Substructure

Include the FPICH substructure in the cfg structure to configure the forward pilot channel (F-PICH). The FPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FAPICH Substructure

Include the FAPICH substructure in the cfg structure to configure the forward auxiliary pilot channel (F-APICH). The FAPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256 | 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FTDPICH Substructure

Include the FTDPICH substructure in the cfg structure to configure the forward transmit diversity pilot Channel (F-TDPICH). The FTDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

FATDPICH

Include the FATDPICH substructure in the cfg structure to configure the forward auxiliary transmit diversity pilot channel (F-ATDPICH). The FATDPICH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256, 512	Walsh code length
WalshCode	Nonnegative integer scalar, such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier

FSYNC Substructure

Include the FSYNC substructure in the cfg structure to configure the forward sync channel (F-SYNC). The FSYNC substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed}, binary vector, or 'SyncMessage'. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or a 'SyncMessage' character vector.
SyncMessage	Structure	See **SyncMessage Substructure**. Optional.

SyncMessage Substructure

If the DataSource field of the FSYNC substructure is set to 'SyncMessage', add the SyncMessage substructure to the cfg.FSYNC substructure to configure the sync channel message. The SyncMessage substructure contains these fields.

Parameter Field	Typical Value	Description
P_REV	6	Protocol revision
MIN_P_REV	6	Minimum protocol revision
SID	hex2dec('14B')	System identifier
NID	1	Network identification
PILOT_PN	0	Pilot PN offset
LC_STATE	hex2dec('20000000000')	Long code state
SYS_TIME	hex2dec('36AE0924C')	System time
LP_SEC	0	Leap second
LTM_OFF	0	Local time offset
DAYLT	0	Daylight saving time indicator
PRAT	0	Paging channel data rate
CDMA_FREQ	hex2dec('2F6')	CDMA frequency
EXT_CDMA_FREQ	hex2dec('2F6')	Extended CDMA frequency

FPCH Substructure

Include the FPCH substructure in the cfg substructure to configure the forward paging channel (F-PCH). The FPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	4800 | 9600	Data rate (bps)
LongCodeMask	Positive scalar integer	Long code identifier

Parameter Field	Values	Description
`WalshCode`	Nonnegative integer scalar, such that 0 ≤ `WalshCode` ≤ 7	Walsh code number
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}`, binary vector, or a paging message character vector. Standard PN types are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`. Paging message options include `'PagingMessage1'`, `'PagingMessage2'`, and `'PagingMessage3'`.	Data source. Specify a standard PN sequence with a random number seed, a binary vector, or one of three paging messages. For a description of paging message contents see footnote 1.

[1] When the `DataSource` enumeration specifies one of the paging message options, simulated paging message data is used as input to the F-PCH physical channel:

- `'PagingMessage1'` — Streams a 7680 bit sequence (800 ms at fullrate) of paging message contents onto the channel that includes the General Page Message, the CDMA Channel List Message, the Extended System Parameter Message, the Extended Neighbor List Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a nonsequential pattern.
- `'PagingMessage2'` — Streams a 2304 bit sequence (240 ms at fullrate) of paging message contents onto the channel that includes a truncated version of the full `'PagingMessage1'` content.
- `'PagingMessage3'` — Streams an 864 bit sequence (90 ms at fullrate) of paging message contents onto the channel that includes the Neighbor List Message, the CDMA Channel List Message, the General Page Message, the System Parameter Message, and the Access Parameter Message. The paging message repeats these messages in a sequential pattern.

For more information on the F-PCH contents, refer to 3GPP2 C.S0004, Table 3.1.2.3.1.1.2–1.

FCCCH Substructure

Include the `FCCCH` substructure in the `cfg` structure to configure the forward common control channel (F-CCCH). The `FCCCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'`| `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`DataRate`	`9600` | `19200` | `38400`	Data rate (bps)
`FrameLength`	`5` | `10` | `20`	Frame length (ms)
`CodingType`	`'conv'` | `'turbo'`	Type of error correction coding

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCACH Substructure

Include the FCACH substructure in the cfg structure to configure the forward common assignment channel (F-CACH). The FCACH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
CodingType	'conv' | 'turbo'	Type of error correction coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FQPCH Substructure

Include the FQPCH substructure in the cfg structure to configure the forward quick paging channel (F-QPCH). The FQPCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	2400 | 4800	Data rate (bps)

Parameter Field	Values	Description
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FCPCCH Substructure

Include the `FCPCCH` substructure in the `cfg` structure to configure the forward common power control channel (F-CPCCH). The `FCPCCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 63	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FBCCH Substructure

Include the `FBCCH` substructure in the `cfg` structure to configure the forward broadcast control channel (F-BCCH). The `FBCCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`DataRate`	`4800` | `9600` | `19200`	Data rate (bps)
`CodingType`	`'conv'` | `'turbo'`	Type of error correction coding

Parameter Field	Values	Description
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 127	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FFCH Substructure

Include the `FFCH` substructure in the `cfg` structure to configure the forward fundamental traffic channel (F-FCH). The `FFCH` substructure contains these fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`RadioConfiguration`	`'RC1'` through `'RC9'`	Radio configuration channel
`DataRate`	`1200` | `1500` | `1800` | `2400` | `2700` | `3600` | `4800` | `7200` | `9600` | `14400`	Data rate (bps)
`FrameLength`	`5` | `10` | `20`	Frame length (ms)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 255	Walsh code number
`LongCodeMask`	Positive scalar integer	Long code identifier
`EnableQOF`	`'On'` | `'Off'`	Enable QOF spreading
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array: `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
`PowerControlEnable`	`'On'` | `'Off'`	Enable or disable power control subchannel
`PowerControlPower`	Real scalar	Power control subchannel power (relative to F-FCH)

Parameter Field	Values	Description
PowerControlDataSource	Cell array: {'PN Type', RNSeed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

FDCCH Substructure

Include the FDCCH substructure in the cfg structure to configure the forward dedicated control channel (F-DCCH). The FDCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' through 'RC9'	Radio configuration channel
DataRate	9600 | 14400	Data rate (bps)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RNSeed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCCH Substructure

Include the FSCCH substructure in the cfg structure to configure the forward supplemental code channel (F-SCCH). The FSCCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC1' | 'RC2'	Radio configuration channel

Parameter Field	Values	Description
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FSCH Substructure

Include the FSCH substructure in the cfg structure to configure the forward supplemental channel (F-SCH). The FSCH substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
RadioConfiguration	'RC3' | 'RC4' | 'RC5' | 'RC6' | 'RC7' | 'RC8' | 'RC9'	Radio configuration channel
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 307200	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
CodingType	'Conv' | 'Turbo'	Channel coding type, convolutional or turbo
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 255	Walsh code number
LongCodeMask	Positive scalar integer	Long code identifier
EnableQOF	'On' | 'Off'	Enable QOF spreading
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

FOCNS Substructure

Include the FOCNS substructure in the cfg structure to configure orthogonal channel noise source information. The FOCNS substructure contains these fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
WalshLength	64 | 128 | 256	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number

Output Arguments

waveform1 — Modulated baseband waveform comprising the primary physical channels
complex vector array

Modulated baseband waveform comprising the primary cdma2000 physical channels, returned as a complex vector array.

waveform2 — Modulated baseband waveform comprising the diversity physical channels
complex vector array

Modulated baseband waveform comprising the diversity cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*.

[2] 3GPP2 C.S0004–F v1.0. “Signaling Link Access Control (LAC) Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*.

See Also

cdma2000ForwardReferenceChannels | cdma2000ReverseWaveformGenerator

Introduced in R2015b

cdma2000ReverseReferenceChannels

Define cdma2000 reverse reference channel

Syntax

```
cfg = cdma2000ReverseReferenceChannels(wv)
cfg = cdma2000ReverseReferenceChannels(wv,numchips)
cfg = cdma2000ReverseReferenceChannels(traffic,numchips,R-SCH-SPEC)
```

Description

`cfg = cdma2000ReverseReferenceChannels(wv)` returns a structure, `cfg`, that defines the cdma2000 reverse link parameters given the input waveform identifier, `wv`. Pass the structure to the `cdma2000ReverseWaveformGenerator` function to generate a reverse link reference channel waveform.

For all syntaxes, `cdma2000ReverseReferenceChannels` creates a configuration structure that is compliant with the physical layer specification for cdma2000 systems described in [1].

`cfg = cdma2000ReverseReferenceChannels(wv,numchips)` specifies the number of chips to generate.

`cfg = cdma2000ReverseReferenceChannels(traffic,numchips,R-SCH-SPEC)` returns `cfg` for the specified traffic channel, `traffic`, and the reverse supplemental channel (R-SCH) and frame length combination, `R-SCH-SPEC`.

Examples

Generate Reverse Common Control Channel Waveform

Generate the structure corresponding to the reverse common control channel (R-CCCH) having a 19,200 bps data rate and 10 ms frames.

```
config = cdma2000ReverseReferenceChannels('R-CCCH-19200-10');
```

Verify that the R-CCCH substructure is configured for the correct data rate and frame duration.

```
config.RCCCH
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 19200
     FrameLength: 10
       WalshCode: 1
```

Generate the reverse channel waveform using the corresponding waveform generator function, `cdma2000ReverseWaveformGenerator`.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate Reverse Channels for RC1 and RC6

Create a configuration structure to generate all possible channels associated with radio configuration 1 in which the number of chips is specified as 2500.

```
config = cdma2000ReverseReferenceChannels('ALL-RC1',2500)
```

```
config = struct with fields:
     RadioConfiguration: 'RC1'
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2500
                   RFCH: [1x1 struct]
                   RACH: [1x1 struct]
                  RSCCH: [1x1 struct]
```

The structure contains substructures corresponding to the R-FCH, R-ACH, and R-SCCH channels.

Call the function again using radio configuration 6.

```
config = cdma2000ReverseReferenceChannels('ALL-RC6',2500)
```

```
config = struct with fields:
     RadioConfiguration: 'RC6'
     PowerNormalization: 'Off'
      OversamplingRatio: 4
             FilterType: 'cdma2000Long'
                InvertQ: 'Off'
       EnableModulation: 'Off'
    ModulationFrequency: 0
               NumChips: 2500
                   RFCH: [1x1 struct]
                  RPICH: [1x1 struct]
                  REACH: [1x1 struct]
                  RCCCH: [1x1 struct]
                  RDCCH: [1x1 struct]
                  RSCH1: [1x1 struct]
                  RSCH2: [1x1 struct]
```

The channels supported by RC6 differ from those supported by RC1. They include R-FCH, R-PICH, R-EACH, R-CCCH, R-DCCH, R-SCH1, and R-SCH2.

Create the waveform corresponding to the set of RC6 channels.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate Reverse Supplemental Channel

Create a configuration structure using radio configuration 3 with a reverse fundamental channel (R-FCH). Specify a 2700 bps data rate and a reverse supplemental channel (R-SCH) having a 76,800 bps data rate and an 80 ms frame length.

```
config = cdma2000ReverseReferenceChannels('TRAFFIC-RC3-2700',2000, ...
    'R-SCH-76800-80');
```

Verify that the R-FCH data rate is 2700 bps and the first R-SCH data rate is 76,800 bps with an 80 ms frame length.

```
config.RFCH.DataRate
```

```
ans = 2700
```

```
config.RSCH1.DataRate
```

```
ans = 76800
```

```
config.RSCH1.FrameLength
```

```
ans = 80
```

Generate the corresponding waveform.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector. The input typically identifies the channel type, radio configuration, data rate, and frame length. To specify wv, connect the substrings with hyphens, for example, 'TRAFFIC-RC2-3600'.

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
wv	'R-PICH-ONLY'			Generates a waveform containing a pilot channel only.
	'R-CCCH'	9600	20	Character vector representing the Reverse Common Control Channel (R-CCCH) data rate in bps and the frame length in
		19200	10 | 20	

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
		38400	5 | 10 | 20	ms. Specify 'R-CCCH-9600-20' to create a structure variable, wv, with a 9600 bps R-CCCH data rate and a 20 ms frame length.
	'TRAFFIC'	RC1	1200 | 2400 | 4800 | 9600	Character vector representing the radio configuration and the Reverse Fundamental Channel (R-FCH) data rate in bps. Specify 'TRAFFIC-RC6-14400', corresponds to radio configuration 6 with a 14400 bps R-FCH data rate.
		RC2 | RC4 | RC6	1800 | 3600 | 7200 | 14400	
		RC3 | RC5 | RC6	1500 | 2700 | 4800 | 9600	
	'R-EACH'	9600	20	Reverse Enhanced Access Channel waveforms. Specify 'R-EACH-38400-5' to create a structure corresponding to an R-EACH channel with a 38400 bps data rate and a 5 ms frame length.
		19200	10 | 20	
		38400	5 | 10 | 20	
	'R-PICH-R-FCH'			Specify tests for the mobile transmitter in accordance with [2].

Parameter Field	Values			Description
	Substring 1	Substring 2	Substring 3	
	`'ALL'`	RC1 | RC2 | RC3 | RC4 | RC5 | RC6	N/A	Returns all channels that are supported for the specified radio configuration. Specify `'ALL-RC4'` to create a structure containing all traffic channels for radio configuration 4.

Example: `'R-CCCH-9600-20'` is a R-CCH channel having a 9600 bps data rate and a 20 ms frame length.

Example: `'R-EACH-38400-5'` is a R-EACH channel having a 38,400 bps data rate and a 5 ms frame length.

Data Types: `char`

`numchips` — Number of chips

`1000` (default) | positive integer scalar

Number of chips, specified as a positive integer.

Example: `2048`

Data Types: `double`

`traffic` — Traffic configuration

character vector

Traffic channel configuration, specified as a character vector. The table shows the valid configurations.

Radio Configuration	Traffic Channel Configuration			
1	`'TRAFFIC-RC1-1200'`	`'TRAFFIC-RC1-2400'`	`'TRAFFIC-RC1-4800'`	`'TRAFFIC-RC1-9600'`
2	`'TRAFFIC-RC2-1800'`	`'TRAFFIC-RC2-3600'`	`'TRAFFIC-RC2-7200'`	`'TRAFFIC-RC2-14400'`
3	`'TRAFFIC-RC3-1500'`	`'TRAFFIC-RC3-2700'`	`'TRAFFIC-RC3-4800'`	`'TRAFFIC-RC3-9600'`
4	`'TRAFFIC-RC4-1800'`	`'TRAFFIC-RC4-3600'`	`'TRAFFIC-RC4-7200'`	`'TRAFFIC-RC4-14400'`
5	`'TRAFFIC-RC5-1500'`	`'TRAFFIC-RC5-2700'`	`'TRAFFIC-RC5-4800'`	`'TRAFFIC-RC5-9600'`

Radio Configuration	Traffic Channel Configuration			
6	'TRAFFIC-RC6-1800'	'TRAFFIC-RC6-3600'	'TRAFFIC-RC6-7200'	'TRAFFIC-RC6-14400'

Example: `'TRAFFIC-RC4-1800'` is a traffic channel using radio configuration 4 and having an R-FCH with an 1800 bps data rate .

Data Types: `char`

`R-SCH-SPEC` — Reverse Supplemental Channel data rate and frame length
character vector

Specify the R-SCH data rate and frame length as a character vector. If omitted, `R-SCH-SPEC` defaults to the lowest R-SCH data rate allowable for a 20 ms frame length given the radio configuration specified by `traffic`. The table summarizes the supported data rate and frame length combinations.

Radio Configuration	Frame Length		
	20 ms	40 ms	80 ms
3 | 5	'R-SCH-1500-20' | 'R-SCH-2700-20' | 'R-SCH-4800-20' | 'R-SCH-9600-20' | 'R-SCH-19200-20' | 'R-SCH-38400-20' | 'R-SCH-76800-20' | 'R-SCH-153600-20' | 'R-SCH-307200-20'	'R-SCH-1350-40' | 'R-SCH-2400-40' | 'R-SCH-4800-40' | 'R-SCH-9600-40' | 'R-SCH-19200-40' | 'R-SCH-38400-40' | 'R-SCH-76800-40'| 'R-SCH-153600-40'	'R-SCH-1350-80' | 'R-SCH-2400-80' | 'R-SCH-4800-80' | 'R-SCH-9600-80' | 'R-SCH-19200-80' | 'R-SCH-38400-80' | 'R-SCH-76800-80'
5	'R-SCH-614400-20'	'R-SCH-307200-40'	'R-SCH-153600-80'
4 | 6	'R-SCH-1800-20' | 'R-SCH-3600-20' | 'R-SCH-7200-20' | 'R-SCH-14400-20' | 'R-SCH-28800-20' | 'R-SCH-57600-20' | 'R-SCH-115200-20' | 'R-SCH-230400-20'	'R-SCH-1800-40' | 'R-SCH-3600-40' | 'R-SCH-7200-40' | 'R-SCH-14400-40' | 'R-SCH-28800-40' | 'R-SCH-57600-40' | 'R-SCH-115200-40'	'R-SCH-1800-80' | 'R-SCH-3600-80' | 'R-SCH-7200-80' | 'R-SCH-14400-80' | 'R-SCH-28800-80' | 'R-SCH-57600-80'
6	'R-SCH-460800-20' | 'R-SCH-1036800-20'	'R-SCH-230400-40' | 'R-SCH-518400-40'	'R-SCH-115200-80' | 'R-SCH-259200-80'

Additional data rate information for the cdma2000 reverse links is given in Tables 2.1.3.1.3-1 and 2.1.3.1.3-2 of [1].

Example: `'R-SCH-153600-20'` is an R-SCH having a 153,600 bps data rate and a 20 ms frame length.

Data Types: `char`

Output Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator
structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
RadioConfiguration	'RC1' | 'RC2' | 'RC3' | 'RC4' | 'RC5' | 'RC6'	Radio configuration of the reverse channel. The spreading rate of the waveform is derived from the radio configuration. Spreading rate 1, SR1, corresponds to a 1.2288 Mcps carrier and is associated with RC1 through RC4. Spreading rate 3, SR3, corresponds to a 3.6864 Mcps carrier and is associated with RC5 and RC6.
PowerNormalization	'Off' | 'NormalizeTo0dB'	Power normalization of the waveform
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Off' | 'Custom'	Type of output filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients used only when the FilterType field is set to 'Custom'
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
RPICH	Structure	See **RPICH Substructure**. Optional.
RACH	Structure	See **RACH Substructure**. Optional.
REACH	Structure	See **REACH Substructure**. Optional.
RCCCH	Structure	See **RCCCH Substructure**. Optional.
RDCCH	Structure	See **RDCCH Substructure**. Optional.
RFCH	Structure	See **RFCH Substructure**. Optional.
RSCCH	Structure	See **RSCCH Substructure**. Optional.
RSCH1	Structure	See **RSCH1 Substructure**. Optional.
RSCH2	Structure	See **RSCH2 Substructure**. Optional.

RPICH Substructure

Include the RPICH substructure in the cfg structure to configure the Reverse Pilot Channel (R-PICH). The RPICH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier

Parameter Field	Values	Description
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to R-PICH)
PowerControlDataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

RACH Substructure

Include the RACH substructure in the cfg structure to configure the Reverse Access Channel (R-ACH). The RACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

REACH Substructure

Include the REACH substructure in the cfg structure to configure the Reverse Enhanced Access Channel (R-EACH). The REACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

RCCCH Substructure

Include the RCCCH substructure in the cfg structure to configure the Reverse Common Control Channel (R-CCCH). The RCCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error control coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RDCCH Substructure

Include the RDCCH substructure in the cfg structure to configure the Reverse Dedicated Control Channel (R-DCCH). The RDCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RFCH Substructure

Include the RFCH substructure in the cfg structure to configure the Reverse Fundamental Traffic Channel (R-FCH). The RFCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCCH Substructure

Include the RSCCH substructure in the cfg structure to configure the Reverse Supplemental Code Channel (R-SCCH). The RSCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH1 Substructure

Include the RSCH1 substructure in the cfg structure to configure the Reverse Supplemental Channel 1 (R-SCH 1). The RSCH1 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	2 | 4	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH2 Substructure

Include the RSCH2 substructure in the cfg structure to configure the Reverse Supplemental Channel 2 (R-SCH 2). The RSCH2 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	4 | 8	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

Data Types: struct

References

[1] 3GPP2 C.S0002–F v2.0. "Physical Layer Standard for cdma2000 Spread Spectrum Systems." *3rd Generation Partnership Project 2*.

[2] 3GPP2 C.S0011–E v2.0. "Recommended Minimum Performance Standards for cdma2000 Spread Spectrum Mobile Stations." *3rd Generation Partnership Project 2*.

See Also

cdma2000ForwardReferenceChannels | cdma2000ReverseWaveformGenerator

Introduced in R2015b

cdma2000ReverseWaveformGenerator

Generate cdma2000 reverse link waveform

Syntax

```
waveform = cdma2000ReverseWaveformGenerator(cfg)
```

Description

`waveform = cdma2000ReverseWaveformGenerator(cfg)` returns the cdma2000 reverse link baseband waveform, `waveform` as defined by the parameter configuration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties used by the function to generate a cdma2000 waveform. You can generate the input argument by using the `cdma2000ReverseReferenceChannels` function. The top-level parameters of `cfg` are `RadioConfiguration`, `LongCodeState`, `PowerNormalization`, `OversamplingRatio`, `FilterType`, `InvertQ`, `EnableModulation`, `ModulationFrequency`, and `NumChips`. To enable specific channels, add their associated substructures, for example, the reverse dedicated control channel, `RDCCH`.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all combinations of spreading rate, radio configuration, frame length, and data rate are supported. To ensure that the input argument is valid, use the `cdma2000ReverseReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate Reverse Common Control Channel Waveform

Generate the structure corresponding to the reverse common control channel (R-CCCH) having a 19,200 bps data rate and 10 ms frames.

```
config = cdma2000ReverseReferenceChannels('R-CCCH-19200-10');
```

Verify that the R-CCCH substructure is configured for the correct data rate and frame duration.

```
config.RCCCH
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
     LongCodeMask: 0
     EnableCoding: 'On'
       DataSource: {'PN9'  [1]}
         DataRate: 19200
      FrameLength: 10
        WalshCode: 1
```

Generate the reverse channel waveform using the corresponding waveform generator function, `cdma2000ReverseWaveformGenerator`.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Generate R-SCH Channels for RC5

Create a configuration structure for a reverse channel having an R-FCH with a 4800 bps data rate and two R-SCHs. Specify that each R-SCH have a 153,600 bps data rate using RC5.

```
config = cdma2000ReverseReferenceChannels('TRAFFIC-RC5-4800',5000, ...
    'R-SCH-153600-40');
```

Determine the sample rate. Because RC5 corresponds to SR3, the chip rate is 3.6864 Mcps. Multiply by the oversampling ratio to obtain the sample rate.

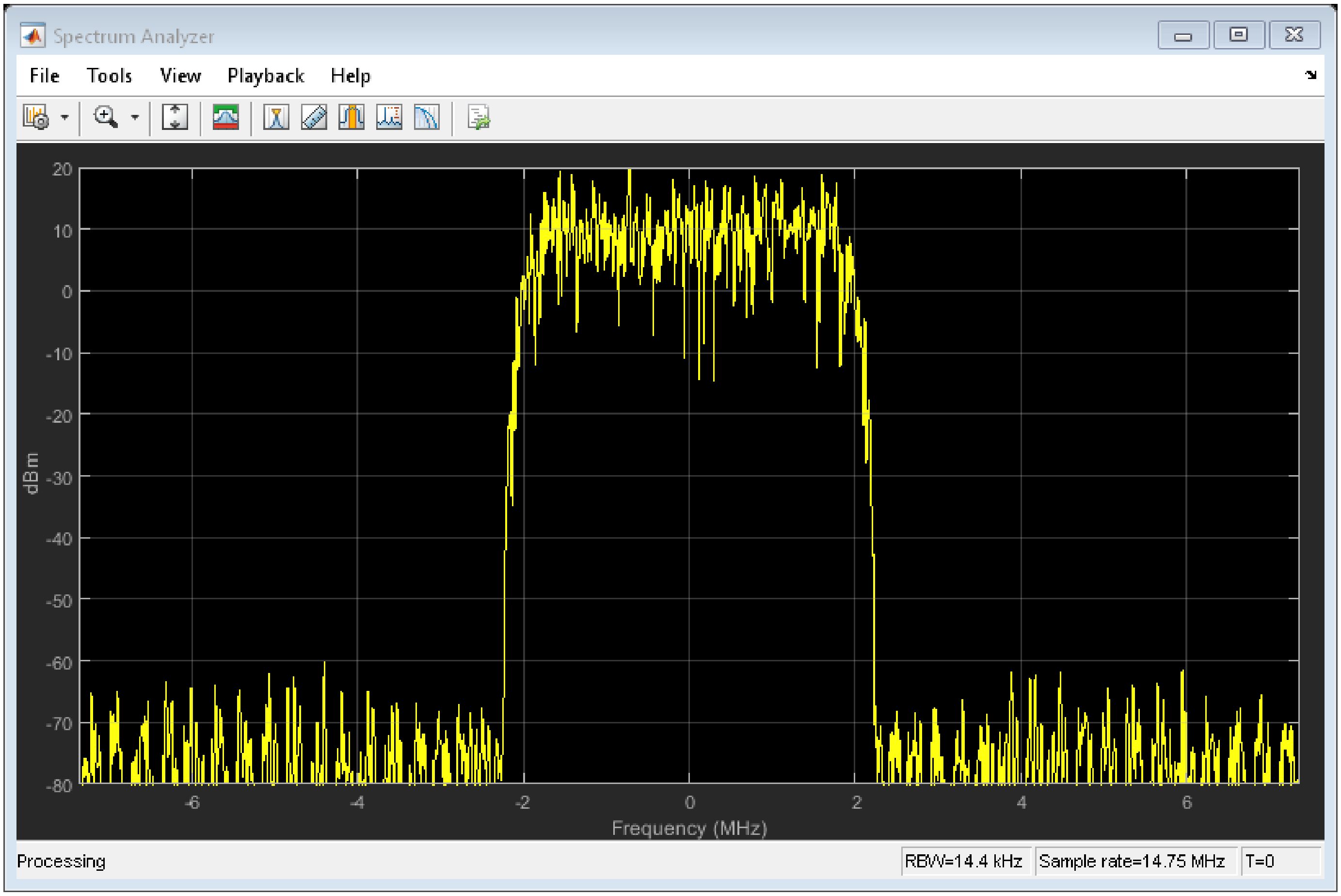
```
fs = 3.6864e6*config.OversamplingRatio;
```

Generate the reverse link waveform.

```
wv = cdma2000ReverseWaveformGenerator(config);
```

Plot the spectrum of the resultant waveform.

```
sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Generate cdma2000 Waveform with Two Reverse Supplemental Channels

Create a parameter structure specifying a reverse traffic channel containing a pair of supplemental channels and generate the corresponding waveform.

Create a parameter structure specifying a traffic channel consisting of a 14,400 bps fundamental channel, 2000 chips, and a 57,600 bps supplemental channel (R-SCH) pair having a 40 ms frame duration.

```
cfg = cdma2000ReverseReferenceChannels('TRAFFIC-RC4-14400',2000,'F-SCH-57600-40');
```

Create a second R-SCH pair by copying the R-SCH fields from the existing pair.

```
cfg(2).RSCH1 = cfg.RSCH1;
cfg(2).RSCH2 = cfg.RSCH2;
```

Set the data rate of the second R-SCH pair to 28,800 bps.

```
cfg(2).RSCH1.DataRate = 28800;
cfg(2).RSCH2.DataRate = 28800;
```

Set the Walsh codes of the second pair so that they differ from the first pair.

```
cfg(2).RSCH1.WalshCode = 4;
cfg(2).RSCH2.WalshCode = 5;
```

Verify that the data rates are set correctly and that no two supplemental channels share the same Walsh code.

```
cfg.RSCH1
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 57600
     FrameLength: 40
     WalshLength: 2
       WalshCode: 0


ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 28800
     FrameLength: 40
     WalshLength: 2
       WalshCode: 4
```

```
cfg.RSCH2
```

```
ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 57600
     FrameLength: 40
     WalshLength: 2
       WalshCode: 1


ans = struct with fields:
          Enable: 'On'
           Power: 0
    LongCodeMask: 0
    EnableCoding: 'On'
      DataSource: {'PN9'  [1]}
        DataRate: 28800
     FrameLength: 40
     WalshLength: 2
       WalshCode: 5
```

Generate the reverse link waveform.

```
wv = cdma2000ReverseWaveformGenerator(cfg);
```

Input Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
`RadioConfiguration`	`'RC1'` | `'RC2'` | `'RC3'` | `'RC4'` | `'RC5'` | `'RC6'`	Radio configuration of the reverse channel. The spreading rate of the waveform is derived from the radio configuration. Spreading rate 1, SR1, corresponds to a 1.2288 Mcps carrier and is associated with RC1 through RC4. Spreading rate 3, SR3, corresponds to a 3.6864 Mcps carrier and is associated with RC5 and RC6.
`PowerNormalization`	`'Off'` | `'NormalizeTo0dB'`	Power normalization of the waveform
`NumChips`	Positive scalar integer	Number of chips in the waveform
`OversamplingRatio`	Positive scalar integer	Oversampling ratio at output
`FilterType`	`'cdma2000Long'` | `'cdma2000Short'` | `'Off'` | `'Custom'`	Type of output filtering
`CustomFilterCoefficients`	Real vector	Custom filter coefficients used only when the `FilterType` field is set to `'Custom'`
`InvertQ`	`'Off'` | `'On'`	Negate the quadrature output
`EnableModulation`	`'Off'` | `'On'`	Enable carrier modulation
`ModulationFrequency`	Nonnegative scalar integer	Carrier modulation frequency (applies when `EnableModulation` is `'On'`)
`RPICH`	Structure	See **RPICH Substructure**. Optional.
`RACH`	Structure	See **RACH Substructure**. Optional.
`REACH`	Structure	See **REACH Substructure**. Optional.
`RCCCH`	Structure	See **RCCCH Substructure**. Optional.
`RDCCH`	Structure	See **RDCCH Substructure**. Optional.
`RFCH`	Structure	See **RFCH Substructure**. Optional.
`RSCCH`	Structure	See **RSCCH Substructure**. Optional.
`RSCH1`	Structure	See **RSCH1 Substructure**. Optional.
`RSCH2`	Structure	See **RSCH2 Substructure**. Optional.

RPICH Substructure

Include the `RPICH` substructure in the `cfg` structure to configure the Reverse Pilot Channel (R-PICH). The `RPICH` substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
PowerControlEnable	'On' | 'Off'	Enable or disable power control subchannel
PowerControlPower	Real scalar	Power control subchannel power (relative to R-PICH)
PowerControlDataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Power control subchannel data source

RACH Substructure

Include the RACH substructure in the cfg structure to configure the Reverse Access Channel (R-ACH). The RACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

REACH Substructure

Include the REACH substructure in the cfg structure to configure the Reverse Enhanced Access Channel (R-EACH). The REACH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number

Parameter Field	Values	Description
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or a binary vector. Standard PN types are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a binary vector.

RCCCH Substructure

Include the RCCCH substructure in the cfg structure to configure the Reverse Common Control Channel (R-CCCH). The RCCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	9600 | 19200 | 38400	Data rate (bps)
FrameLength	5 | 10 | 20	Frame length (ms)
CodingType	'conv' | 'turbo'	Type of error control coding
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 7	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array: {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RDCCH Substructure

Include the RDCCH substructure in the cfg structure to configure the Reverse Dedicated Control Channel (R-DCCH). The RDCCH substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
FrameLength	5 | 20	Frame length (ms)
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ 15	Walsh code number

Parameter Field	Values	Description
`LongCodeMask`	42-bit binary number	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RFCH Substructure

Include the `RFCH` substructure in the `cfg` structure to configure the Reverse Fundamental Traffic Channel (R-FCH). The `RFCH` substructure contains the following fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`DataRate`	`1200` | `1500` | `1800` | `2400` | `2700` | `3600` | `4800` | `7200` | `9600` | `14400`	Data rate (bps)
`FrameLength`	`5` | `10` | `20`	Frame length (ms)
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ 15	Walsh code number
`LongCodeMask`	42-bit binary number	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCCH Substructure

Include the `RSCCH` substructure in the `cfg` structure to configure the Reverse Supplemental Code Channel (R-SCCH). The `RSCCH` substructure contains the following fields.

Parameter Field	Values	Description
`Enable`	`'On'` | `'Off'`	Enable or disable the channel
`Power`	Real scalar	Relative channel power (dB)
`LongCodeMask`	42-bit binary number	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding

Parameter Field	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH1 Substructure

Include the RSCH1 substructure in the cfg structure to configure the Reverse Supplemental Channel 1 (R-SCH 1). The RSCH1 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)
DataRate	1200 | 1350 | 1500 | 1800 | 2400 | 2700 | 3600 | 4800 | 7200 | 9600 | 14400 | 19200 | 28800 | 38400 | 57600 | 76800 | 115200 | 153600 | 230400 | 259200 | 307200 | 460800 | 518400 | 614400 | 1036800	Data rate (bps)
FrameLength	20 | 40 | 80	Frame length (ms)
WalshLength	2 | 4	Walsh code length
WalshCode	Nonnegative integer scalar such that 0 ≤ WalshCode ≤ WalshLength – 1	Walsh code number
LongCodeMask	42-bit binary number	Long code identifier
EnableCoding	'On' | 'Off'	Enable or disable channel coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

RSCH2 Substructure

Include the RSCH2 substructure in the cfg structure to configure the Reverse Supplemental Channel 2 (R-SCH 2). The RSCH2 substructure contains the following fields.

Parameter Field	Values	Description
Enable	'On' | 'Off'	Enable or disable the channel
Power	Real scalar	Relative channel power (dB)

Parameter Field	Values	Description
`DataRate`	`1200` | `1350` | `1500` | `1800` | `2400` | `2700` | `3600` | `4800` | `7200` | `9600` | `14400` | `19200` | `28800` | `38400` | `57600` | `76800` | `115200` | `153600` | `230400` | `259200` | `307200` | `460800` | `518400` | `614400` | `1036800`	Data rate (bps)
`FrameLength`	`20` | `40` | `80`	Frame length (ms)
`WalshLength`	`4` | `8`	Walsh code length
`WalshCode`	Nonnegative integer scalar such that 0 ≤ `WalshCode` ≤ `WalshLength` – 1	Walsh code number
`LongCodeMask`	42-bit binary number	Long code identifier
`EnableCoding`	`'On'` | `'Off'`	Enable or disable channel coding
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

Output Arguments

`waveform` — Modulated baseband waveform comprising the physical channels

complex vector array

Modulated baseband waveform comprising the cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0002–F v2.0. “Physical Layer Standard for cdma2000 Spread Spectrum Systems.” *3rd Generation Partnership Project 2*.

See Also

`cdma2000ForwardWaveformGenerator` | `cdma2000ReverseReferenceChannels`

Introduced in R2015b

cma

(To be removed) Construct constant modulus algorithm (CMA) object

Note will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead.

Syntax

```
alg = cma(stepsize)
alg = cma(stepsize,leakagefactor)
```

Description

The `cma` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

Note After you use either `lineareq` or `dfe` to create a CMA equalizer object, you should initialize the equalizer object's `Weights` property with a nonzero vector. Typically, CMA is used with differential modulation; otherwise, the initial weights are very important. A typical vector of initial weights has a 1 corresponding to the center tap and 0s elsewhere.

`alg = cma(stepsize)` constructs an adaptive algorithm object based on the constant modulus algorithm (CMA) with a step size of `stepsize`.

`alg = cma(stepsize,leakagefactor)` sets the leakage factor of the CMA. `leakagefactor` must be between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.

Properties

The table below describes the properties of the CMA adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Fixed value, `'Constant Modulus'`
`StepSize`	CMA step size parameter, a nonnegative real number
`LeakageFactor`	CMA leakage factor, a real number between 0 and 1

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use CMA Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from CMA algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,cma(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'Constant Modulus'
  nWeights: 5
  nSampPerSym: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [1 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','CMA','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'CMA'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputSamplesPerSymbol: 1
  AdaptWeightsSource: 'Property'
  AdaptWeights: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Algorithms

Referring to the schematics in “Equalization”, define *w* as the vector of all weights w_i and define *u* as the vector of all inputs u_i. Based on the current set of weights, *w*, this adaptive algorithm creates the new set of weights given by

(`LeakageFactor`) w + (`StepSize`) u^*e

where the * operator denotes the complex conjugate.

Compatibility Considerations

`cma` will be removed
Warns starting in R2020a

- cma will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to CMA.
- The `comm.LinearEqualizer` or `comm.DecisionFeedback` System objects do not have a leakage factor property. This is equivalent to setting `LeakageFactor` to `1` in the cma function.
- For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-159.

References

[1] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[2] Johnson, Richard C., Jr., Philip Schniter, Thomas. J. Endres, et al., “Blind Equalization Using the Constant Modulus Criterion: A Review,” *Proceedings of the IEEE*, Vol. 86, October 1998, pp. 1927–1950.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

comm_links

Library link information for Communications Toolbox blocks

Syntax

```
comm_links
comm_links(sys)
comm_links(sys,color)
```

Description

`comm_links` returns a structure with two elements. Each element contains a cell array of strings containing names of library blocks in the current system. The blocks are grouped into two categories: obsolete and current. Blocks at all levels of the model are analyzed.

`comm_links(sys)` works as above on the named system `sys`, instead of the current system.

`comm_links(sys,`*`color`*`)` additionally colors all obsolete blocks according to the specified *`color`*. *`color`* is one of the following strings: 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', or 'black'.

Obsolete blocks are blocks that are no longer supported. They might or might not work properly.

Current blocks are supported and represent the latest block functionality.

See Also

`liblinks`

Introduced before R2006a

commlib

Open main Communications Toolbox block library

Syntax

```
commlib
```

Description

`commlib` opens the latest version of the Communications Toolbox block library.

See Also

`dsplib`

Introduced before R2006a

commscope

(To be removed) Package of communications scope classes

Note `commscope.eyediagram` will be removed in a future release. Use `comm.ConstellationDiagram` and `eyediagram` instead.

Syntax

```
h = commscope.<type>(...)
```

Description

`h = commscope.<type>(...)` returns a communications scope object `h` of type `type`.

Type `help commscope` to get a complete list of available types.

Each `type` of communications scope object is equipped with functions for simulation and visualization. Type `help commscope.<type>` to get the complete help on a specific communications scope object, for example `help commscope.eyediagram`.

Compatibility Considerations

`commscope` will be removed
Warns starting in R2017b

`commscope.eyediagram` will be removed in a future release. Use `comm.ConstellationDiagram` and `eyediagram` instead.

See Also

Objects
`comm.ConstellationDiagram`

Functions
`eyediagram` | `scatterplot`

Introduced in R2007b

commscope.eyediagram

(Removed) Eye diagram analysis

Note `commscope.eyediagram` has been removed. Use `eyediagram` instead.

Syntax

```
h = commscope.eyediagram
h = commscope.eyediagram(property1,value1,...)
```

Description

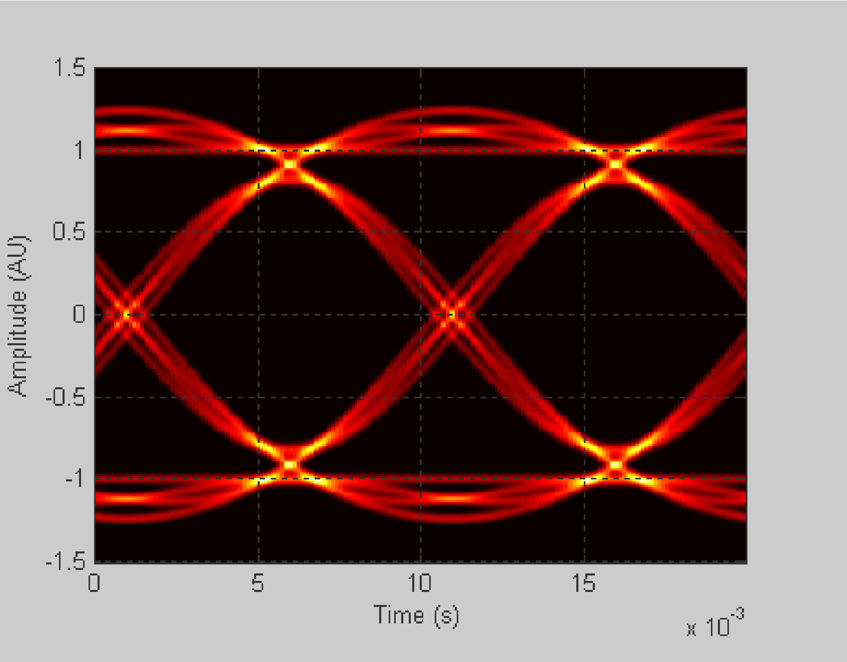
`h = commscope.eyediagram` constructs an eye diagram object, `h`, with default properties. This syntax is equivalent to:

```
H = commscope.eyediagram('SamplingFrequency', 10000, ...
                         'SamplesPerSymbol', 100, ...
                         'SymbolsPerTrace', 2, ...
                         'MinimumAmplitude', -1, ...
                         'MaximumAmplitude', 1, ...
                         'AmplitudeResolution', 0.0100, ...
                         'MeasurementDelay', 0, ...
                         'PlotType', '2D Color', ...
                         'PlotTimeOffset', 0, ...
                         'PlotPDFRange', [0 1], ...
                         'ColorScale', 'linear', ...
                         'RefreshPlot', 'on');
```

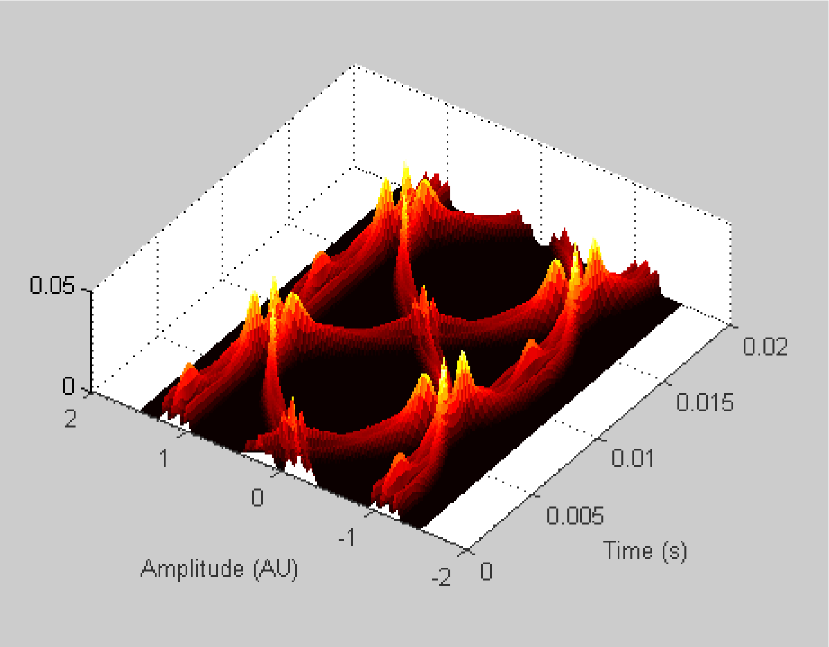
`h = commscope.eyediagram(property1,value1,...)` constructs an eye diagram object, `h`, with properties as specified by property/value pairs.

The eye diagram object creates a series of vertical histograms from zero to *T* seconds, at T_s second intervals, where *T* is a multiple of the symbol duration of the input signal and T_s is the sampling time. A vertical histogram is defined as the histogram of the amplitude of the input signal at a given time. The histogram information is used to obtain an approximation to the probability density function (PDF) of the input amplitude distribution. The histogram data is used to generate `'2D Color'` plots, where the color indicates the value of the PDF, and `'3D Color'` plots. The `'2D Line'` plot is obtained by constructing an eye diagram from the last *n* traces stored in the object, where a trace is defined as the segment of the input signal for a *T* second interval.

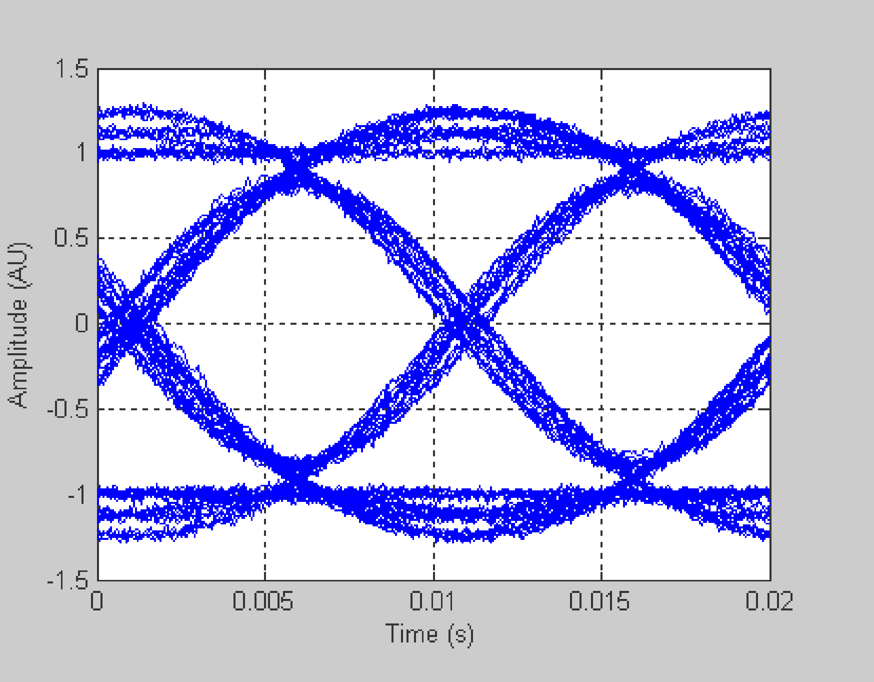
You can change the plot type by setting the `PlotType` property. The following plots are examples of each type.



2D-Color Eye Diagram



3D-Color Eye Diagram



2D-Line Eye Diagram

To see a detailed demonstration of this object's use, type `showdemo scattereyedemo;` at the command line.

Properties

An eye diagram scope object has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
`Type`	Type of scope object (`'Eye Diagram'`). This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz.
`SamplesPerSymbol`	Number of samples used to represent a symbol. An increase in `SamplesPerSymbol` improves the resolution of an eye diagram.
`SymbolRate`	The symbol rate of the input signal. This property is not writable and is automatically computed based on `SamplingFrequency` and `SamplesPerSymbol`.
`SymbolsPerTrace`	The number of symbols spanned on the time axis of the eye diagram scope.
`MinimumAmplitude`	Minimum amplitude of the input signal. Signal values less than this value are ignored both for plotting and for measurement computation.

Property	Description
MaximumAmplitude	Maximum amplitude of the input signal. Signal values greater than this value are ignored both for plotting and for measurement computation.
AmplitudeResolution	The resolution of the amplitude axis. The amplitude axis is created from MinimumAmplitude to MaximumAmplitude with AmplitudeResolution steps.
MeasurementDelay	The time in seconds the scope waits before starting to collect data.
PlotType	Type of the eye diagram plot. The choices are '2D Color' (two dimensional eye diagram, where color intensity represents the probability density function values), '3D Color' (three dimensional eye diagram, where the z-axis represents the probability density function values), and '2D Line' (two dimensional eye diagram, where each trace is represented by a line).
NumberOfStoredTraces	The number of traces stored to display the eye diagram in '2D Line' mode.
PlotTimeOffset	The plot time offset input values must reside in the closed interval [-Tsym Tsym], where Tsym is the symbol duration. Since the eye diagram is periodic, if the value you enter is out of range, it wraps to a position on the eye diagram that is within range.
RefreshPlot	The switch that controls the plot refresh style. The choices are 'on' (the eye diagram plot is refreshed every time the update method is called) and 'off' (the eye diagram plot is not refreshed when the update method is called).
PlotPDFRange	The range of the PDF values that will be displayed in the '2D Color' mode. The PDF values outside the range are set to a constant mask color.
ColorScale	The scale used to represent the color, the z-axis, or both. The choices are 'linear' (linear scale) and 'log' (base ten logarithmic scale).
SamplesProcessed	The number of samples processed by the eye diagram object. This value does not include the discarded samples during the MeasurementDelay period. This property is not writable.
OperationMode	When the operation mode is complex signal, the eye diagram collects and plots data on both the in-phase component and the quadrature component. When the operation mode is real signal, the eye diagram collects and plots real signal data.
Measurements	An eye diagram can display various types of measurements. All measurements are done on both the in-phase and quadrature signal, unless otherwise stated. For more information, see the Measurements section.

The resolution of the eye diagram in '2D Color' and '3D Color' modes can be increased by increasing SamplingFrequency, decreasing AmplitudeResolution, or both.

Changing MinimumAmplitude, MaximumAmplitude, AmplitudeResolution, SamplesPerSymbol, SymbolsPerTrace, and MeasurementDelay resets the measurements and updates the eye diagram.

Methods

An eye diagram object is equipped with seven methods for inspection, object management, and visualization.

update

This method updates the eye diagram object data.

update(h,x) updates the collected data of the eye diagram object h with the input x.

If the RefreshPlot property is set to 'on', the update method also refreshes the eye diagram figure.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off')

% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan,0.5*sin(2*pi*(0:1/100:10))+j*cos(2*pi*(0:1/100:10)));
% update the eyediagram
update(h, x);
% Check the number of processed samples
h.SamplesProcessed
```

plot

This method displays the eye diagram figure.

The plot method has three usage cases:

plot(h) plots the eye diagram for the eye diagram object h with the current colormap or the default linespec.

plot(h,cmap), when used with the plottype set to '2D Color' or '3D Color', plots the eye diagram for the object h, and sets the colormap to cmap.

plot(h,linespec), when used with the plottype set to '2D Line', plots the eye diagram for the object h using linespec as the line specification. See the help for plot for valid linespecs.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
% Prepare a noisy sinusoid as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10)));
% Update the eye diagram
update(h, x);
% Display the eye diagram figure
plot(h)

% Display the eye diagram figure with jet colormap
plot(h, jet(64))

% Display 2D Line eye diagram with red dashed lines
h.PlotType = '2D Line';
plot(h, 'r--')
```

exportdata

This method exports the eye diagram data.

`[VERHIST EYEL HORHISTX HORHISTRF] = EXPORTDATA(H)` Exports the eye diagram data collected by the eyediagram object *H*.

`VERHIST` is a matrix that holds the vertical histogram, which is also used to plot '2D Color' and '3D Color' eye diagrams.

`EYEL` is a matrix that holds the data used to plot 2D Line eye diagram. Each row of the EYEC holds one trace of the input signal.

`HORHISTX` is a matrix that holds the crossing point histogram data collected for the values defined by the CrossingAmplitudes property of the MeasurementSetup object. HORHISTX(i, :) represents the histogram for CrossingAmplitudes(i).

`HORHISTRF` is a matrix that holds the crossing point histograms for rise and fall time levels. HORHISTRF(i,:) represents the histogram for AmplitudeThreshold(i).

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off');
% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10)));
% Update the eyediagram
update(h, x);
% Export the data
[eyec eyel horhistx horhistrf] = exportdata(h);
% Plot line data
t=0:1/h.SamplingFrequency:h.SymbolsPerTrace/h.SymbolRate;
plot(t, real(eyel)); xlabel('time (s)');...
          ylabel('Amplitude (AU)'); grid on;
% Plot 2D Color data
t=0:1/h.SamplingFrequency:h.SymbolsPerTrace/h.SymbolRate;
a=h.MinimumAmplitude:h.AmplitudeResolution:h.MaximumAmplitude;
imagesc(t,a,eyec); xlabel('time (s)'); ylabel('Amplitude (AU)');
```

reset

This method resets the eye diagram object.

reset(h) resets the eye diagram object h. Resetting h clears all the collected data.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('RefreshPlot', 'off');
% Prepare a noisy sinusoidal as input
hChan = comm.AWGNChannel('NoiseMethod', 'Signal to noise ratio (SNR)',...
  'SNR', 20);
x = step(hChan, 0.5*sin(2*pi*(0:1/100:10))+ j*0.5*cos(2*pi*(0:1/100:10)));
update(h, x);             % update the eyediagram
h.SamplesProcessed        % Check the number of processed samples
reset(h);                 % reset the object
h.SamplesProcessed        % Check the number of processed samples
```

copy

This method copies the eye diagram object.

h = copy(ref_obj) creates a new eye diagram object h and copies the properties of object h from properties of ref_obj.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram('MinimumAmplitude', -3, ...
    'MaximumAmplitude', 3);
disp(h); % display object properties
h1 = copy(h)
```

disp

This method displays properties of the eye diagram object.

disp(h) displays relevant properties of eye diagram object h.

If a property is not relevant to the object's configuration, it is not displayed. For example, for a commscope.eyediagram object, the ColorScale property is not relevant when PlotType property is set to '2D Line'. In this case the ColorScale property is not displayed.

The following is an example of its use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
% Display object properties
disp(h);
h = commscope.eyediagram('PlotType', '2D Line')
```

close

This method closes the eye diagram object figure.

close(h) closes the figure of the eye diagram object h.

The following example shows this method's use:

```
% Create an eye diagram scope object
h = commscope.eyediagram;
```

```
% Call the plot method to display the scope
plot(h);
% Wait for 1 seconds
pause(1)
% Close the scope
close(h)
```

analyze

This methods executes eye diagram measurements. analyze(h) executes the eye diagram measurements on the collected data of the eye diagram scope object *h*. The results of the measurements are stored in the Measurements property of *h*. For more information, see “Eye Diagram Analysis”.

In some cases, the analyze method cannot determine a measurement value. If this problem occurs, verify that your settings for measurement setup values or the eye diagram are valid.

Measurement Setup Parameters

A number of set-up parameters control eye diagram measurements. This section describes these set-up parameters and the measurements they affect. For more information, see “Eye Diagram Analysis”.

Eye Level Boundaries

Eye Level Boundaries are defined as a percentage of the symbol duration. The analyze method calculates the eye levels by averaging the vertical histogram within a given time interval defined by the eye level boundaries. A common value you can use for NRZ signals is 40% to 60%. For RZ signals, a narrower band of 5% is more appropriate. The default setting for *Eye level Boundaries* is a 2-by-1 vector where the first element is the lower boundary and the second element is the upper boundary. When the eye level boundary changes, the object recalculates this value.

Reference Amplitude

Reference Amplitude is the boundary value at which point the signal crosses from one signal level to another. Reference amplitude represents the decision boundary of the modulation scheme. This value is used to perform jitter measurements. The default setting for *Reference Amplitude* is a 2-by-1 double vector where the first element is the lower boundary and the second element is the upper boundary. Setting the reference amplitude resets the eye diagram.

The crossing instants of the input signal are detected and recorded as crossing times. A common value you can use for NRZ signals is `0`. For RZ signals, you can use the mean value of `1` and `0` levels. Reference amplitude is stored in a 2-by-N matrix, where the first row is the in-phase values and second row is the quadrature values. See Eye Crossing Time for more information.

Crossing Bandwidth

Crossing Bandwidth is the amplitude band used to measure the crossing times of the eye diagram. *Crossing Bandwidth* represents a percentage of the amplitude span of the eye diagram, typically 5%. See Eye Crossing Time for more information. The default setting for *Crossing Bandwidth* is 0.0500.

Bit Error Rate Threshold

The eye opening measurements, random, and total jitter measurements are performed at a given BER value. This BER value defines the BER threshold. A typical value is 1e^{-12}. The default setting for *Bit*

Error Threshold is 1.0000e^{-12}. When the bit error rate threshold changes, the object recalculates this value.

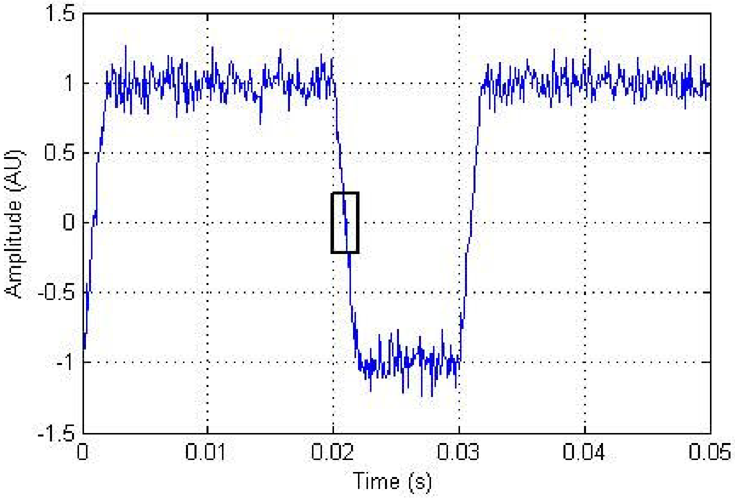
Amplitude Threshold

The rise time of the signal is defined as the time required for the signal to travel from the lower amplitude threshold to the upper amplitude threshold. The fall time, measured from the upper amplitude threshold to the lower amplitude threshold, is defined as a percentage of the eye amplitude. The default setting is 10% for the lower threshold and 90% for the upper threshold. Setting the amplitude threshold resets the eye diagram. See Eye Rise Time and Eye Fall Time for more information.

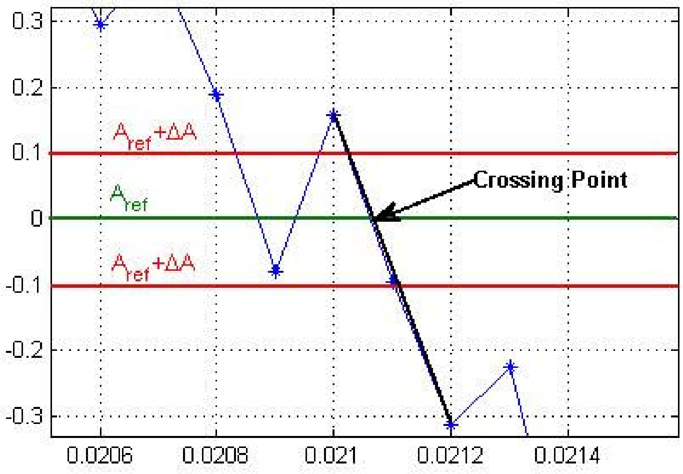
Jitter Hysteresis

You can use the *JitterHysteresis* property of the eye measurement setup object to remove the effect of noise from the horizontal histogram estimation. The default value for *Jitter Hysteresis* is zero. Setting the jitter hysteresis value resets the eye diagram.

If channel noise impairs the signal being tested, as shown in the following figure, the signal may seem like it crosses the reference amplitude level multiple times during a single 0-1 or 1-0 transition.



See the zoomed—in image for more detail.



To eliminate the effect of noise, define a hysteresis region between two threshold values: A_{ref} + ΔA and A_{ref} - ΔA, where A_{ref} is the reference amplitude value and ΔA is the jitter hysteresis value. If the signal crosses both threshold values, level crossing is declared. Then, linear interpolation calculates the crossing point in the horizontal histogram estimation.

Compatibility Considerations

`commscope.eyediagram` has been removed
Errors starting in R2020a

`commscope.eyediagram` has been removed. Use `eyediagram` instead.

References

[1] Nelson Ou, et al, *Models for the Design and Test of Gbps-Speed Serial Interconnects,*IEEE Design & Test of Computers, pp. 302-313, July-August 2004.

[2] HP E4543A Q Factor and Eye Contours Application Software, Operating Manual, http://agilent.com

[3] Agilent 71501D Eye-Diagram Analysis, User’s Guide, http://www.agilent.com

[4] 4] Guy Foster, *Measurement Brief: Examining Sampling Scope Jitter Histograms,* White Paper, SyntheSys Research, Inc., July 2005.

[5] *Jitter Analysis: The dual-Dirac Model, RJ/DJ, and Q-Scale,* White Paper, Agilent Technologies, December 2004, http://www.agilent.com

See Also

`eyediagram`

Topics
“Eye Diagram Analysis”

Introduced in R2007b

commsrc.combinedjitter

Construct combined jitter generator object

Syntax

```
combJitt = commsrc.combinedjitter
combJitt = commsrc.combinedjitter(Name,Value)
```

Description

`combJitt = commsrc.combinedjitter` constructs a default combined jitter generator object, `combJitt`, with all jitter components disabled.

Use the object to generate jitter samples that include any combination of random, periodic, and Dirac components.

`combJitt = commsrc.combinedjitter(Name,Value)` creates a combined jitter generator object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

A combined jitter generator object includes these properties. You can edit all properties, except those explicitly noted.

Property	Description
`Type`	Type of object, `Combined Jitter Generator`. This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz. Default value 1e4.
`RandomJitter`	Variable to enable the random jitter generator. Specify as either `'off'` (default) or `'on'`.
`RandomStd`	Standard deviation of the random jitter generator in seconds. Applies when `RandomJitter` is `'on'`. Default value 1e-4.
`PeriodicJitter`	Variable to enable the periodic jitter generator. Specify as either `'off'` (default) or `'on'`.
`PeriodicNumber`	Number of sinusoidal components. The `PeriodicNumber` must be a finite positive scalar integer. Applies when `PeriodicJitter` is `'on'`. Default value 1.
`PeriodicAmplitude`	Amplitude of each sinusoidal component of the periodic jitter in seconds. Applies when `PeriodicJitter` is `'on'`. Default value 5e-4.

Property	Description
PeriodicFrequencyHz	Frequency of each sinusoidal component of the periodic jitter measured in Hz. Applies when PeriodicJitter is 'on'. Default value is 200.
PeriodicPhase	Phase of each sinusoidal component of the periodic jitter in radians. Applies when PeriodicJitter is 'on'. Default value 0.
DiracJitter	Variable to enable the Dirac jitter generator. Specify as either 'off' (default) or 'on'.
DiracNumber	Number of Dirac components. The DiracNumber must be a finite positive scalar integer. Applies when DiracJitter is 'on'. Default value 2.
DiracDelta	Time delay of each Dirac component in seconds. Applies when DiracJitter is 'on'. Default value [-5.e-4 5.e-4].
DiracProbability	Probability of each Dirac component represented as a vector of length DiracNumber. The sum of the probabilities must equal one. Applies when DiracJitter is 'on'. Default value [0.5 0.5].

Object Functions

A combined jitter generator object has three object functions, as described in this section.

generate

This object function generates jitter samples based on the jitter generator object. It has one input argument, which is the number of samples in a frame. Its output is a single-column vector of length N. You can call this object function using this syntax:

```
x = generate(combJitt,N)
```

where combJitt is the generator object, N is the number of output samples, and x is a real single-column vector.

reset

This object function resets the internal states of the combined jitter generator. You can call this object function using this syntax:

```
reset(combJitt)
```

where combJitt is the generator object.

disp

Display the properties of the combined generator object, combJitt. You can call this object function using this syntax:

```
disp(combJitt)
```

where combJitt is the generator object.

Examples

Generate Combined Random and Periodic Jitter

Generate 500 jitter samples composed of random and periodic components.

Create a commsrc.combinedjitter object configured to apply a combination of random and periodic jitter components. Use name-value pairs to enable RandomJitter and PeriodicJitter, and to assign jitter settings. Set the standard deviation of the random jitter to 2e-4 seconds, the periodic jitter amplitude to 5e-4 seconds, and the periodic jitter frequency to 2 Hz.

```
numSamples = 500;
combJitt = commsrc.combinedjitter(...
    'RandomJitter','on', ...
    'RandomStd',2e-4, ...
    'PeriodicJitter','on', ...
    'PeriodicAmplitude',5e-4, ...
    'PeriodicFrequencyHz',200)
```

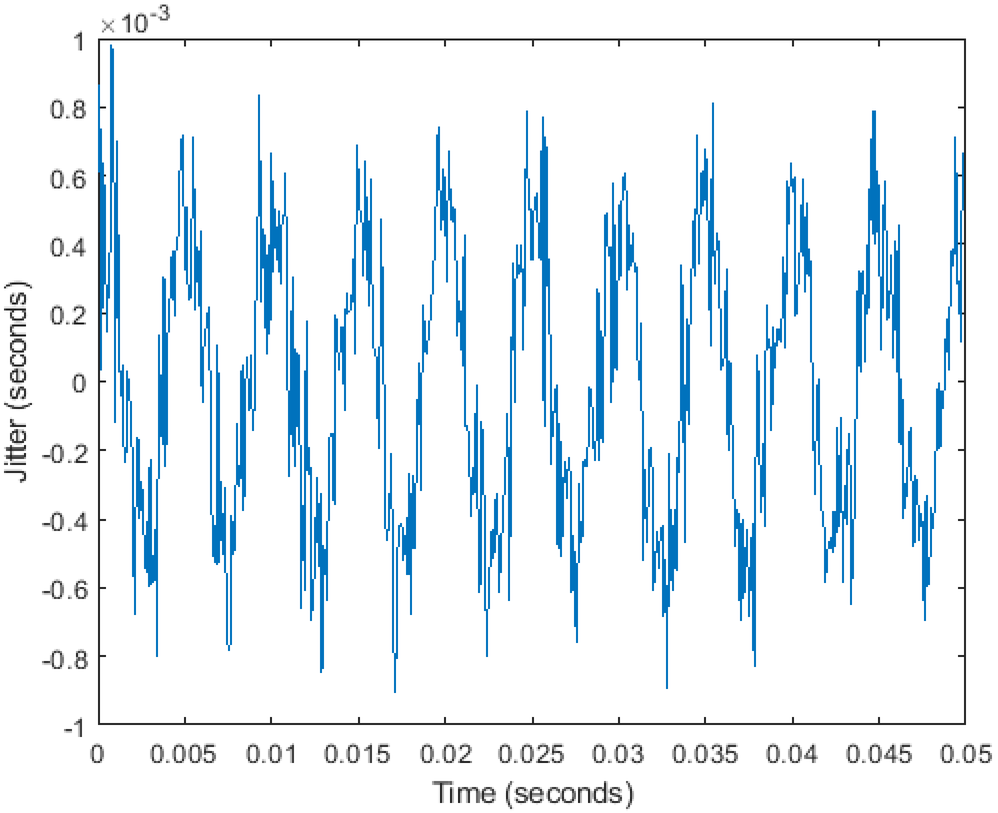
```
combJitt =
                   Type: 'Combined Jitter Generator'
      SamplingFrequency: 10000
           RandomJitter: 'on'
              RandomStd: 2.0000e-04
         PeriodicJitter: 'on'
         PeriodicNumber: 1
      PeriodicAmplitude: 5.0000e-04
    PeriodicFrequencyHz: 200
          PeriodicPhase: 0
            DiracJitter: 'off'
```

Use the generate method to create the combined jitter samples.

```
y = generate(combJitt,numSamples);
x = [0:numSamples-1];
```

Plot the jitter samples. You can see the Gaussian and periodic nature of the combined jitter.

```
plot(x/combJitt.SamplingFrequency,y)
xlabel('Time (seconds)')
ylabel('Jitter (seconds)')
```



Display commsrc.combinedjitter Object Settings

Create a `commsrc.combinedjitter` object. Display the default object property values.

```
combJitt = commsrc.combinedjitter;
disp(combJitt)
```

```
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Create a `commsrc.combinedjitter` object with random, periodic, and Dirac jitters enabled. Display the object property values.

```
combJitt = commsrc.combinedjitter('RandomJitter','on', ...
    'PeriodicJitter','on','DiracJitter','on');
disp(combJitt)
```

```
                   Type: 'Combined Jitter Generator'
      SamplingFrequency: 10000
           RandomJitter: 'on'
              RandomStd: 1.0000e-04
         PeriodicJitter: 'on'
         PeriodicNumber: 1
      PeriodicAmplitude: 5.0000e-04
    PeriodicFrequencyHz: 200
          PeriodicPhase: 0
            DiracJitter: 'on'
            DiracNumber: 2
             DiracDelta: [-5.0000e-04 5.0000e-04]
       DiracProbability: [0.5000 0.5000]
```

Generate Non-return-to-zero Pattern Signal

Generate a binary non-return-to-zero (NRZ) signal utilizing the pattern generator object. View the NRZ signal with and without jitter applied to the signal.

Initialize system parameters.

```
Fs = 10000;            % Sample rate
Rs = 50;               % Symbol rate (Sps)
sps = Fs/Rs;           % Number of samples per symbol
Trise = 1/(5*Rs);      % Rise time of the NRZ signal
Tfall = 1/(5*Rs);      % Fall time of the NRZ signal
frameLen = 100;        % Number of symbols in a frame
spt = 200;               % Number of samples per trace on eye diagram
```

Create a pattern generator object with no jitter component assigned.

```
src = commsrc.pattern('SamplingFrequency', Fs, ...
                      'SamplesPerSymbol', sps, ...
                      'RiseTime', Trise, ...
                      'FallTime', Tfall)
```

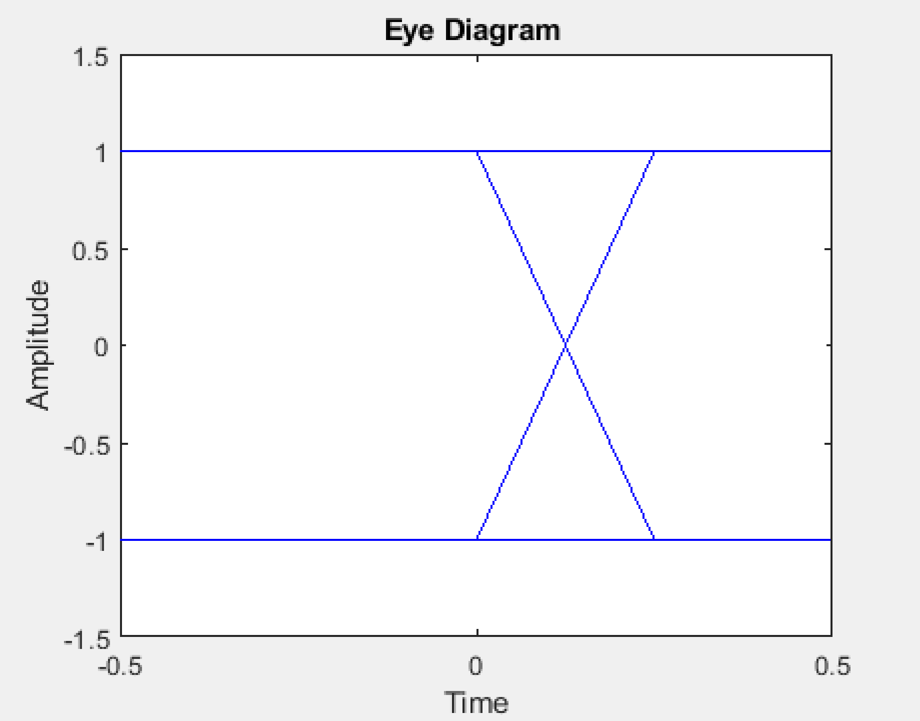
```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

```
src.Jitter
```

```
ans =
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Generate an NRZ signal and view the eye diagram of the signal.

```
message = generate(src,frameLen);
eyediagram(message,spt)
```



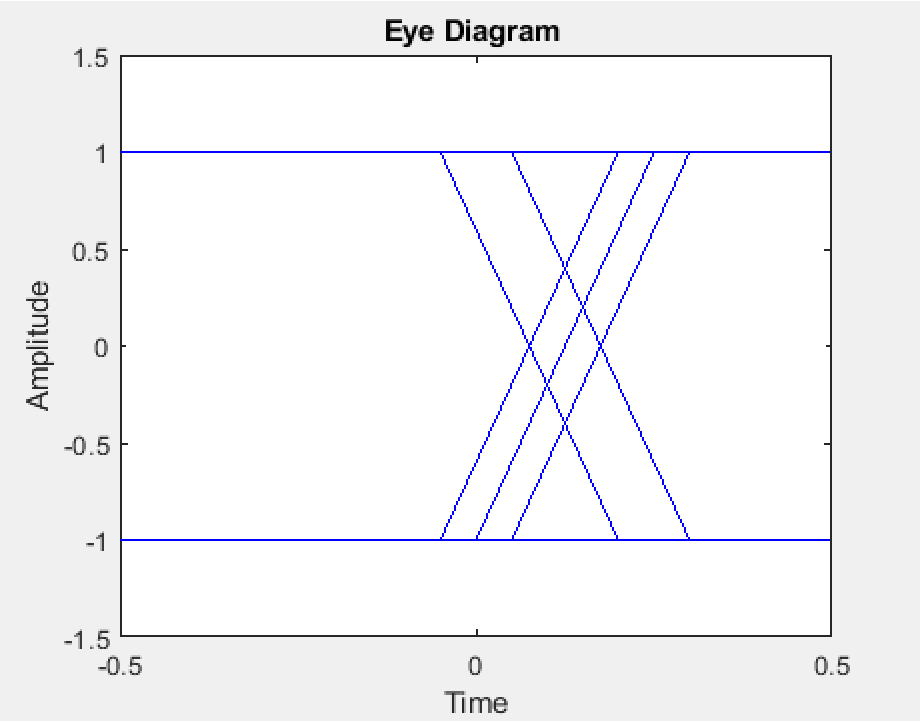
Add inter-symbol-interference (ISI) to an NRZ signal. ISI is modeled by two equal amplitude Dirac functions. Create a combined jitter object with Dirac jitter and assign it to the pattern generator object.

```
jitterSrc = commsrc.combinedjitter('DiracJitter','on', ...
    'DiracDelta',0.05/Rs*[-1 1]);
src.Jitter = jitterSrc
```

```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

Generate an NRZ signal that has jitter added to it and view the eye diagram of the signal.

```
reset(src);
message = generate(src, frameLen);
eyediagram(message,spt)
```



See Also

Functions
`commsrc.pattern` | `eyediagram`

Topics
"Eye Diagram Analysis"

Introduced in R2015a

commsrc.pattern

Construct pattern generator object

Syntax

```
h = commsrc.pattern
h = commsrc.pattern(Name,Value)
```

Description

`h = commsrc.pattern` constructs a pattern generator object, `h`.

The pattern generator object produces modulated data patterns. The object can be used to inject jitter into modulated signals.

`h = commsrc.pattern(Name,Value)` creates a combined jitter generator object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

A pattern generator object includes these properties. You can edit all properties, except those explicitly noted.

Property	Description
`Type`	Type of pattern generator object (`'Pattern Generator'`). This property is not writable.
`SamplingFrequency`	Sampling frequency of the input signal in hertz.
`SymbolRate`	The symbol rate of the input signal. This property depends upon the `SamplingFequency` and `SamplesPerSymbol` properties. This property is not writable.
`SamplesPerSymbol`	The number of samples representing a symbol. `SamplesPerSymbol` must be an integer. This property affects `SymbolRate`.
`PulseType`	The type of pulse the object generates. Pulse types available: return-to-zero (`'RZ'`) and non-return-to-zero (`'NRZ'`). The initial condition for an `'NRZ'` pulse is `0`.
`OutputLevels`	Amplitude levels that correspond to the symbol indices. For an `'NRZ'` pulse, specify as a 1-by-2 vector. The first element of the 1-by-2 vector corresponds to the 0th symbol (data bit value 0). The second element corresponds to the 1st symbol (data bit value 1). For an `'RZ'` pulse, specify as a scalar and the value corresponds to the data bit value `1`.

Property	Description
`DutyCycle`	The duty cycle of the pulse the object generates. Displays calculated duty cycle based on pulse parameters. This property is not writable.
`RiseTime`	Specifies 10% to 90% rise time of the pulse in seconds.
`PulseDuration`	Pulse duration in seconds defined by IEEE STD 181 standard. See the Return-to-Zero (RZ) Signal Conversion: Ideal Pulse to STD–181 figure in the “Object Functions” on page 2-184. Applies when `PulseType` is `'RZ'`.
`FallTime`	Fall time of the pulse in seconds, specified as a percentage from 10 to 90.
`DataPattern`	The bit sequence the object uses, specified as `'PRBS5'`, `'PRBS6'`, ..., `'PRBS15'`, `'PRBS23'`, `'PRBS31'`, and`'User Defined'`.
`UserDataPattern`	User-defined bit pattern consisting of a vector of ones and zeroes. Applies when `DataPattern` is `'User Defined'`.
`Jitter`	Jitter characteristics, specified as a `commsrc.combinedjitter` object. Use this property to configure Random, Periodic and Dual Dirac Jitter.

Object Functions

A pattern generator object has five object functions, as described in this section.

generate

This object function outputs a frame worth of modulated and interpolated symbols. It has one input argument, which is the number of symbols in a frame. Its output is a column vector. You can call the object function using this syntax:

```
x = generate(h, N)
```

where `h` is the handle to the object, `N` is the number of output symbols, and `x` is a column vector whose length is `N` multiplied by `h.SamplesPerSymbol`.

reset

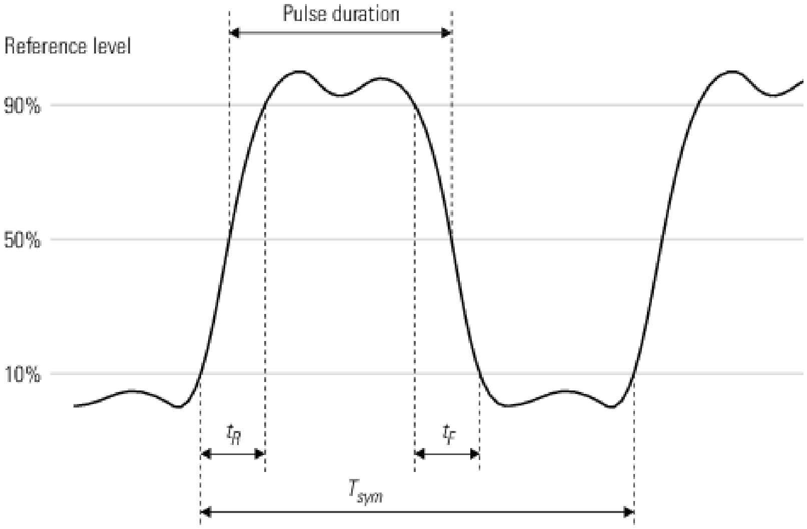
This object function resets the pattern generator to its default state. The property values do not reset unless they relate to the state of the object. This object function has no input arguments.

idealtostd181

This object function converts the ideal pulse specifications to IEEE STD-181 specifications. The ideal 0% to 100% span rise time (`tr`) and fall time (`tf`) are converted to 10% to 90% spans with a 50% pulse width duration (`pw`). Call the `idealtostd181` object function using this syntax:

```
h = idealtostd181(tr,tf,pw)
```

The object function sets the appropriate properties. The IEEE STD-181 Return-to-Zero (RZ) signal parameters are shown in this figure.

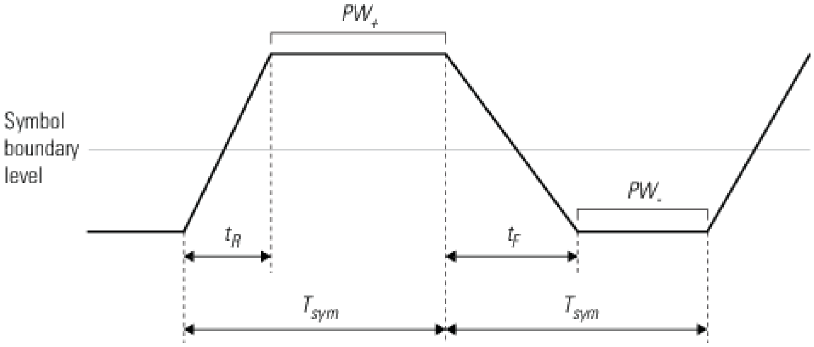


std181toideal

The `std181toideal` object function converts the IEEE STD-181 pulse specifications, stored in the pattern generator, to ideal pulse specifications. The function converts the rise and fall times from 10% – 90% span to 0% – 100% span, and converts the 50% pulse duration to pulse width. Call the `std181toideal` object function using this syntax:

```
[tr tf pw] = std181toideal(h)
```

where `h` is the pattern generator object handle, `tr` is the ideal 0% – 100% rise time, `tf` is the ideal 0% – 100% fall time, and `pw` is the ideal pulse width. The ideal pulse non-return-to-zero (NRZ) signal parameters are shown in this figure.



Use the property values for IEEE STD-181 specifications.

computedcd

The `computedcd` object function computes the duty cycle distortion, *DCD*, of the pulse defined by the pattern generator object *h*.

DCD represents the ratio of the pulse on duration to the pulse off duration. For an NRZ pulse, on duration is the duration the pulse spends above the symbol boundary level. Off duration is the duration the pulse spends below zero. Call the `computedcd` object function using this syntax:

```
dcd = computedcd(h)
```

The software calculates *DCD* given t_R, t_F, T_{sym}. This formula assumes that the symbol boundary level is zero.

$$T_h = (A_h\text{-}A_l) * \frac{t_R}{A_l} + (A_h\text{-}A_l) * \frac{t_F}{A_l} + PW_+$$

$$T_l = (A_h\text{-}A_l) * \frac{t_R}{A_l} + (A_h\text{-}A_l) * \frac{t_F}{A_l} + PW_-$$

$$DCD = \frac{T_h}{T_l}$$

Where T_h is the duration of the high signal, T_l is the duration of the low signal, and *DCD* represents the ratio of the duration of the high signal to the low signal.

Examples

Display commsrc.pattern Object Settings

Create a `commsrc.pattern` object. Display the default object property values.

```
h = commsrc.pattern;
disp(h)
```

```
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 100
           SymbolRate: 100
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0
             FallTime: 0
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

Generate Non-return-to-zero Pattern Signal

Generate a binary non-return-to-zero (NRZ) signal utilizing the pattern generator object. View the NRZ signal with and without jitter applied to the signal.

Initialize system parameters.

```
Fs = 10000;            % Sample rate
Rs = 50;               % Symbol rate (Sps)
sps = Fs/Rs;           % Number of samples per symbol
Trise = 1/(5*Rs);      % Rise time of the NRZ signal
Tfall = 1/(5*Rs);      % Fall time of the NRZ signal
frameLen = 100;        % Number of symbols in a frame
spt = 200;               % Number of samples per trace on eye diagram
```

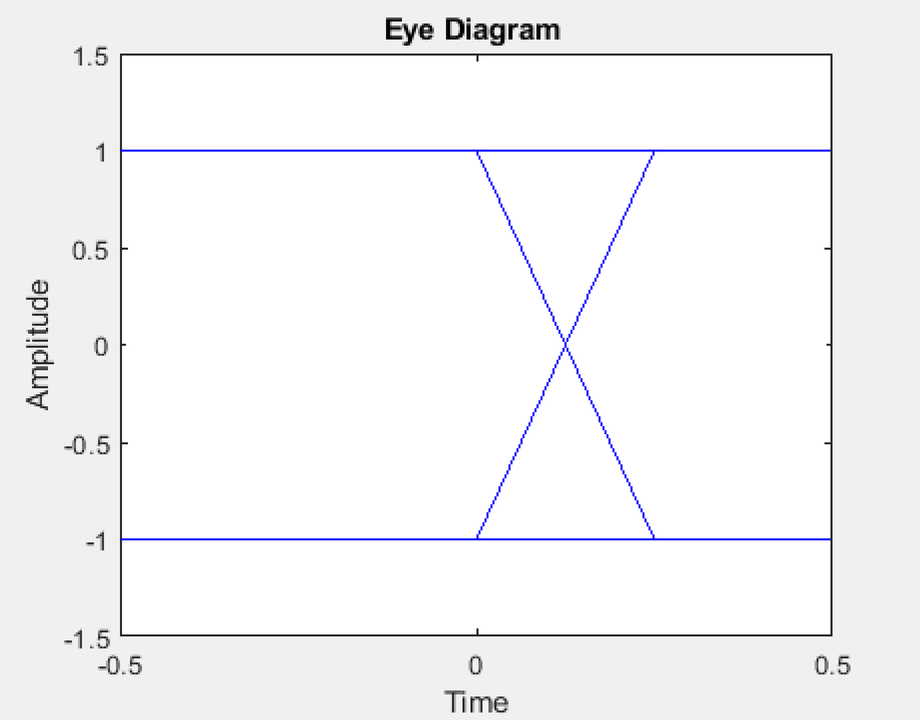
Create a pattern generator object with no jitter component assigned.

```
src = commsrc.pattern('SamplingFrequency', Fs, ...
                      'SamplesPerSymbol', sps, ...
                      'RiseTime', Trise, ...
                      'FallTime', Tfall)

src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

```
src.Jitter

ans =
                 Type: 'Combined Jitter Generator'
    SamplingFrequency: 10000
         RandomJitter: 'off'
       PeriodicJitter: 'off'
          DiracJitter: 'off'
```

Generate an NRZ signal and view the eye diagram of the signal.

```
message = generate(src,frameLen);
eyediagram(message,spt)
```



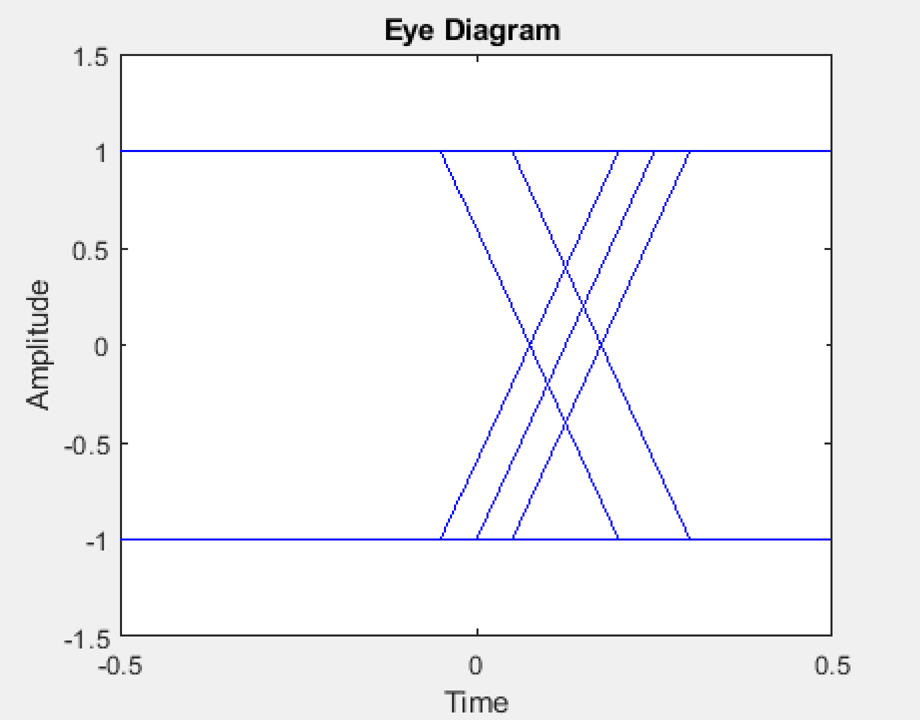
Add inter-symbol-interference (ISI) to an NRZ signal. ISI is modeled by two equal amplitude Dirac functions. Create a combined jitter object with Dirac jitter and assign it to the pattern generator object.

```
jitterSrc = commsrc.combinedjitter('DiracJitter','on', ...
    'DiracDelta',0.05/Rs*[-1 1]);
src.Jitter = jitterSrc
```

```
src =
                 Type: 'Pattern Generator'
    SamplingFrequency: 10000
     SamplesPerSymbol: 200
           SymbolRate: 50
            PulseType: 'NRZ'
         OutputLevels: [-1 1]
             RiseTime: 0.0040
             FallTime: 0.0040
          DataPattern: 'PRBS7'
               Jitter: [1x1 commsrc.combinedjitter]
```

Generate an NRZ signal that has jitter added to it and view the eye diagram of the signal.

```
reset(src);
message = generate(src, frameLen);
eyediagram(message,spt)
```



References

[1] IEEE Standard for Transitions, Pulses, and Related Waveforms, STD-181-2011. Piscataway, NJ. 6 September 2011.

See Also

Functions

`commsrc.combinedjitter` | `eyediagram`

Topics

"Eye Diagram Analysis"

Introduced in R2008b

commsrc.pn

(To be removed) Create PN sequence generator object

Compatibility

commsrc.pn will be removed in a future release. To generate a pseudo-noise (PN) sequence, use the `comm.PNSequence` System object instead. For more details on the recommended workflow, see "Compatibility Considerations" on page 2-195.

Syntax

```
h = commsrc.pn
h = commsrc.pn(property1,value1,...)
```

Description

`h = commsrc.pn` creates a default PN sequence generator object *h*, and is equivalent to the following:

```
H = commsrc.pn('GenPoly',       [1 0 0 0 0 1 1], ...
              'InitialStates', [0 0 0 0 0 1],   ...
              'CurrentStates', [0 0 0 0 0 1],   ...
              'Mask',          [0 0 0 0 0 1],   ...
              'NumBitsOut',    1)
```

or

```
H = commsrc.pn('GenPoly',       [1 0 0 0 0 1 1], ...
              'InitialStates', [0 0 0 0 0 1],   ...
              'CurrentStates', [0 0 0 0 0 1],   ...
              'Shift',         0,               ...
              'NumBitsOut',    1)
```

`h = commsrc.pn(property1,value1,...)` creates a PN sequence generator object, *h*, with properties you specify as property/value pairs.

Properties

A PN sequence generator has the properties shown on the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
GenPoly	Generator polynomial vector array of bits; must be descending order
InitialStates	Vector array (with length of the generator polynomial order) of initial shift register values (in bits)

Property	Description
CurrentStates	Vector array (with length of the generator polynomial order) of present shift register values (in bits)
NumBitsOut	Number of bits to output at each `generate` method invocation
`Mask` or `Shift`	A mask vector of binary 0 and 1 values is used to specify which shift register state bits are XORed to produce the resulting output bit value. Alternatively, a scalar shift value may be used to specify an equivalent shift (either a delay or advance) in the output sequence.

The `'GenPoly'` property values specify the shift register connections. Enter these values as either a binary vector or a vector of exponents of the nonzero terms of the generator polynomial in descending order of powers. For the binary vector representation, the first and last elements of the vector must be 1. For the descending-ordered polynomial representation, the last element of the vector must be 0. For more information and examples, see the LFSR SSRG Details section of this page.

Methods

A PN sequence generator is equipped with the following methods.

generate

Generate [NumBitsOut x 1] PN sequence generator values

reset

Set the `CurrentStates` values to the `InitialStates` values

getshift

Get the actual or equivalent `Shift` property value

getmask

Get the actual or equivalent `Mask` property value

copy

Make an independent copy of a `commsrc.pn` object

disp

Display PN sequence generator object properties

Side Effects of Setting Certain Properties

Setting the GenPoly Property

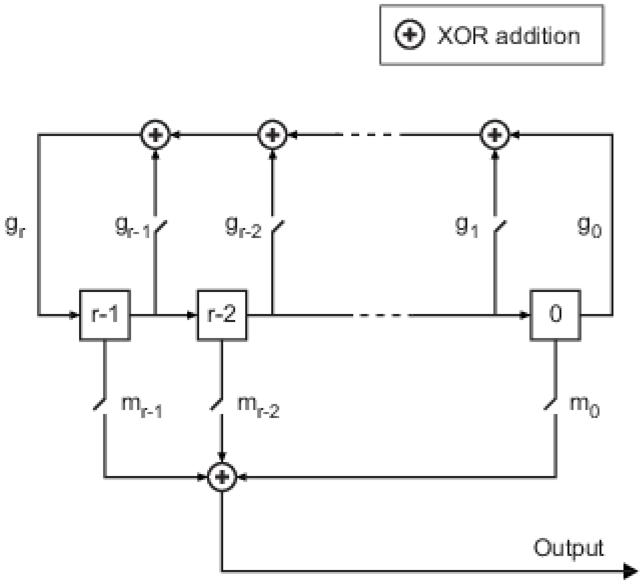
Every time this property is set, it will reset the entire object. In addition to changing the polynomial values, `'CurrentStates'`, `'InitialStates'`, and `'Mask'` will be set to their default values (`'NumBitsOut'` will remain the same), and no warnings will be issued.

Setting the InitialStates Property

Every time this property is set, it will also set `'CurrentStates'` to the new `'InitialStates'` setting.

LFSR SSRG Details

The `generate` method produces a pseudorandom noise (PN) sequence using a linear feedback shift register (LFSR). The LFSR is implemented using a simple shift register generator (SSRG, or Fibonacci) configuration, as shown below.



All *r* registers in the generator update their values at each time step according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The shift register is described by the `'GenPoly'` property (generator polynomial), which is a primitive binary polynomial in *z*, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + ... + g_0$. The coefficient g_k is 1 if there is a connection from the kth register, as labeled in the preceding diagram, to the adder. The leading term g_r and the constant term g_0 of the `'GenPoly'` property must be 1 because the polynomial must be primitive.

You can specify the **Generator polynomial** parameter using either of these formats:

- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, p(z) = $z^8 + z^2 + 1$.

The **Initial states** parameter is a vector specifying the initial values of the registers. The **Initial states** parameter must satisfy these criteria:

- All elements of the **Initial states** vector must be binary numbers.
- The length of the **Initial states** vector must equal the degree of the generator polynomial.

Note At least one element of the **Initial states** vector must be nonzero in order for the block to generate a nonzero sequence. That is, the initial state of at least one of the registers must be nonzero.

For example, the following table indicates two sets of parameter values that correspond to a generator polynomial of p(z) = $z^8 + z^2 + 1$.

Quantity	Example 1	Example 2
Generator polynomial	`g1 = [1 0 0 0 0 0 1 0 1]`	`g2 = [8 2 0]`
Degree of generator polynomial	8, which is `length(g1)-1`	8
Initial states	`[1 0 0 0 0 0 1 0]`	`[1 0 0 0 0 0 1 0]`

Output mask vector (or scalar shift value) shifts the starting point of the output sequence. With the default setting for this parameter, the only connection is along the arrow labeled m_0, which corresponds to a shift of 0. The parameter is described in greater detail below.

You can shift the starting point of the PN sequence with **Output mask vector (or scalar shift value)**. You can specify the parameter in either of two ways:

- An integer representing the length of the shift
- A binary vector, called the *mask vector*, whose length is equal to the degree of the generator polynomial

The difference between the block's output when you set **Output mask vector (or scalar shift value)** to 0, versus a positive integer d, is shown in the following table.

	T = 0	T = 1	T = 2	...	T = d	T = d+1
Shift = 0	x_0	x_1	x_2	...	x_d	x_{d+1}
Shift = d	x_d	x_{d+1}	x_{d+2}	...	x_{2d}	x_{2d+1}

Alternatively, you can set **Output mask vector (or scalar shift value)** to a binary vector, corresponding to a polynomial in *z*, $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + \ldots + m_1z + m_0$, of degree at most r-1. The mask vector corresponding to a shift of d is the vector that represents m(z) = z^d modulo g(*z*), where g(*z*) is the generator polynomial. For example, if the degree of the generator polynomial is 4, then the mask vector corresponding to d = 2 is `[0 1 0 0]`, which represents the polynomial m(z) = z^2. The preceding schematic diagram shows how **Output mask vector (or scalar shift value)** is implemented when you specify it as a mask vector. The default setting for **Output mask vector (or scalar shift value)** is `0`. You can calculate the mask vector using the Communications Toolbox function `shift2mask`.

Sequences of Maximum Length

If you want to generate a sequence of the maximum possible length for a fixed degree, *r*, of the generator polynomial, you can set **Generator polynomial** to a value from the following table. See Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995 for more information about the shift-register configurations that these polynomials represent.

r	Generator Polynomial	r	Generator Polynomial
2	[2 1 0]	21	[21 19 0]
3	[3 2 0]	22	[22 21 0]
4	[4 3 0]	23	[23 18 0]
5	[5 3 0]	24	[24 23 22 17 0]
6	[6 5 0]	25	[25 22 0]
7	[7 6 0]	26	[26 25 24 20 0]
8	[8 6 5 4 0]	27	[27 26 25 22 0]
9	[9 5 0]	28	[28 25 0]
10	[10 7 0]	29	[29 27 0]
11	[11 9 0]	30	[30 29 28 7 0]
12	[12 11 8 6 0]	31	[31 28 0]
13	[13 12 10 9 0]	32	[32 31 30 10 0]
14	[14 13 8 4 0]	33	[33 20 0]
15	[15 14 0]	34	[34 15 14 1 0]
16	[16 15 13 4 0]	35	[35 2 0]
17	[17 14 0]	36	[36 11 0]
18	[18 11 0]	37	[37 12 10 2 0]
19	[19 18 17 14 0]	38	[38 6 5 1 0]
20	[20 17 0]	39	[39 8 0]
40	[40 5 4 3 0]	47	[47 14 0]
41	[41 3 0]	48	[48 28 27 1 0]
42	[42 23 22 1 0]	49	[49 9 0]
43	[43 6 4 3 0]	50	[50 4 3 2 0]
44	[44 6 5 2 0]	51	[51 6 3 1 0]
45	[45 4 3 1 0]	52	[52 3 0]
46	[46 21 10 1 0]	53	[53 6 2 1 0]

Examples

Typically `commsrc.pn` is used to output pseudorandom data streams.

Construct a PN object.

```
h = commsrc.pn('Shift',0);
```

Output 10 PN bits.

```
set(h,'NumBitsOut',10);
generate(h)
```

```
ans = 10×1

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

Output 10 more PN bits.

```
generate(h)
```

```
ans = 10×1

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

Reset the object to the initial shift register state values.

```
reset(h);
```

Output 4 PN bits.

```
set(h,'NumBitsOut',4);
generate(h)
```

```
ans = 4×1

     1
     0
     0
     0
```

Compatibility Considerations

`commsrc.pn` will be removed in a future release.
Not recommended starting in R2020b

commsrc.pn will be removed in a future release. To generate a pseudo-noise (PN) sequence, use the `comm.PNSequence` System object instead.

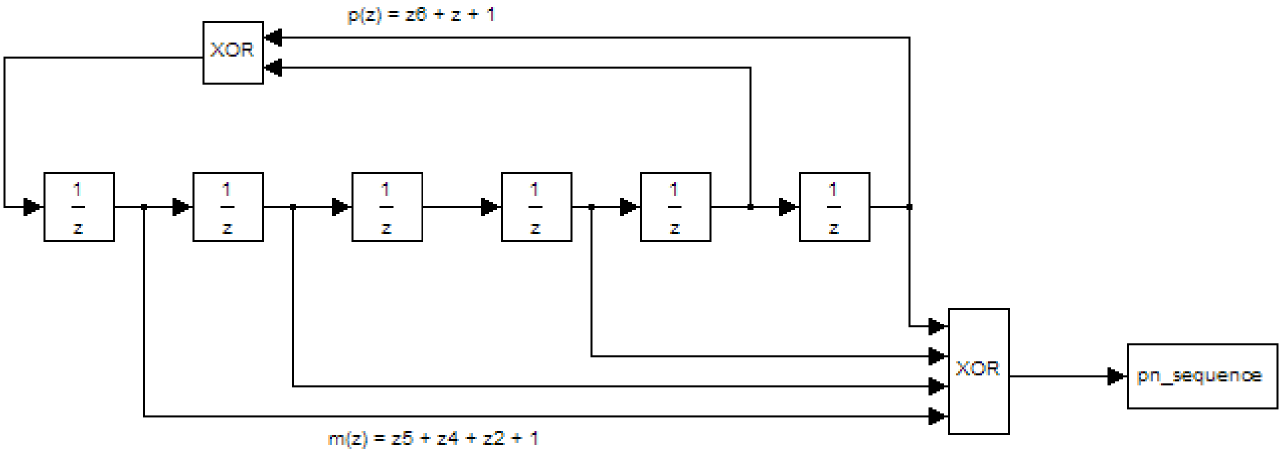
Replace instances of commsrc.pn with a `comm.PNSequence` System object. Note the mapping between commsrc.pn properties and `comm.PNSequence` properties:

commsrc.pn	**comm.PNSequence**
GenPoly	Polynomial
InitialStates	InitialConditions
Mask	Mask
NumBitsOut	SamplesPerFrame
CurrentStates	N/A

See the following table for an example of migrating the old workflow to the recommended workflow.

Set up a PN sequence generator. Define the polynomial in binary vector format or exponential vector format.

This figure defines a PN sequence generator with a generator polynomial p(z) = $z^6 + z + 1$.



See the following table for examples of migrating the old workflow to the recommended workflow.

Previous Workflow	**Recommended Workflow**

<table>
<tr>
<td>

Define this PN sequence generator as:

```
h1 = commsrc.pn('GenPoly', [1 0 0 0 0 1 1], 'Mask', [1 1 0 1 0 1]);
h2 = commsrc.pn('GenPoly', [1 0 0 0 0 1 1], 'Shift', 22);
mask2shift ([1 0 0 0 0 1 1],[1 1 0 1 0 1])
```

Alternatively, you can input `GenPoly` as the exponents of z for the nonzero terms of the polynomial in descending order of powers:

```
h = commsrc.pn('GenPoly', [6 1 0], 'Mask', [1 1 0 1 0 1])

h =
          GenPoly: [1 0 0 0 0 1 1]
    InitialStates: [0 0 0 0 0 1]
    CurrentStates: [0 0 0 0 0 1]
             Mask: [1 1 0 1 0 1]
       NumBitsOut: 1
```

</td>
<td>

Define this PN sequence generator as:

```
h1 = comm.PNSequence('Polynomial', [1 0 0 0 0 1 1],..
 'InitialConditions', [1 1 0 1 0 1]);
h2 = comm.PNSequence('Polynomial', [1 0 0 0 0 1 1],..
 'Mask', 22);
mask2shift ([1 0 0 0 0 1 1],[1 1 0 1 0 1])

ans = 22
```

Alternatively you can input the polynomial exponents of `z` for the nonzero terms of the polynomial in descending order of powers.

```
h = comm.PNSequence('Polynomial',...
 [6 1 0], 'InitialConditions', [1 1 0 1 0 1])

h =
  comm.PNSequence with properties:

               Polynomial: [6 1 0]
    InitialConditionsSource: 'Property'
        InitialConditions: [1 1 0 1 0 1]
               MaskSource: 'Property'
                     Mask: 0
       VariableSizeOutput: false
          SamplesPerFrame: 1
           ResetInputPort: false
          BitPackedOutput: false
           OutputDataType: 'double'
```

</td>
</tr>
</table>

See Also

`comm.PNSequence` | `mask2shift` | `shift2mask`

Introduced in R2009a

commtest.ErrorRate

(To be removed) Create error rate test console

Compatibility

commtest.ErrorRate will be removed in a future release. Use `comm.ErrorRate` or `bertool` instead. For more information, see "Compatibility Considerations" on page 2-215.

Syntax

```
h = commtest.ErrorRate
h = commtest.ErrorRate(sys)
h = commtest.ErrorRate(sys,'PropertyName',PropertyValue,...)
h = commtest.ErrorRate('PropertyName',PropertyValue,...)
```

Description

`h = commtest.ErrorRate` returns an error rate test console, h. The error rate test console runs simulations of a system under test to obtain error rates.

`h = commtest.ErrorRate(sys)` returns an error rate test console, error rate test console, h, with each specified property set to the h, with an attached system under test, SYS.

`h = commtest.ErrorRate(sys,'PropertyName',PropertyValue,...)` returns an error rate test console, h, with an attached system under test, sys. Each specified property, 'PropertyName', is set to the specified value, PropertyValue.

`h = commtest.ErrorRate('PropertyName',PropertyValue,...)` returns an error rate test console, h, with each specified property 'PropertyName', set to the specified value, PropertyValue.

Properties

The error rate test console object has the properties in the following table. Setting any property resets the object. A property that is irrelevant is one that you can set, but its value does not affect measurements. Similarly, you cannot display irrelevant properties using the `disp` method. You can write to all properties, except for the ones explicitly noted otherwise.

Property	Description
Description	'Error Rate Test Console'. Read-only.
SystemUnderTestName	System under test name. Read-only.

Property	Description
FrameLength	Specify the length of the transmission frame at each iteration. This property becomes relevant only when the system under test registers a valid test input. • If the system under test registers a `NumTransmissions` test input and calls its `getInput` method, the error rate test console returns the value stored in `FrameLength`. Using an internal data source, the system under test uses this value to generate a transmission frame of the specified length. • If the system under test registers a `DiscreteRandomSource` test input and calls its `getInput` method, the test console generates and returns a frame of symbols. The length of the frame of symbols matches the `FrameLength` property. This property defaults to 500.
IterationMode	Specify how the object determines simulation points. • If set to `Combinatorial`, the object performs simulations for all possible combinations of registered test parameter sweep values. • If set to `Indexed`, the object performs simulations for all indexed sweep value sets. The i^{th} sweep value set consists of the i^{th} element of every sweep value vector for each registered test parameter. All sweep value vectors must have equal length, except for values that are unit length. Note that for the following sweep parameter settings: • Parameter1 = [a_1 a_2] • Parameter2 = [b_1 b_2] • Parameter3 = [c_1] In Indexed Mode, the test console performs simulations for the following sweep parameter sets: (a_1, b_1, c_1) (a_2, b_2, c_1) In Combinatorial Mode, the test console performs simulations for the following sweep parameter sets: (a_1, b_1, c_1) (a_1, b_2, c_1) (a_2, b_1, c_1) (a_2, b_2, c_1)

Property	Description
SystemResetMode	Specify the stage of a simulation run at which the system resets. • Setting to `Reset at new simulation point` resets the system under test at the beginning of a new simulation point. • Setting to `Reset at every iteration` resets the system under test at every iteration.
SimulationLimitOption	Specify how to stop the simulation for each sweep parameter point. • If set to `Number of transmissions` the simulation for a sweep parameter point stops when the number of transmissions equals the value for `MaxNumTransmissions`. • Set `TransmissionCountTestPoint` to the name of the registered test point containing the transmission count you are comparing to `MaxNumTransmissions`. • If set to `Number of errors` the simulation for a sweep parameter point stops when the number of errors equals the value for `MinNumErrors`. • Set the `ErrorCountTestPoint` to the name of the registered test point containing the error count you are comparing to the `MinNumErrors`. • Setting to `Number of errors or transmissions` stops the simulation for a sweep parameter point when meeting one of two conditions. • The simulation stops when the number of transmissions equals the value for `MaxNumTransmissions`. • The simulation stops when obtaining the number of errors matching `NumErrors`. • Setting this property to `Number of errors and transmissions` stops the simulation for a sweep parameter point when meeting the following condition. • The simulation stops when the number of transmissions *and* the number errors have at least reached the values in `MinNumTransmissions` and `MinNumErrors`. Set `TransmissionCountTestPoint` to the name of the registered test point that contains the transmission count you are comparing to the `MaxNumTransmissions` property. To control the simulation length, set `ErrorCountTestPoint` to the name of the registered test point containing the error count you are comparing to `MinNumErrors`. Call the `info` method of the error rate test console to see the valid registered test point names.

Property	Description
MaxNumTransmissions	Specify the maximum number of transmissions the object counts before stopping the simulation for a sweep parameter point. This property becomes relevant only when `SimulationLimitOption` is `Number of transmissions` or `Number of errors or transmissions`. • When setting `SimulationLimitOption` to `Number of transmissions` the simulation for each sweep parameter point stops when reaching the number of transmissions `MaxNumTransmissions` specifies. • Setting `SimulationLimitOption` to `Number of errors or transmissions` stops the simulation for each sweep parameter point for one of two conditions. • The simulation stops when completing the number of transmissions `MaxNumTransmissions` specifies. • The simulation stops when obtaining the number of errors `MinNumErrors` specifies. The `TransmissionCountTestPoint` property supplies the name of a registered test point containing the count transmission type. Calling the `info` method of the error rate test console displays the valid registered test points. If this property contains registered test points, the test console runs iterations equal to the value for `MaxNumTransmissions` for each sweep parameter point. If this property has no registered test parameters, the test console runs the number of iterations equal to the value for `MaxNumTransmissions` and stops. The value defaults to 1000.
MinNumErrors	Specify the minimum number of errors the object counts before stopping the simulation for a sweep parameter point. This property becomes relevant only when setting the `SimulationLimitOption` to `Number of errors` or `Number of errors or transmissions`. • When setting `SimulationLimitOption` to `Number of errors` the simulation for each parameter point stops when reaching the number of errors you specify for the `MinNumErrors` property. • When setting the `SimulationLimitOption` property to `Number of errors or transmissions` the simulation for each sweep parameter point stops for one of two conditions. • The simulation stops when reaching the number of errors you specify for the `MaxNumTransmissions` property. • The simulation stops when reaching the number of errors you specify for the `MinNumErrors` property. Specify the type of errors the error count uses by setting the `ErrorCountTestPoint` property to the name of a registered test point containing the count. Call the `info` method of the error rate test console to see the valid registered test point names. This value defaults to 100.

Property	Description
TransmissionCountTestPoint	Specify and register a test point containing the transmission count that controls the test console simulation stop mechanism. This property becomes relevant only when setting `SimulationLimitOption` to `Number of transmissions`, `Number of errors or transmissions`, or `Number of errors and transmissions`. In this scenario, if you register a test point, and `TransmissionCountTestPoint` equals `Not set`, the value of this property automatically updates to that of the registered test point name. Call the `info` method to see the valid test point names.
ErrorCountTestPoint	Specify and register the name of a test point containing the error count that controls the simulation stop mechanism. This property is only relevant when setting the `SimulationLimitOption` property to `Number of errors`, `Number of errors or transmissions`, or `Number of errors and transmissions`. In this scenario, if you register a test point, and `ErrorCountTestPoint` equals `Not set`, the value of this property automatically updates to that of the registered test point name. Call the `info` method to see the valid test point names.

Methods

The error rate test console object has the following methods:

run

Runs a simulation.

Runs the number of error rate simulations you specify for a system under test with a specified set of parameter values. If a Parallel Computing Toolbox™ license is available and a parpool is open, then the object distributes the iterations among the number of workers available.

getResults

Returns the simulation results.

r = getResults(h) returns the simulation results, *r*, for the test console, *h*. *r* is an object of the type you specify using testconsole.Results. It contains the simulation data for all the registered test points and methods to parse the data and plot it.

info

Returns a report of the current test console settings.

info(h) displays the current test console settings, such as registered test parameters and registered test points.

reset

Resets the error rate test console.

reset(h) resets test parameters and test probes and then clears all simulation results of test console, *h*.

attachSystem

Attaches a system to test console.

attachSystem(ho,sys) attaches a valid user-defined system, sys, to the test console, h.

detachSystem

Detaches the system from the test console.

detachSystem(h) detaches a system from the test console, h. This method also clears the registered test inputs, test parameters, test probes, and test points.

setTestParameterSweepValues

Sets test parameter sweep values.

setTestParameterSweepValues(h,name,sweep) specifies a set of sweep values, 'sweep', for the registered test parameter, 'name', in the test console, h. You only specify sweep values for registered test parameters. sweep must have values within the specified range of the test parameter. It can be a row vector of numeric values, or a cell array of char values. Display the valid ranges using the getTestParameterValidRanges method.

setTestParameterSweepValues(h,name1,sweep1,name2,sweep2...) simultaneously specifies sweep values for multiple registered test parameters.

getTestParameterSweepValues

Returns test parameter sweep values.

getTestParameterSweepValues(h,name) gets the sweep values currently specified for the registered test parameter, name, in the test console, h.

getTestParameterValidRanges

Returns the test parameter valid ranges.

getTestParameterValidRanges(h,name) gets the valid ranges for a registered test parameter, name, in the test console, h.

registerTestPoint

Registers a test point.

registerTestPoint(h, name, actprobe,expprobe) registers a new test point object, name, to the error rate test console, h. The test point must contain a pair of registered test probes, actprobe, and expprobe. actprobe contains actual data, and expprobe contains expected data. The object compares the data from these probes and obtains error rate values. The error rate calculation uses a default error rate calculator function that simply performs one-to-one comparisons of the data vectors available in the probes.

registerTestPoint(h, name, actprobe,expprobe, handle) adds the handle, handle, to a user-defined error calculation function that compares the data in the probes and then obtains error rate results.

The user-defined error calculation function must comply with the following syntax: [ecnt tcnt] = functionName(act, exp, udata) where

- ecnt output corresponds to the error count
- tcnt output is the number of transmissions used to obtain the error count
- act and exp correspond to actual and expected data

The error rate test console sets the inputs to the data available in the pair of test point probes, actprobe, and expprobe.

udata is a data input that the system under test passes to the test console at run time, using the setUserData method. udata contains the data necessary to compute errors, such as delays and data buffers.

The error rate test console passes the data that the system under test logs to the error calculation functions for all the registered test points. Calling the info method returns the names of the registered test points and the error rate calculator functions associated with them. It also returns the names of the registered test probes.

unregisterTestPoint

Unregister a test point.

unregisterTestPoint(h,name) removes the test point, name, from the test console, h.

Examples

Error Rate Simulation Sweeps

The `commtest.ErrorRate` and `testconsole.Results` object packages will be removed in a future release. They can be used to perform parameter sweeps to analyze communication system performance. This example demonstrates a workflow that uses them and along with recommended alternate workflows.

Multiple Parameter Sweep and Parallel Run using commtest.ErrorRate

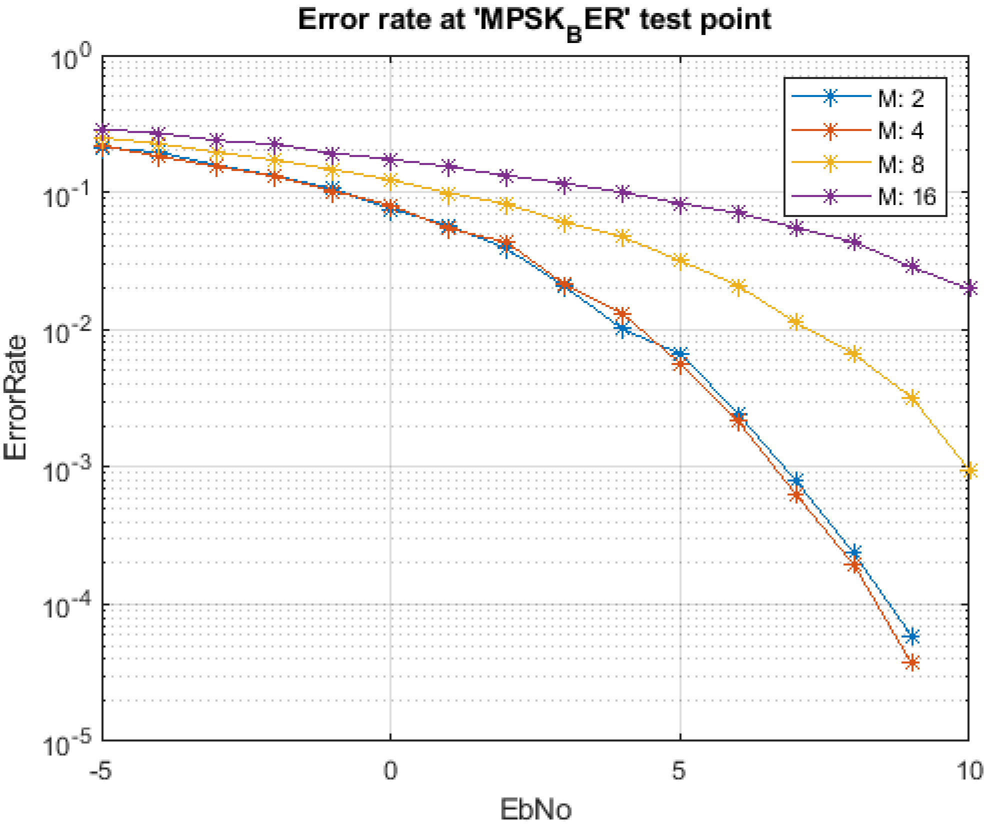
Obtain bit error rate and symbol error rate of an M-PSK system for different modulation orders and EbNo values. System under test is `commtest.MPSKSystem`.

```
% Create an M-ary PSK system
systemUnderTest = commtest.MPSKSystem;

% Instantiate an Error Rate Test Console and attach the system
errorRateTester = commtest.ErrorRate(systemUnderTest);
```

```
Warning: commtest.ErrorRate will be removed in the future. Use comm.ErrorRate or bertool instead
```

```
errorRateTester.SimulationLimitOption = 'Number of errors or transmissions';
errorRateTester.MaxNumTransmissions = 1e5;

% Set sweep values for simulation test parameters
setTestParameterSweepValues(errorRateTester,'M',2.^[1 2 3 4],'EbNo',(-5:10))

% Register a test point
registerTestPoint(errorRateTester,'MPSK_BER','TxInputBits','RxOutputBits')

% Get information about the simulation settings
info(errorRateTester)
```

```
Test console name:            commtest.ErrorRate
System under test name:       commtest.MPSKSystem
Available test inputs:        NumTransmissions, RandomIntegerSource
Registered test inputs:       NumTransmissions
Registered test parameters:   EbNo, M
Registered test probes:       RxOutputBits, RxOutputSymbols, TxInputBits, TxInputSymbols
Registered test points:       MPSK_BER
Metric calculator functions: @commtest.ErrorRate.defaultErrorCalculator
Test metrics:                 ErrorCount, TransmissionCount, ErrorRate

% Run the M-PSK simulations
run(errorRateTester)

Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
12 workers available for parallel computing. Simulations will be distributed among these workers
Running simulations...

% Get the results
mpskResults = getResults(errorRateTester);

Warning: testconsole.Results will be removed in the future. See <a href="matlab:helpview(fullfile

% Get a semi-log scale plot of EbNo versus bit error rate for
% different values of modulation order M
mpskResults.TestParameter2 = 'M';
semilogy(mpskResults,'*-')
```



Multiple Parameter Sweep and Parallel Run using nested for loops and comm.ErrorRate

Run an error rate simulation over M=2.^(1:4) and EbNo=-5:10. Use `comm.ErrorRate` to collect both bit error rate (BER) and symbol error rate (SER) data. Run the simulations to collect a minimum of 100 symbol errors or for a maximum of 1e5 symbols.

```
% Set the M sweep values same as the commtest.ErrorRate object
getTestParameterSweepValues(errorRateTester,'M')
```

```
ans = 1×4

     2     4     8    16
```

```
MSweep = 2.^[1 2 3 4];

% Set EbNo sweep values same as the commtest.ErrorRate object
getTestParameterSweepValues(errorRateTester,'EbNo')
```

```
ans = 1×16

    -5    -4    -3    -2    -1     0     1     2     3     4     5     6     7     8     9    10
```

```
EbNoSweep = -5:10;

% Set minumum number of errors same as the commtest.ErrorRate object
errorRateTester.MinNumErrors
```

```
ans = 100
```

```
minNumErrors = 100;

% Set maximum number of transmissions same as the commtest.ErrorRate
% object. In this example a transmission is a symbol.
errorRateTester.MaxNumTransmissions
```

```
ans = 100000
```

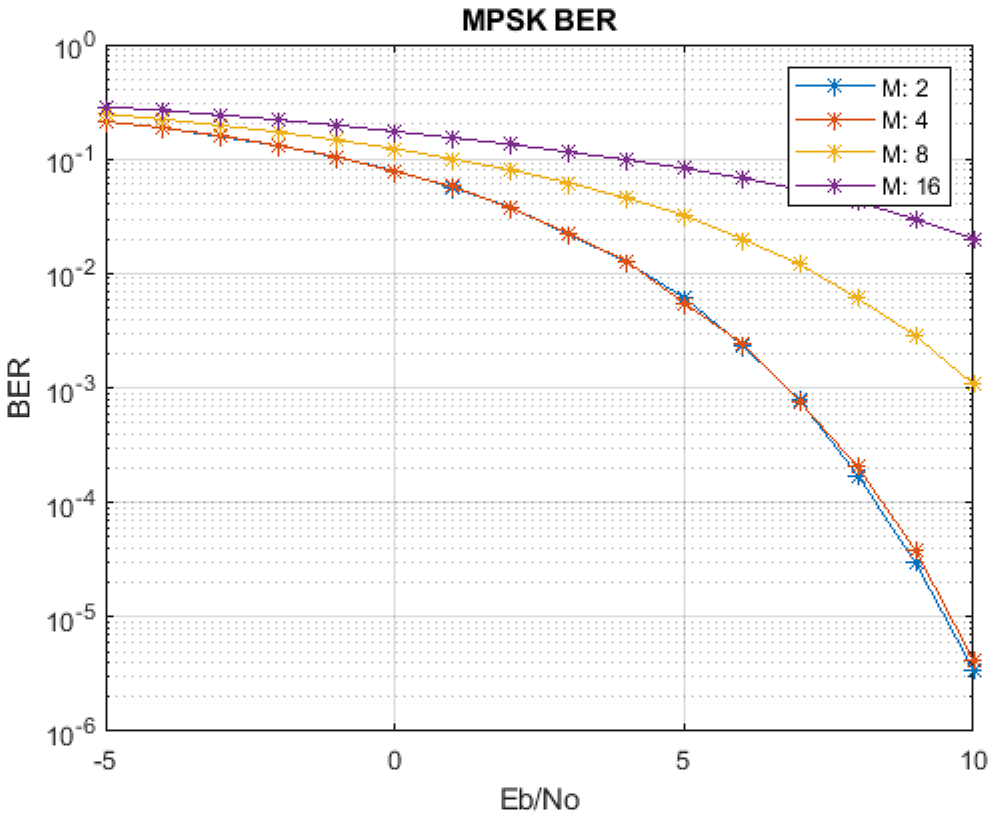
```
MaxNumTransmissions = 1e5;

% Set frame length same as the commtest.ErrorRate object
errorRateTester.FrameLength
```

```
ans = 500
```

```
frameLength = 500;

% Find out if there is a parallel pool and how many workers are available
[licensePCT,~] = license('checkout','distrib_computing_toolbox');
if (licensePCT && ~isempty(ver('parallel')))
    p = gcp;
    if isempty(p)
        numWorkers = 1;
    else
        numWorkers = p.NumWorkers
    end
else
    numWorkers = 1;
end
```

```
numWorkers = 12
```

```
minNumErrorsPerWorker = minNumErrors/numWorkers;
maxNumSymbolsPerWorker = MaxNumTransmissions/numWorkers;

% Store results in an array, where first dimension is M and second
% dimension is EbNo. Initialize the vector with NaN values.
ser = nan(length(MSweep),length(EbNoSweep));
ber = nan(length(MSweep),length(EbNoSweep));

% First sweep is over M (modulation order)
for MIdx = 1:length(MSweep)
    M = MSweep(MIdx);
    bitsPerSymbol = log2(M);

    % Second sweep is over EbNo
    for EbNoIdx = 1:length(EbNoSweep)
        EbNo = EbNoSweep(EbNoIdx);

        SNR = EbNo+10*log10(bitsPerSymbol);

        numSymbolErrors = zeros(numWorkers,1);
        numBitErrors = zeros(numWorkers,1);
        numSymbols = zeros(numWorkers,1);

        parfor worker = 1:numWorkers
            symErrRate = comm.ErrorRate;
            bitErrRate = comm.ErrorRate;

            while (numSymbolErrors(worker) < minNumErrorsPerWorker) ...
                    || (numSymbols(worker) < maxNumSymbolsPerWorker)
                % Generate frameLength source outputs
                txMsg = randi([0 M-1],frameLength,1);

                % Modulate the data
                txOutput = pskmod(txMsg,M,0,'gray');
                % Pass data through an AWGN channel with current SNR value
                chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
                % Demodulate the data
                rxOutput = pskdemod(chnlOutput,M,0,'gray');

                % Calculate number of symbol errors
                symErrVal = symErrRate(txMsg,rxOutput);
                numSymbolErrors(worker) = symErrVal(2);
                numSymbols(worker) = symErrVal(3);

                % Convert symbol streams to bit streams
                bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
                bTx = bTx(:);
                bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
                bRx = bRx(:);

                % Calculate number of bit errors
                bitErrVal = bitErrRate(bTx,bRx);
                numBitErrors(worker) = bitErrVal(2);
            end
        end
        ber(MIdx,EbNoIdx) = sum(numBitErrors)/(sum(numSymbols)*bitsPerSymbol);
        ser(MIdx,EbNoIdx) = sum(numSymbolErrors)/sum(numSymbols);
    end
end

% Plot results
semilogy(EbNoSweep,ber,'*-')
grid on
title('MPSK BER')
xlabel('Eb/No')
ylabel('BER')
legendText = cell(length(MSweep),1);
for p=1:length(MSweep)
    legendText{p} = sprintf('M: %d',MSweep(p));
end
legend(legendText)
```



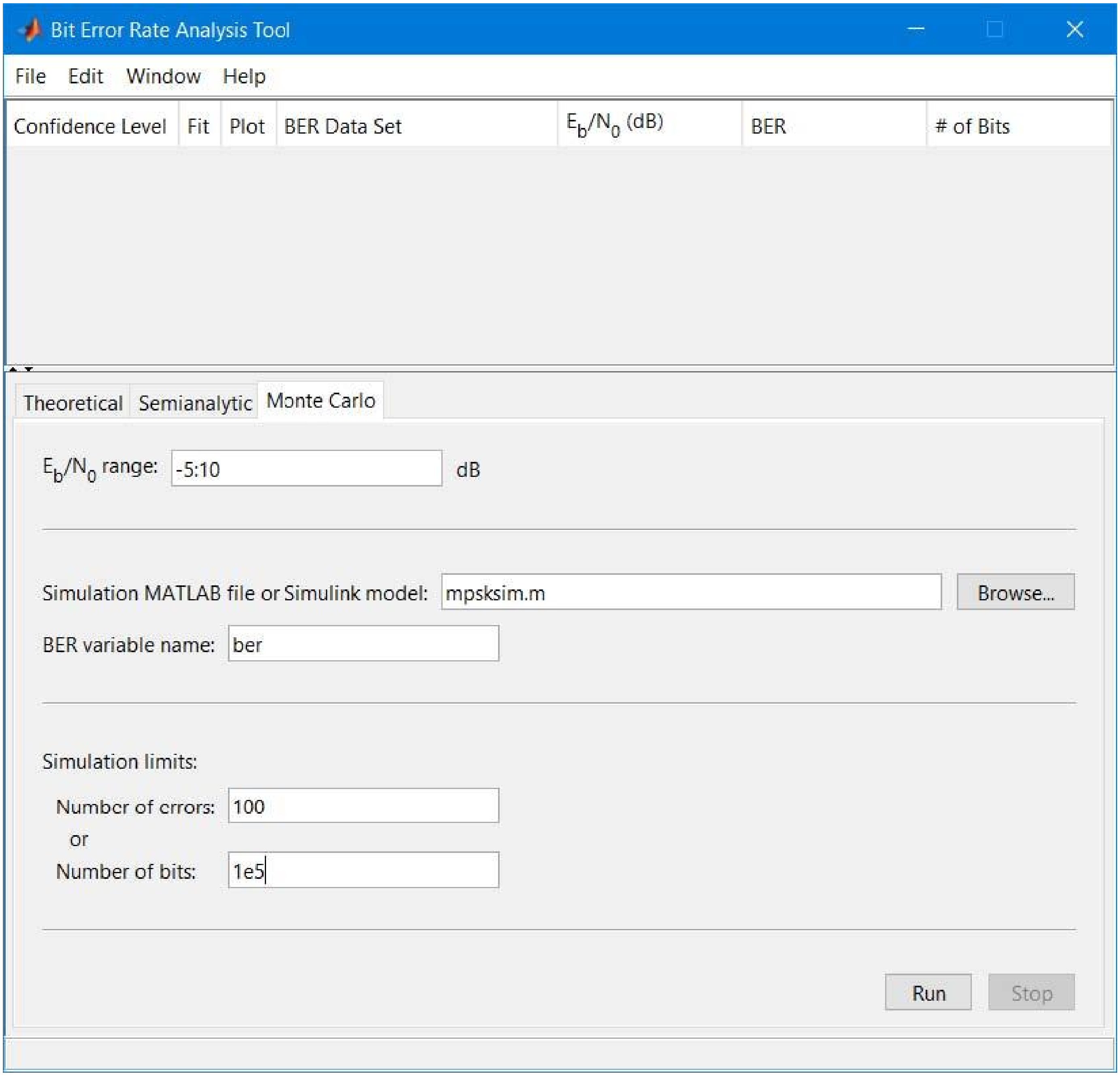
Multiple Variable Sweeps using BERTool

`BERTool` computes the BER as a function of signal-to-noise ratio. It analyzes performance either with Monte-Carlo simulations of MATLAB® functions and Simulink® models or with theoretical closed-form expressions for selected types of communication systems. The `bertool` function opens the `BERTool`. Here `BERTool` is configured to call the simulation defined in the function `mpsksim` included below.

```
function [ber,numBits] = mpsksim(EbNo,minNumErrs,maxNumBits)
% Import the Java class for BERTool, so that you will be able to stop the simulation using the "S
import com.mathworks.toolbox.comm.BERTool;
```

```
frameLength = 500;

M = 16; % Can be 2, 4, 8, 16
bitsPerSymbol = log2(M);

maxNumSymbols = maxNumBits/bitsPerSymbol;

SNR = EbNo + 10*log10(bitsPerSymbol);

% Initialize variables related to exit criteria.
numBitErrors = 0;
numSymbols = 0;

while (numBitErrors < minNumErrs) || (numSymbols < maxNumSymbols)

  % Check if the user clicked the Stop button of BERTool.
  if (BERTool.getSimulationStop)
    break;
  end

  % Generate frameLength source outputs
  txMsg = randi([0 M-1],frameLength,1);
  numSymbols = numSymbols+frameLength;

  % Modulate the data
  txOutput = pskmod(txMsg,M,0,'gray');
  % Pass data through an AWGN channel with current SNR value
  chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
  % Demodulate the data
  rxOutput = pskdemod(chnlOutput,M,0,'gray');

  % Convert symbol streams to bit streams
  bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
  bTx = bTx(:);
  bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
  bRx = bRx(:);

  % Calculate number of bit errors
  numBitErrors = numBitErrors+sum(bTx~=bRx);
end

% Assign values to the output variables.
numBits = numSymbols*bitsPerSymbol;
ber = numBitErrors/numBits;
```

Configure `BERTool` as follows.



E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

BER variable name: ber

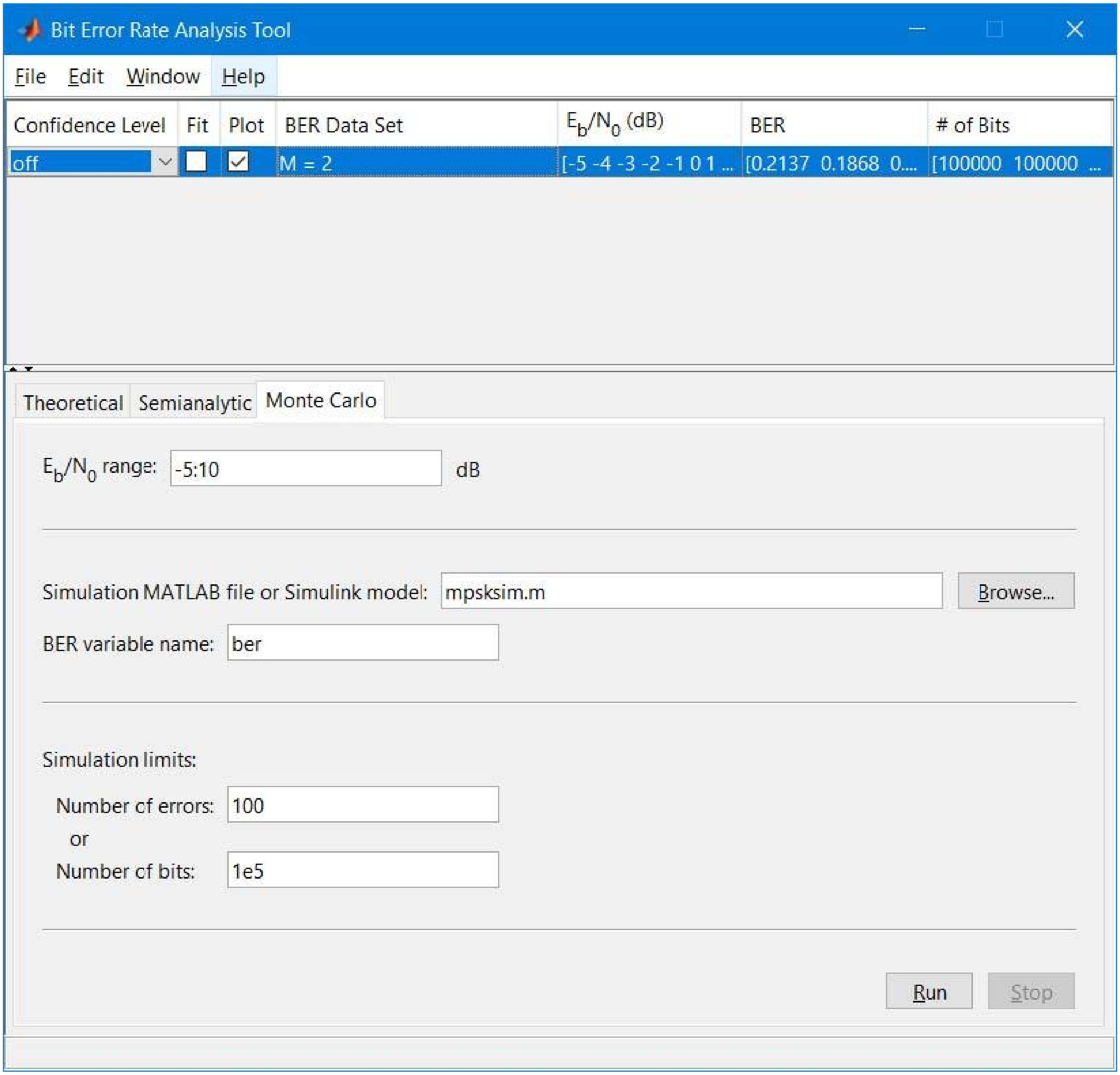
Simulation limits:

Number of errors: 100

or

Number of bits: 1e5

Set M=2 in the `mpsksim` function and click Run. Set the **BER Data Set** name to `'M=2'`.



Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	□	☑	M = 2	[-5 -4 -3 -2 -1 0 1 ...	[0.2137 0.1868 0....	[100000 100000 ...

E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

BER variable name: ber

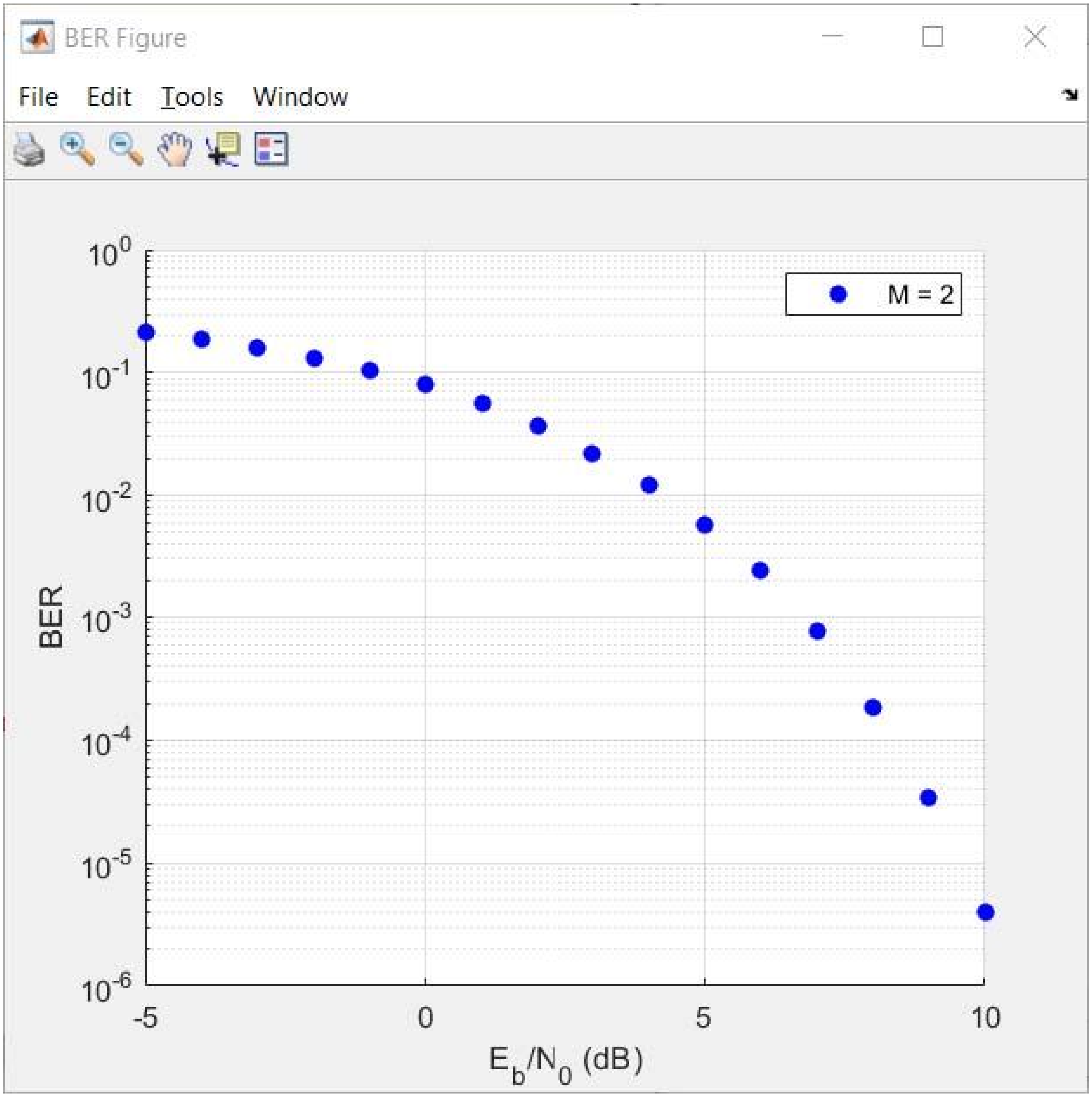
Simulation limits:

Number cf errors: 100

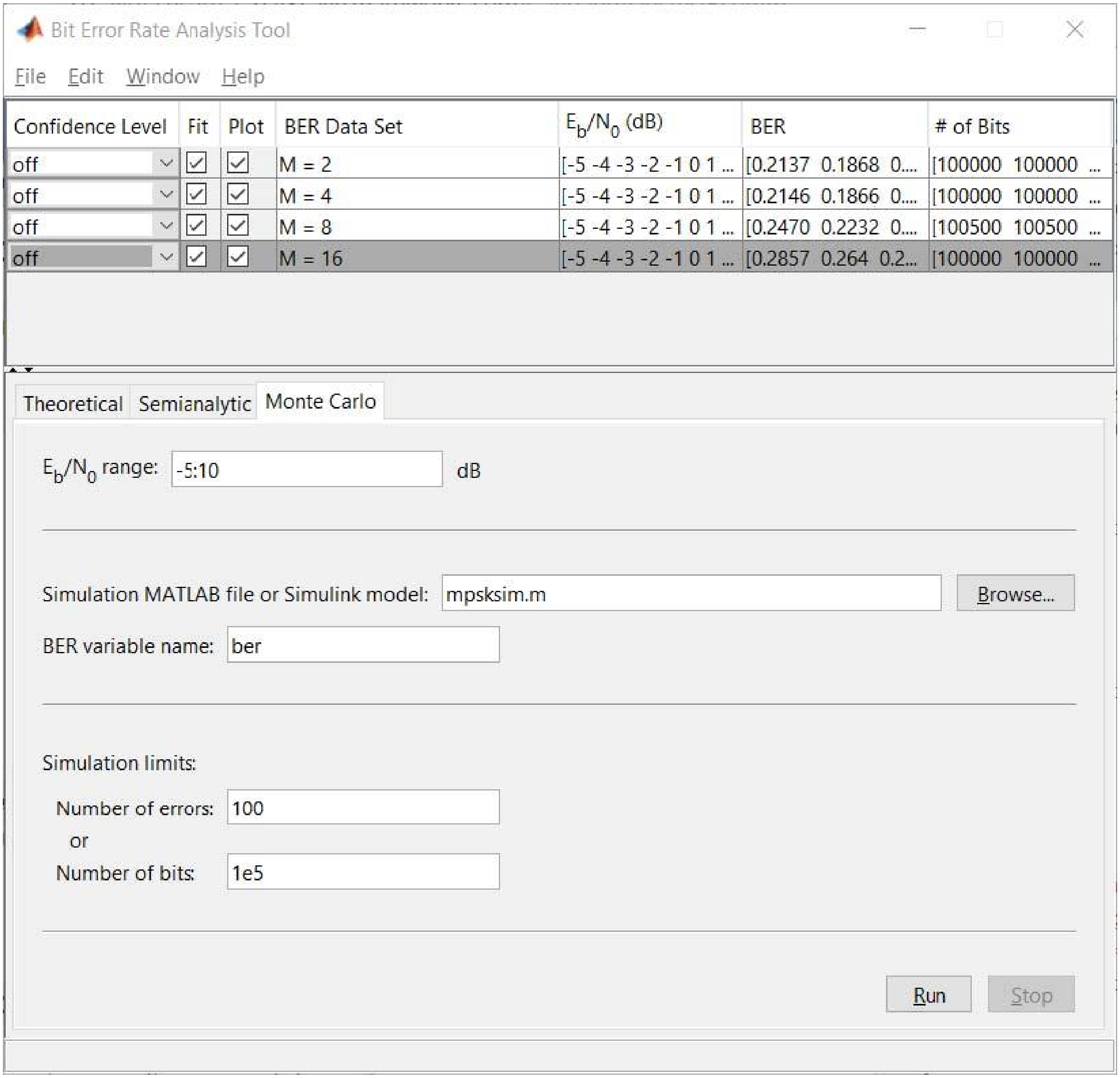
or

Number cf bits: 1e5

Display the BER curve for M=2.



Update the value for `M` in the `mpsksim` function, repeating this process for `M` = `4`, `8`, `16`. You will see results similar to those below in the **Bit Error Rate Analysis Tool** window and the **BER** figure.



Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☑	☑	M = 2	[-5 -4 -3 -2 -1 0 1 ...	[0.2137 0.1868 0....	[100000 100000 ...
off	☑	☑	M = 4	[-5 -4 -3 -2 -1 0 1 ...	[0.2146 0.1866 0....	[100000 100000 ...
off	☑	☑	M = 8	[-5 -4 -3 -2 -1 0 1 ...	[0.2470 0.2232 0....	[100500 100500 ...
off	☑	☑	M = 16	[-5 -4 -3 -2 -1 0 1 ...	[0.2857 0.264 0.2...	[100000 100000 ...

E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

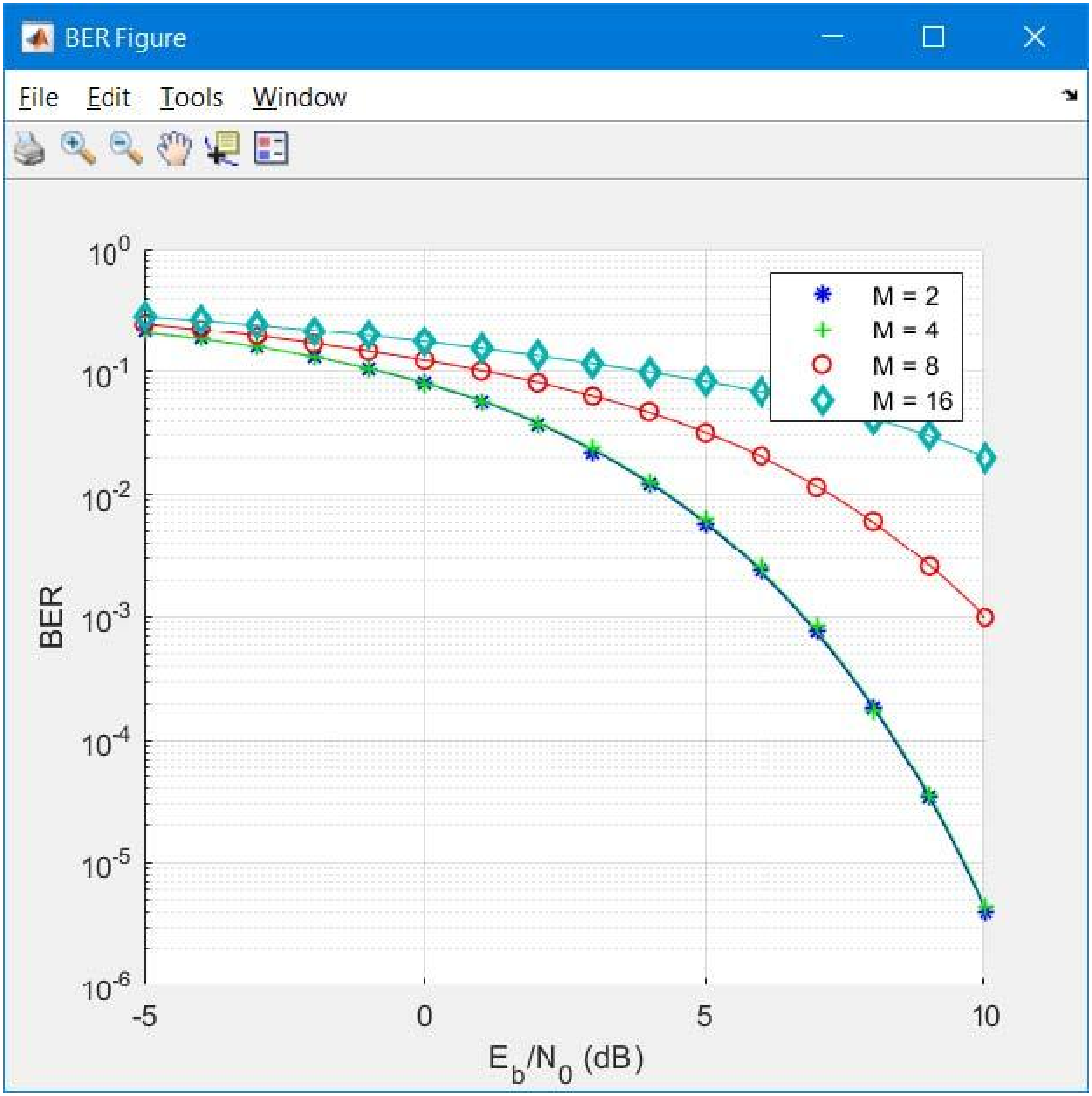
BER variable name: ber

Simulation limits:

Number of errors: 100

or

Number of bits: 1e5



Parallel SNR Sweep using BERTool

Using `parfor`, run each simulation point in parallel by configuring your simulation function similar to the `mpsksim_parfor` function included below. Since `parfor` cannot work with the Java class for `BERTool`, you will not be able to stop the simulation using the **Stop** button.

```
function [ber,numBits] = mpsksim_parfor(EbNo,minNumErrs,maxNumBits)

% Find out if there is a parallel pool and how many workers are available
if license('test','Distrib_Computing_Toolbox')
  p = gcp;
  if isempty(p)
    numWorkers = 1;
  else
    numWorkers = p.NumWorkers;
  end
else
  numWorkers = 1;
end

M = 2;
bitsPerSymbol = log2(M);

maxNumSymbols = maxNumBits/bitsPerSymbol;

minNumErrorsPerWorker = minNumErrs/numWorkers;
maxNumSymbolsPerWorker = maxNumSymbols/numWorkers;
frameLength = 500;

SNR = EbNo + 10*log10(bitsPerSymbol);

% Initialize variables related to exit criteria.
numBitErrors = zeros(numWorkers,1);
numSymbols = zeros(numWorkers,1);

parfor worker = 1:numWorkers
  while (numBitErrors(worker) < minNumErrorsPerWorker) ...
      || (numSymbols(worker) < maxNumSymbolsPerWorker)

    % Generate frameLength source outputs
    txMsg = randi([0 M-1],frameLength,1);
    numSymbols(worker) = numSymbols(worker)+frameLength;

    % Modulate the data
    txOutput = pskmod(txMsg, M, 0, 'gray');
    % Pass data through an AWGN channel with current SNR value
    chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
    % Demodulate the data
    rxOutput = pskdemod(chnlOutput,M,0,'gray');

    % Convert symbol streams to bit streams
    bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
    bTx = bTx(:);
    bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
    bRx = bRx(:);

    % Calculate number of bit errors
    numBitErrors(worker) = numBitErrors(worker)+sum(bTx~=bRx);
  end
end

% Assign values to the output variables.
ber = sum(numBitErrors)/sum(numSymbols);
numBits = sum(numSymbols)*bitsPerSymbol;
```

Compatibility Considerations

`commtest.ErrorRate` will be removed

Warns starting in R2019b

commtest.ErrorRate will be removed in a future release. Use `comm.ErrorRate` or `bertool` instead. The “Error Rate Simulation Sweeps” on page 2-204 example demonstrates alternate workflows using `comm.ErrorRate` and `bertool`.

See Also

Objects
`comm.ErrorRate`

Functions
`bertool`

Topics
“Bit Error Rate (BER)”

Introduced in R2009b

compand

Source coding mu-law or A-law compressor or expander

Syntax

```
out = compand(in,param,v)
out = compand(in,param,v,method)
```

Description

`out = compand(in,param,v)` performs mu-law compression on the input data sequence. The `param` input specifies the mu-law compression value and must be set to a mu value for mu-law compressor computation (a mu-law value of 255 is used in practice). `v` specifies the peak magnitude of the input data sequence.

`out = compand(in,param,v,method)` performs mu-law or A-law compression or expansion on the input data sequence. `param` specifies the mu-law compander or A-law compander value (a mu-law value of 255 and an A-law value of 87.6 are used in practice). `method` specifies the type of compressor or expander computation for the function to perform on the input data sequence.

Examples

Compress and Expand Data Sequence Using Mu-Law

Generate a data sequence.

```
data = 2:2:12
```

```
data = 1×6

     2     4     6     8    10    12
```

Compress the data sequence by using a mu-law compressor. Set the value for mu to 255. The compressed data sequence now ranges between 8.1 and 12.

```
compressed = compand(data,255,max(data),'mu/compressor')
```

```
compressed = 1×6

    8.1644    9.6394   10.5084   11.1268   11.6071   12.0000
```

Expand the compressed data sequence by using a mu-law expander. The expanded data sequence is nearly identical to the original data sequence.

```
expanded = compand(compressed,255,max(data),'mu/expander')
```

```
expanded = 1×6
```

```
    2.0000    4.0000    6.0000    8.0000   10.0000   12.0000
```

Calculate the difference between the original data sequence and the expanded sequence.

```
diffvalue = expanded - data
```

```
diffvalue = 1×6
10^-14 ×

   -0.0444    0.1776    0.0888    0.1776    0.1776   -0.3553
```

Compress and Expand Data Sequence Using A-Law

Generate a data sequence.

```
data = 1:5;
```

Compress the data sequence by using an A-law compressor. Set the value for A to 87.6. The compressed data sequence now ranges between 3.5 and 5.

```
compressed = compand(data,87.6,max(data),'A/compressor')
```

```
compressed = 1×5

    3.5296    4.1629    4.5333    4.7961    5.0000
```

Expand the compressed data sequence by using an A-law expander. The expanded data sequence is nearly identical to the original data sequence.

```
expanded = compand(compressed,87.6,max(data),'A/expander')
```

```
expanded = 1×5

    1.0000    2.0000    3.0000    4.0000    5.0000
```

Calculate the difference between the original data sequence and the expanded sequence.

```
diffvalue = expanded - data
```

```
diffvalue = 1×5
10^-14 ×

         0         0    0.1332    0.0888    0.0888
```

Quantize and Compand an Exponential Signal

Set the mu-law parameter.

```
mu = 255;
```

Create an exponential signal and calculate its maximum value.

```
sig = exp(-4:0.1:4);
V = max(sig);
```

Quantize the signal by using equal-length intervals. Set partition and codebook values, assuming 6-bit quantization. Calculate the mean square distortion.

```
partition = 0:2^6 - 1;
codebook = 0:2^6;
[~,~,distortion] = quantiz(sig,partition,codebook);
```

Compress the signal by using the `compand` function. Apply quantization and expand the quantized signal. Calculate the mean square distortion of the companded signal.

```
compsig = compand(sig,mu,V,'mu/compressor');
[~,quants] = quantiz(compsig,partition,codebook);
newsig = compand(quants,mu,max(quants),'mu/expander');
distortion2 = sum((newsig - sig).^2)/length(sig);
```

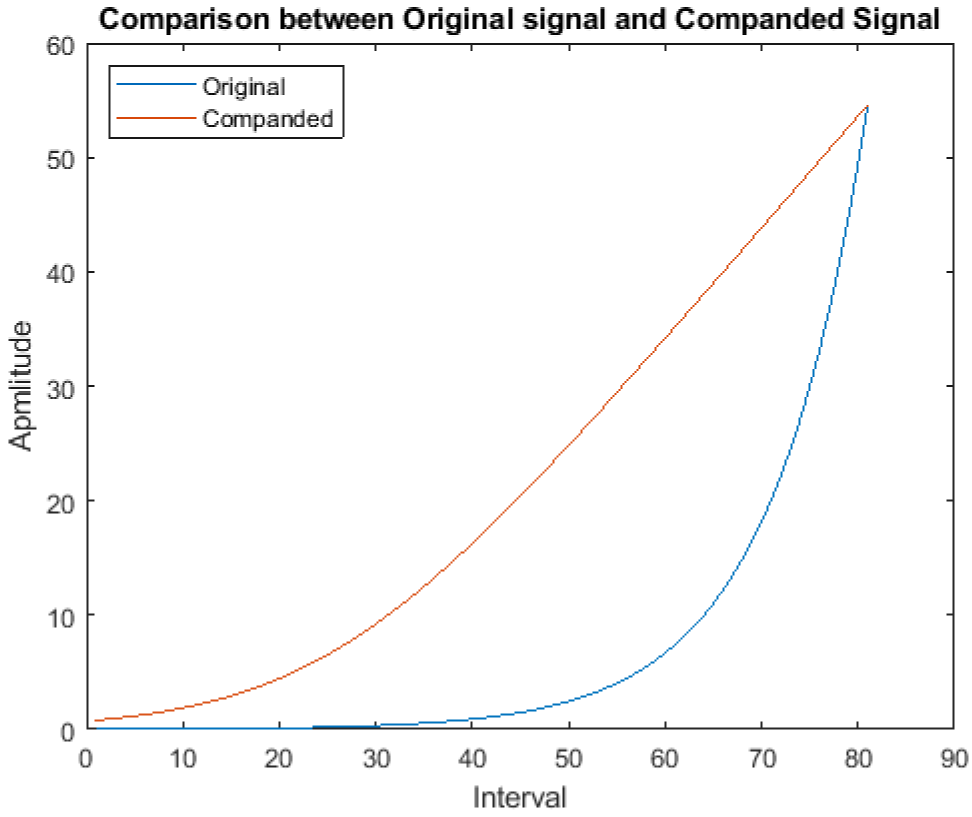
Compare the mean square distortions. The distortion is smaller when you perform companding. This difference is because equal-length intervals are well suited to the logarithm of the exponential signal but not well suited to exponential signal itself.

```
[distortion, distortion2]
```

```
ans = 1×2

    0.5348    0.0397
```

Plot the original exponential signal and the companded signal.

```
plot([sig' compsig']);
title('Comparison between Original signal and Companded Signal');
xlabel('Interval');
ylabel('Apmlitude');
legend('Original','Companded','location','nw');
```



Input Arguments

in — Input data sequence

row vector

Input data sequence, specified as a row vector. This input specifies the data sequence for the function to perform compression or expansion.

Data Types: `double`

param — mu or A value of compander

positive scalar | `255` | `87.6`

mu or A value of the compander, specified as a positive scalar. The prevailing values used in practice are µ = `255` and A = `87.6`.

Data Types: `double`

method — Type of compressor or expander computation

`mu/compressorr` | `mu/expander` | `A/compressor` | `A/expander`

Type of compressor or expander computation for the function to perform on the input data sequence, specified as one of these values.

- `mu/compressor`

- `mu/expander`
- `A/compressor`
- `A/expander`

Data Types: `char` | `string`

v — Peak magnitude of input data sequence
positive scalar

Peak magnitude of the input data sequence, specified as a positive scalar.

Data Types: `double`

Output Arguments

`out` — Compressed or expanded signal
positive row vector

Compressed or expanded signal, returned as a positive row vector. The size of `out` matches that of input argument `in`.

Algorithms

In certain applications, such as speech processing, using a logarithmic computation (called a compressor) before quantizing the input data is common. The inverse operation of a compressor is called an *expander*. The combination of a compressor and expander is called a *compander*.

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

$$y = \frac{\log(1 + \mu|x|)}{\log(1 + \mu)}\mathrm{sgn}(x) .$$

μ is the µ-law parameter of the compander, `log` is the natural logarithm, and `sgn` is the signum function (`sign` in MATLAB).

µ-law expansion for input signal x is given by the inverse function y^{1},

$$y^{-1} = \mathrm{sgn}(y)\left(\frac{1}{\mu}\right)\left((1 + \mu)^{|y|} - 1\right) \qquad \text{for -1} \le y \le 1$$

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

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

A is the A-law parameter of the compander, `log` is the natural logarithm, and `sgn` is the signum function (`sign` in MATLAB).

A-law expansion for input signal x is given by the inverse function y^{1},

$$y^{-1} = \mathrm{sgn}(y)\begin{cases} \dfrac{|y|(1 + \log(A))}{A} & \text{for } 0 \le |y| < \dfrac{1}{1 + \log(A)} \\ \dfrac{\exp(|y|(1 + \log(A)) - 1)}{A} & \text{for } \dfrac{1}{1 + \log(A)} \le |y| < 1 \end{cases}$$

References

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

See Also

Functions
`dpcmdeco` | `dpcmenco` | `huffmandeco` | `huffmanenco` | `lloyds` | `quantiz`

Introduced before R2006a

convdeintrlv

Restore ordering of symbols using shift registers

Syntax

```
deintrlved = convdeintrlv(data,nrows,slope)
[deintrlved,state] = convdeintrlv(data,nrows,slope)
[deintrlved,state] = convdeintrlv(data,nrows,slope,init_state)
```

Description

`deintrlved = convdeintrlv(data,nrows,slope)` restores the ordering of elements in `data` by using a set of `nrows` internal shift registers. The delay value of the kth shift register is `(nrows-k)*slope`, where k = 1, 2, 3,..., `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[deintrlved,state] = convdeintrlv(data,nrows,slope)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[deintrlved,state] = convdeintrlv(data,nrows,slope,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `convintrlv` function, use the same `nrows` and `slope` inputs in both functions. In that case, the two functions are inverses in the sense that applying `convintrlv` followed by `convdeintrlv` leaves data unchanged, after you take their combined delay of `nrows*(nrows-1)*slope` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers”.

Examples

The example in “Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB” uses `convdeintrlv` and illustrates how you can handle the delay of the interleaver/deinterleaver pair when recovering data.

The example on the reference page for `muxdeintrlv` illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

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

See Also

`convintrlv` | `muxdeintrlv`

Topics

"Interleaving"

Introduced before R2006a

convenc

Convolutionally encode binary message

Syntax

```
codedout = convenc(msg,trellis)
codedout = convenc(msg,trellis,puncpat)
codedout = convenc( ___ ,istate)
[codedout,fstate] = convenc( ___ )
```

Description

`codedout = convenc(msg,trellis)` encodes the input binary message by using a convolutional encoder represented by a trellis structure. For details about trellis structures in MATLAB, see “Trellis Description of a Convolutional Code”. The input message contains one or more symbols, each of which consists of log2(`trellis`.`numInputSymbols`) bits. The coded output, `codedout`, contains one or more symbols, each of which consists of log2(`trellis`.`numOutputSymbols`) bits.

`codedout = convenc(msg,trellis,puncpat)`specifies a puncture pattern, `puncpat`, to enable higher rate encoding than unpunctured coding.

For some commonly used puncture patterns for specific rates and polynomials, see the last three references.

`codedout = convenc( ___ ,istate)` enables the encoder registers to start at a state specified by `istate`. Specify `istate` as the last input parameter preceded by any of the input argument combinations in the previous syntaxes.

`[codedout,fstate] = convenc( ___ )` also returns the final state of the encoder. When calling `convenc` iteratively, `fstate` is typically used to set `istate` for subsequent calls to the `convenc` function.

Examples

Create Convolutional Codes

Create convolutional codes by using a trellis structure. You can define the trellis by using the `poly2trellis` function or by manually specifying the trellis structure. The example shows both methods.

Define trellis by using `poly2trellis` function

Define the trellis structure to be used to configure the encoder by using the `poly2trellis` function.

```
trellis_a = poly2trellis([5 4],[23 35 0; 0 5 13])
```

```
trellis_a = struct with fields:
     numInputSymbols: 4
    numOutputSymbols: 8
```

```
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

Use the trellis structure to configure the `convenc` function. Encode five two-bit symbols for a K/N rate 2/3 convolutional code by using the `convenc` function.

```
K = log2(trellis_a.numInputSymbols) % Number of input bit streams
```

```
K = 2
```

```
N = log2(trellis_a.numOutputSymbols) % Number of output bit streams
```

```
N = 3
```

```
numReg = log2(trellis_a.numStates) % Number of coder registers
```

```
numReg = 7
```

```
numSymPerFrame = 5; % Number of symbols per frame
data = randi([0 1],K*numSymPerFrame,1);
[code_a,fstate_a] = convenc(data,trellis_a);
```

Verify that the encoded output is 15 bits, which is 3/2 (N/K) times the length of the input sequence, `data`.

```
code_a'
```

```
ans = 1×15

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

```
length(data)
```

```
ans = 10
```

```
length(code_a)
```

```
ans = 15
```

Define trellis manually

Manually define a trellis structure for a K/N rate 1/2 convolutional code.

```
trellis_b = 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])
```

```
trellis_b = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 4
          nextStates: [4x2 double]
             outputs: [4x2 double]
```

Use the trellis structure to configure the `convenc` function when encoding 10 one-bit symbols.

```
K = log2(trellis_b.numInputSymbols) % Number of input bit streams
```

```
K = 1
```

```
N = log2(trellis_b.numOutputSymbols) % Number of output bit streams
```

```
N = 2
```

```
numReg = log2(trellis_b.numStates) % Number of coder registers
```

```
numReg = 2
```

```
numSymPerFrame = 10; % Number of symbols per frame
data = randi([0 1],K*numSymPerFrame,1);
code_b = convenc(data,trellis_b);
```

Verify that the encoded output is 20 bits, which is 2/1 (N/K) times the length of the input sequence, `data`.

```
code_b'
```

```
ans = 1×20

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

```
length(data)
```

```
ans = 10
```

```
length(code_b)
```

```
ans = 20
```

Adjust Convolutional Encoding Code Rate by Using Puncturing

Use puncturing to adjust the K/N code rate of the convolutional encoder from 1/2 to 3/4.

Initialize parameters for the encoding operation.

```
trellis = poly2trellis(7,[171 133])
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 64
          nextStates: [64x2 double]
             outputs: [64x2 double]
```

```
puncpat = [1;1;0];
```

Calculate the unpunctured and punctured code rates.

```
K = log2(trellis.numInputSymbols); % Number of input streams
N = log2(trellis.numOutputSymbols); % Number of output streams
unpunc_coderate = K/N; % Unpunctured code rate
punc_coderate = (K/N)*length(puncpat)/sum(puncpat); % Punctured code rate
fprintf('K is %d and N is %d. The unpunctured code rate is %3.2f and the punctured code rate is %
```

```
K is 1 and N is 2. The unpunctured code rate is 0.50 and the punctured code rate is 0.75.
```

Convolutionally encode an all 1s three-bit message without puncturing applied to the coded output. Then, convolutionally encode the same message with puncturing.

```
msg = ones(length(puncpat),1);
unpuncturedcode = convenc(msg,trellis);
puncturedcode = convenc(msg,trellis,puncpat);
```

Show the message, the unpunctured code, the punctured code, and the puncture pattern.

```
msg'
```

```
ans = 1×3

     1     1     1
```

```
unpuncturedcode'
```

```
ans = 1×6

     1     1     0     1     1     0
```

```
puncpat'
```

```
ans = 1×3

     1     1     0
```

```
puncturedcode'
```

```
ans = 1×4

     1     1     1     1
```

Without puncturing, the configured convolutional encoding inputs three message bits and outputs six coded bits. Confirm the resulting code rate matches the expected code rate of 1/2.

```
length(msg)/length(unpuncturedcode)
```

```
ans = 0.5000
```

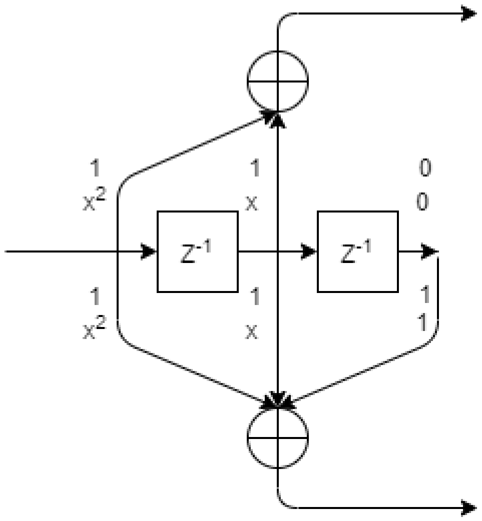
With puncturing, bits in positions 1 and 2 of the input message are transmitted, while the bit in position 3 is removed. For every three bits of input, the punctured code generates four bits of output. Confirm the resulting code rate matches the expected code rate of 3/4.

```
length(msg)/length(puncturedcode)
```

```
ans = 0.7500
```

Use Trellis Structure for Rate 1/2 Feedforward Convolutional Encoder

Use a trellis structure to configure the rate 1/2 feedforward convolutional code in this diagram.



Create a trellis structure, setting the constraint length to 3 and specifying the code generator as a vector of octal values. The diagram indicates the binary values and polynomial form, indicating the left-most bit is the most-significant-bit (MSB). The binary vector [1 1 0] represents octal 6 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 1 1] represents octal 7 and corresponds to the lower row of binary digits in the diagram. These binary digits indicate connections from the outputs of the registers to the two adders in the diagram.

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

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 4
          nextStates: [4x2 double]
             outputs: [4x2 double]
```

Generate random binary data. Convolutionally encode the data, by using the specified trellis structure. Decode the coded data by using the Viterbi algorithm with the specified trellis structure, 34 for its traceback depth, truncated operation mode, and hard decisions.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
tbdepth = 34;
decodedData = vitdec(codedData,trellis,tbdepth,'trunc','hard');
```

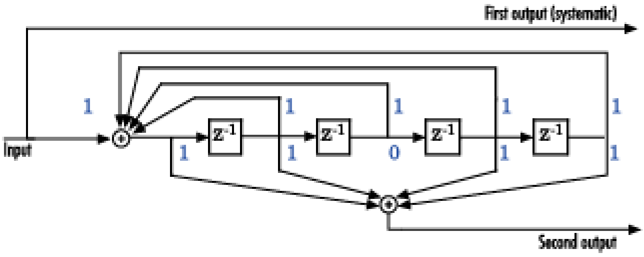
Verify the decoded data has zero bit errors.

```
biterr(data,decodedData)
```

```
ans = 0
```

Use Trellis Structure for Rate 1/2 Feedback Convolutional Encoder

Create a trellis structure to represent the rate 1/2 systematic convolutional encoder with feedback shown in this diagram.



This encoder has 5 for its constraint length, [37 33] as its generator polynomial matrix, and 37 for its feedback connection polynomial.

The first generator polynomial is octal 37. The second generator polynomial is octal 33. The feedback polynomial is octal 37. The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits.

The binary vector [1 1 1 1 1] represents octal 37 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 1 0 1 1] represents octal 33 and corresponds to the lower row of binary digits in the diagram. These binary digits indicate connections from the outputs of the registers to the two adders in the diagram. The initial 1 corresponds to the input bit.

Convert the polynomial to a trellis structure by using the `poly2trellis` function. When used with a feedback polynomial, `poly2trellis` makes a feedback connection to the input of the trellis.

```
trellis = poly2trellis(5,[37 33],37)
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 16
          nextStates: [16x2 double]
             outputs: [16x2 double]
```

Generate random binary data. Convolutionally encode the data by using the specified trellis structure. Decode the coded data by using the Viterbi algorithm with the specified trellis structure, 34 for its traceback depth, truncated operation mode, and hard decisions.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
tbdepth = 34; % Traceback depth for Viterbi decoder
decodedData = vitdec(codedData,trellis,tbdepth,'trunc','hard');
```

Verify the decoded data has zero bit errors.

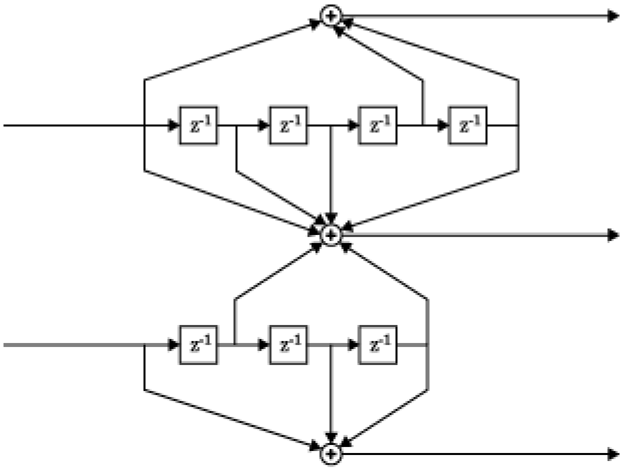
```
biterr(data,decodedData)
```

```
ans = 0
```

Compare Full Message to Piecewise Message Convolutional Encoding

Compare the convolutional encoding of a full message to the convolutional encoding of a message in two segments.

This diagram shows a rate 2/3 encoder with two input streams, three output streams, and seven shift registers.



Define the trellis structure in the diagram by using the `poly2trellis` function. Set the constraint length of the upper path to 5 and the constraint length of the lower path to 4. The octal representation of the code generator matrix corresponds to the taps from the upper and lower shift registers.

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

Inspect the coder configuration.

```
K = log2(trellis.numInputSymbols) % Number of input bit streams
```

```
K = 2
```

```
N = log2(trellis.numOutputSymbols) % Number of output bit streams
```

```
N = 3
```

```
coderate = K/N
```

```
coderate = 0.6667
```

```
numReg = log2(trellis.numStates) % Number of coder registers
```

```
numReg = 7
```

Define a message with five two-bit input symbols.

```
numSymPerFrame = 5; % Number of symbols per frame
msg = randi([0 1],K*numSymPerFrame,1);
```

Encode the full message by using the trellis to configure the `convenc` function.

```
[code_a,fstate_a] = convenc(msg,trellis);
```

Apply piecewise message encoding by using the same trellis structure. Use the final and initial state arguments when using the `convenc` function. For piecewise message encoding, message segments must be a multiple of the number of bits in an input symbol.

Encode part of the message, recording the final state for later use.

```
[code_a1,fstate_a1] = convenc(msg(1:6),trellis);
```

Encode the rest of the message, using the final state, `fstate_a1`, as the initial state input argument.

```
[code_a2,fstate_a2] = convenc(msg(7:end),trellis,fstate_a1);
```

Verify that the full coded message, `code_a`, matches the concatenated piecewise coded message, `[code_a1; code_a2]`.

```
isequal(code_a,[code_a1; code_a2])
```

```
ans = logical
   1
```

Verify that the final state, `fstate_a`, of the encoder after the full message encoding matches the final state, `fstate_a2`, of the encoder after piecewise message encoding.

```
isequal(fstate_a,fstate_a2)
```

```
ans = logical
   1
```

Input Arguments

`msg` — Binary message

vector of binary values

Binary message, specified as a vector of binary values. `msg` must contain one or more symbols. Each symbol must consist of log2(`trellis`.`numInputSymbols`) bits.

Example: `[1 1 0 1 0 0 1 1]` specifies the message as a binary row vector with eight elements.

Data Types: `double` | `logical`

`trellis` — Trellis description

structure

Trellis description, specified as a MATLAB structure that contains the trellis description for a rate *K*/*N* code. *K* represents the number of input bit streams, and *N* represents the number of output bit streams.

The trellis structure contains these fields. You can either use the `poly2trellis` function to create the trellis structure or create it manually. For more about this structure, see “Trellis Description of a Convolutional Code” and the `istrellis` function.

`numInputSymbols` — Number of symbols input to encoder

2^K

Number of symbols input to the encoder, specified as an integer equal to 2^K, where K is the number of input bit streams.

Data Types: `double`

`numOutputSymbols` — Number of symbols output from encoder
2^N

Number of symbols output from the encoder, specified as an integer equal to 2^N, where N is the number of output bit streams.

Data Types: `double`

`numStates` — Number of states in encoder
power of 2

Number of states in the encoder, specified as a power of 2.

Data Types: `double`

`nextStates` — Next states
matrix of integers

Next states for all combinations of current states and current inputs, specified as a matrix of integers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

`outputs` — Outputs
matrix of octal numbers

Outputs for all combinations of current states and current inputs, specified as a matrix of octal numbers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

Data Types: `struct`

`puncpat` — Puncture pattern
vector of binary values

Puncture pattern, specified as a vector of binary values. Indicate punctured bits with `0`s and unpunctured bits with `1`s. The length of the `puncpat` vector must be an integer divisor of the input message vector length, `length(msg)`.

Data Types: `double`

`istate` — Initial state
integer scalar

Initial state used for the encoder registers, specified as an integer scalar in the range [0, (`trellis.numStates` – 1)].

Data Types: `double`

Output Arguments

`codedout` — Convolutionally encoded message

vector of binary values

Convolutionally encoded message, returned as a vector of binary values. This output vector has the same data type and orientation as input `msg`. Each symbol in `codedout` consists of log2(`trellis`.`numOutputSymbols`) bits.

Data Types: `double` | `logical`

`fstate` — Final state

integer scalar

Final state of the encoder registers, returned as an integer scalar. When calling `convenc` iteratively, such as in a loop, `fstate` is typically used to set `istate` for subsequent calls to the `convenc` function.

Data Types: `double`

More About

Convolutional Coding

Convolutional coding is an error-control coding that has memory. Specifically, the computations and coded output depend on the current set of input symbols and on a number of previous input symbols that varies depending on the trellis configuration. A convolutional encoder accepts a fixed number of message symbols and produces a fixed number of code symbols.

Using a trellis structure that defines a set of generator polynomials, you can model nonsystematic, systematic feedforward, or systematic feedback convolutional codes. Examples on this page demonstrate various convolutional code architectures. For more information, see “Convolutional Codes”.

To decode the convolutionally coded output, you can use:

- The `vitdec` function or `comm.ViterbiDecoder` System object™ — Uses the Viterbi algorithm with hard-decision and soft-decision decoding
- The `comm.APPDecoder` System object — Uses an *a posteriori* probability decoder for the soft output decoding of convolutional codes

References

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

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

[3] Yasuda, Y., K. Kashiki, and Y. Hirata. “High-Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding.” *IEEE Transactions on Communications* 32, no. 3 (March 1984): 315–19. https://doi.org/10.1109/TCOM.1984.1096047.

[4] Haccoun, D., and G. Begin. “High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 37, no. 11 (November 1989): 1113–25. https://doi.org/10.1109/26.46505.

[5] Begin, G., D. Haccoun, and C. Paquin. “Further Results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 38, no. 11 (November 1990): 1922–28. https://doi.org/10.1109/26.61470.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The input arguments `trellis`, and `puncpat` must be compile-time constants. For more information, see `coder.Constant`.

See Also

Functions

`distspec` | `istrellis` | `poly2trellis` | `vitdec`

Objects

`comm.APPDecoder` | `comm.ConvolutionalEncoder` | `comm.TurboEncoder` | `comm.ViterbiDecoder`

Topics

“Convolutional Codes”
“Trellis Description of a Convolutional Code”
“Estimate BER for Hard and Soft Decision Viterbi Decoding”

Introduced before R2006a

convintrlv

Permute symbols using shift registers

Syntax

```
intrlved = convintrlv(data,nrows,slope)
[intrlved,state] = convintrlv(data,nrows,slope)
[intrlved,state] = convintrlv(data,nrows,slope,init_state)
```

Description

`intrlved = convintrlv(data,nrows,slope)` permutes the elements in `data` by using a set of `nrows` internal shift registers. The delay value of the kth shift register is `(k-1)*slope`, where k = 1, 2, 3,... `nrows`. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[intrlved,state] = convintrlv(data,nrows,slope)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[intrlved,state] = convintrlv(data,nrows,slope,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver.

Examples

The example below shows that `convintrlv` is a special case of the more general function `muxintrlv`. Both functions yield the same numerical results.

```
x = randi([0 1],100,1); % Original data
nrows = 5; % Use 5 shift registers
slope = 3; % Delays are 0, 3, 6, 9, and 12.
y = convintrlv(x,nrows,slope); % Interleaving using convintrlv.
delay = [0:3:12]; % Another way to express set of delays
y1 = muxintrlv(x,delay); % Interleave using muxintrlv.
isequal(y,y1)
```

The output below shows that `y`, obtained using `convintrlv`, and `y1`, obtained using `muxintrlv`, are the same.

```
ans =

     1
```

Another example using this function is in “Effect of Delays on Recovery of Convolutionally Interleaved Data Using MATLAB”.

The example on the `muxdeintrlv` reference page illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

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

See Also

`convdeintrlv` | `helintrlv` | `muxintrlv`

Topics

"Interleaving"

Introduced before R2006a

convmtx

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(randi([0 2^m-1],n,1),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` | `gf`

Topics

"Signal Processing Operations in Galois Fields"

Introduced before R2006a

cosets

Produce cyclotomic cosets for 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” on page 2-241.

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}'
c{2}'
c{3}'
```

The output is below.

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

Array elements =

     1



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

Array elements =

     2     4     6



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

Array elements =

     3     5     7
```

References

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

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

See Also

`gf` | `minpol`

Introduced before R2006a

crc.detector

(To be removed) Construct CRC detector object

Note will be removed in a future release. To detect errors in input data using cyclic redundancy check (CRC), use the `comm.CRCDetector` System object instead. For more details on the recommended workflow, see “Compatibility Considerations”.

Syntax

```
h= crc.detector(polynomial)
```

```
h= crc.detector(generatorObj)
```

```
h= crc.detector(‘Polynomial’, polynomial, ‘param1’, val1, etc.)
```

```
h= crc.detector
```

Description

`h= crc.detector(polynomial)` constructs a CRC detector object H defined by the generator polynomial POLYNOMIAL

`h= crc.detector(generatorObj)` constructs a CRC detector object H defined by the parameters found in the CRC generator object GENERATOROBJ

`h= crc.detector('property1', val1, ...)` constructs a CRC detector object H with properties as specified by PROPERTY/VALUE pairs.

`h= crc.detector` constructs a CRC detector object H with default properties. It constructs a CRC-CCITT detector, and is equivalent to:

```
h=
crc.detector('Polynomial','0x1021','InitialState','0xFFFF','ReflectInput',fal
se,'ReflectRemainder',false,'FinalXOR','0x0000')
```

Properties

The following table describes the properties of a CRC detector object. All properties are writable, except `Type`.

Property	Description
Type	Specifies the object as a 'CRC Detector'.

Property	Description
`Polynomial`	The generator polynomial that defines connections for a linear feedback shift register. This property can be specified as a binary vector representing descending powers of the polynomial. In this case, the leading '1' of the polynomial must be included. It can also be specified as a string, prefaced by '0x', that is a hexadecimal representation of the descending powers of the polynomial. In this case, the leading '1' of the polynomial is omitted.
`InitialState`	The initial contents of the shift register. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.
`ReflectInput`	A Boolean quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register.
`ReflectRemainder`	A Boolean quantity that specifies whether the binary output CRC checksum should be flipped around its center after the input data is completely through the shift register.
`FinalXOR`	The value with which the CRC checksum is to be XORed just prior to detecting the input data. This property can be specified as a binary scalar, a binary vector or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.

A detect method is used with the object to detect errors in digital transmission.

CRC Generation Algorithm

For information pertaining to the CRC generation algorithm, see “Cyclic Redundancy Check Codes”.

Detector Method

`[OUTDATA ERROR] = DETECT(H, INDATA)` detects transmission errors in the encoded input message `INDATA` by regenerating a CRC checksum using the CRC detector object H. The detector then compares the regenerated checksum with the checksum appended to INDATA. The binary-valued INDATA can be either a column vector or a matrix. If it is a matrix, each column is considered to be a separate channel. OUTDATA is identical to the input message INDATA, except that it has the CRC checksum stripped off. ERROR is a 1xC logical vector indicating if the encoded message INDATA has errors, where C is the number of channels in INDATA. An ERROR value of 0 indicates no errors, and a value of 1 indicates errors.

Examples

Create a CRC-16 CRC generator, then use it to generate a checksum for the binary vector represented by the ASCII sequence '123456789'. Introduce an error, then detect it using a CRC-16 CRC detector.

```
gen = crc.generator('Polynomial', '0x8005', 'ReflectInput', ...
true, 'ReflectRemainder', true);
det = crc.detector('Polynomial', '0x8005', 'ReflectInput', ...
true, 'ReflectRemainder', true);
% The message below is an ASCII representation
% of the digits 1-9
msg = reshape(de2bi(49:57, 8, 'left-msb')', 72, 1);
encoded = generate(gen, msg);
encoded(1) = ~encoded(1);                % Introduce an error
[outdata error] = detect(det, encoded);  % Detect the error
noErrors = isequal(msg, outdata)         % Should be 0
error                                    % Should be 1
```

This example generates the following output:

```
noErrors = 0 error = 1
```

Compatibility Considerations

`crc.detector` will be removed in a future release.
Warns starting in R2020b

crc.detector will be removed in a future release. To detect errors in input data using cyclic redundancy check (CRC), use the `comm.CRCDetector` System object instead.

Replace instances of crc.detector with a `comm.CRCDetector` System object. Note the mapping between crc.detector properties and `comm.CRCDetector` properties:

crc.detector	comm.CRCDetector
Polynomial	Polynomial
InitialState	InitialConditions
ReflectInput	ReflectInputBytes
ReflectRemainder	ReflectChecksums
FinalXOR	FinalXOR

See the following table for examples of migrating the old workflow to the recommended workflow.

Previous Workflow	Recommended Workflow

<table>
<tr>
<td><pre>
oldO = crc.generator;
oldOdet = crc.detector

data = randi([0 1],100,1);

enc_old = generate(oldO,data);

[dec_old, err_old] = detect(oldOdet,enc_old);

oldOdet =

                Type: CRC Detector
          Polynomial: 0x1021
        InitialState: 0xFFFF
        ReflectInput: false
    ReflectRemainder: false
            FinalXOR: 0x0000
</pre></td>
<td><pre>
sysO = comm.CRCGenerator('InitialConditions',1);
sysOdet = comm.CRCDetector('InitialConditions',1)

data = randi([0 1],100,1);

enc_new = sysO(data);

[dec_new, err] = sysOdet(enc_new);

sysOdet =

  comm.CRCDetector with properties:

           Polynomial: 'z^16 + z^12 + z^5 + 1'
    InitialConditions: 1
         DirectMethod: false
    ReflectInputBytes: false
     ReflectChecksums: false
             FinalXOR: 0
     ChecksumsPerFrame: 1
</pre></td>
</tr>
<tr>
<td><pre>
oldO = crc.generator([1 1 1 1 1]);
oldOdet = crc.detector([1 1 1 1 1])

data = randi([0 1],100,1);
enc_old = generate(oldO,data);
[dec_old, err_old] = detect(oldOdet,enc_old);

oldOdet =

                Type: CRC Detector
          Polynomial: 0xF
        InitialState: 0x0
        ReflectInput: false
    ReflectRemainder: false
            FinalXOR: 0x0
</pre></td>
<td><pre>
sysO = comm.CRCGenerator('Polynomial',[1 1 1 1 1]);
sysOdet = comm.CRCDetector('Polynomial',[1 1 1 1 1])

data = randi([0 1],100,1);
enc_new = sysO(data);
[dec_new, err] = sysOdet(enc_new);
sysOdet =

  comm.CRCDetector with properties:

           Polynomial: [1 1 1 1 1]
    InitialConditions: 0
         DirectMethod: false
    ReflectInputBytes: false
     ReflectChecksums: false
             FinalXOR: 0
    ChecksumsPerFrame: 1
</pre></td>
</tr>
</table>

See Also

Functions
crc.generator

Objects
comm.CRCDetector | comm.CRCGenerator

Blocks
General CRC Generator | General CRC Syndrome Detector

Introduced in R2008a

crc.generator

(To be removed) Construct CRC generator object

Note will be removed in a future release. To generate cyclic redundancy check (CRC) code bits, use the `comm.CRCGenerator` System object instead. For more details on the recommended workflow, see “Compatibility Considerations”.

Syntax

```
h = crc.generator(polynomial)
```

```
h = crc.generator(detectorObj)
```

```
h = crc.generator(‘Polynomial’, polynomial, ‘param1’, val1, etc.)
```

```
h = crc.generator
```

Description

`h = crc.generator(polynomial)` constructs a CRC generator object H defined by the generator polynomial POLYNOMIAL.

`h = crc.generator(detectorObj)` constructs a CRC generator object H defined by the parameters found in the CRC detector object DETECTOROBJ.

`h = crc.generator(‘property1', val1, ...)` constructs a CRC generator object H with properties as specified by the PROPERTY/VALUE pairs.

`h = crc.generator` constructs a CRC generator object H with default properties. It constructs a CRC-CCITT generator, and is equivalent to: h = crc.generator('Polynomial', '0x1021', 'InitialState', '0xFFFF', ...

'ReflectInput', false, 'ReflectRemainder', false, 'FinalXOR', '0x0000').

Properties

The following table describes the properties of a CRC generator object. All properties are writable, except `Polynomial`.

Property	Description
`Polynomial`	The generator polynomial that defines connections for a linear feedback shift register. This property can be specified as a binary vector representing descending powers of the polynomial. In this case, the leading '1' of the polynomial must be included. It can also be specified as a string, prefaced by '0x', that is a hexadecimal representation of the descending powers of the polynomial. In this case, the leading '1' of the polynomial is omitted.
`InitialState`	The initial contents of the shift register. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.
`ReflectInput`	A Boolean quantity that specifies whether the input data should be flipped on a bytewise basis prior to entering the shift register.
`ReflectRemainder`	A Boolean quantity that specifies whether the binary output CRC checksum should be flipped around its center after the input data is completely through the shift register.
`FinalXOR`	The value with which the CRC checksum is to be XORed just prior to being appended to the input data. This property can be specified as a binary scalar, a binary vector, or as a string, prefaced by '0x', that is a hexadecimal representation of the binary vector. As a binary vector, its length must be one less than the length of the binary vector representation of the Polynomial.

CRC Generation Algorithm

For information pertaining to the CRC generation algorithm, refer to the “CRC Non-Direct Algorithm” section of the Communications Toolbox User's Guide.

Generator Method

encoded = generate(h, msg) generates a CRC checksum for an input message using the CRC generator object H. It appends the checksum to the end of MSG. The binary-valued MSG can be either a column vector or a matrix. If it is a matrix, then each column is considered to be a separate channel.

Examples

Create a CRC-16 CRC generator, then use it to generate a checksum for the binary vector represented by the ASCII sequence '123456789'.

```
gen = crc.generator('Polynomial', '0x8005', ...
'ReflectInput', true, 'ReflectRemainder', true);
```

The message below is an ASCII representation of the digits 1-9.

```
msg = reshape(de2bi(49:57, 8, 'left-msb')', 72, 1);
encoded = generate(gen, msg);
```

```
h =

                Type: CRC Generator
          Polynomial: 0xF
        InitialState: 0xF
        ReflectInput: true
    ReflectRemainder: false
            FinalXOR: 0x0
```

Compatibility Considerations

`crc.generator` will be removed in a future release.
Warns starting in R2020b

crc.generator will be removed in a future release. To generate cyclic redundancy check (CRC) code, use the `comm.CRCGenerator` System object instead.

Replace instances of crc.generator with a `comm.CRCGenerator` System object. Note the mapping between crc.generator properties and `comm.CRCGenerator` properties:

crc.generator	comm.CRCGenerator
Polynomial	Polynomial
InitialState	InitialConditions
ReflectInput	ReflectInputBytes
ReflectRemainder	Reflectchecksums
FinalXOR	FinalXOR

See the following table for examples of migrating the old workflow to the recommended workflow.

Previous Workflow	Recommended Workflow

```
data = randi([0 1],100,1);
oldO = crc.generator
encOld = generate(oldO,data);

oldO =

                Type: CRC Generator
          Polynomial: 0x1021
        InitialState: 0xFFFF
        ReflectInput: false
    ReflectRemainder: false
            FinalXOR: 0x0000
```

```
data = randi([0 1],100,1);

sysO = comm.CRCGenerator('InitialConditions',1)
encNew = sysO(data);

sysO =

  comm.CRCGenerator with properties:

           Polynomial: 'z^16 + z^12 + z^5 + 1'
    InitialConditions: 1
         DirectMethod: false
    ReflectInputBytes: false
     ReflectChecksums: false
             FinalXOR: 0
    ChecksumsPerFrame: 1
```

```
data = randi([0 1],96,1);

oldO = crc.generator('Polynomial', '0xF', 'InitialState', '0xF',...
'ReflectInput', true, 'FinalXOR', '0x0')

encOld = generate(oldO,data);

 Type: CRC Generator
          Polynomial: 0xF
        InitialState: 0xF
        ReflectInput: true
    ReflectRemainder: false
            FinalXOR: 0x0
```

```
data = randi([0 1],96,1);

sysO = comm.CRCGenerator('Polynomial',...
'0x1F','InitialConditions', [1 1 1 1],...
'ReflectInputBytes', true, 'FinalXOR', 0)

encNew = sysO(data);

sysO =

  comm.CRCGenerator with properties:

           Polynomial: '0x1F'
    InitialConditions: [1 1 1 1]
         DirectMethod: false
    ReflectInputBytes: true
     ReflectChecksums: false
             FinalXOR: 0
    ChecksumsPerFrame: 1
```

```
data = randi([0 1],100,5);

oldO = crc.generator([1 1 1 1 1])

encOld = generate(oldO,data);

oldO =

                Type: CRC Generator
          Polynomial: 0xF
        InitialState: 0x0
        ReflectInput: false
    ReflectRemainder: false
            FinalXOR: 0x0
```

```
data = randi([0 1],100,5);

sysO = comm.CRCGenerator('Polynomial',[1 1 1 1 1],...
 'ChecksumsPerFrame',5)

encNew = sysO(data(:));

sysO =

  comm.CRCGenerator with properties:

           Polynomial: [1 1 1 1 1]
    InitialConditions: 0
         DirectMethod: false
    ReflectInputBytes: false
     ReflectChecksums: false
             FinalXOR: 0
    ChecksumsPerFrame: 5
```

See Also

Functions
`crc.detector`

Objects
`comm.CRCDetector` | `comm.CRCGenerator`

Blocks
General CRC Generator | General CRC Syndrome Detector

Introduced in R2008a

cyclgen

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. Alternatively, you can specify `pol` as a polynomial character vector. For more information, see “Character Representation of Polynomials”.

`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

Parity Check and Generator Matrices for Binary Cyclic Codes

Create parity check and generator matrices for a binary cyclic code having codeword length 7 and message length 4.

Create the generator polynomial using `cyclpoly`.

```
pol = cyclpoly(7,4);
```

Create the parity check and generator matrices. The parity check matrix `parmat` has a 3-by-3 identity matrix embedded in its leftmost columns.

```
[parmat,genmat,k] = cyclgen(7,pol)
```

```
parmat = 3×7
```

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

```
genmat = 4×7

     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
```

Create a parity check matrix in which the code is not systematic. The matrix `parmatn` does not have an embedded 3-by-3 identity matrix.

```
parmatn = cyclgen(7,pol,'nonsys')
```

```
parmatn = 3×7

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

Create the parity check and generator matrices for a (7,3) binary cyclic code. As this is a systematic code, there is a 4-by-4 identity matrix in the leftmost columns of `parmat2`.

```
parmat2 = cyclgen(7,'1 + x^2 + x^3 + x^4')
```

```
parmat2 = 4×7

     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
```

See Also

`bchgenpoly` | `cyclpoly` | `decode` | `encode`

Topics

“Block Codes”

Introduced before R2006a

cyclpoly

Produce generator polynomials for 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	Row vector representing the polynomial
`'max'`	One generator polynomial having the greatest possible weight	Row vector representing the polynomial
`'all'`	All generator polynomials	Matrix, each row of which represents one such polynomial
a positive integer, *L*	All generator polynomials having weight *L*	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, the output `pol` is empty and a warning message is displayed.

Examples

Cyclic Code Generator Polynomials

Create [15,4] cyclic code generator polynomials.

Use the input `'all'` to show all possible generator polynomials for a [15,4] cyclic code. Use the input `'max'` to show that $1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^{11}$ is one such polynomial that has the largest number of nonzero terms.

```
c1 = cyclpoly(15,4,'all')
```

```
c1 = 3×12
```

```
     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 = cyclpoly(15,4,'max')
```

```
c2 = 1×12

     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)
```

```
Warning: No cyclic generator polynomial satisfies the given constraints.

c3 =

     []
```

Algorithms

If *opt* is `'min'`, `'max'`, or omitted, 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`

Topics

“Block Codes”

Introduced before R2006a

de2bi

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, the output `b` is a matrix in which each row is the binary form of the corresponding element in `d`.

`b = de2bi(d,n)` has an output with `n` columns.

`b = de2bi(d,n,p)` converts a nonnegative decimal integer `d` to a base-`p` row vector.

`b = de2bi(d,[],p)` specifies the base, `p` .

`b = de2bi(d,___,flg)` uses `flg` to determine whether the first column of `b` contains the lowest-order or highest-order digits.

Examples

Convert Decimals to Binary Numbers

Convert decimals 1 through 10 into their equivalent binary representations.

```
d = (1:10)';
b = de2bi(d);
[d b]
```

```
ans = 10×5

     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
```

Convert 3 and 9 to binary numbers. Each value is represented by a four-element row.

```
b = de2bi([3 9])
```

```
b = 2×4

     1     1     0     0
     1     0     0     1
```

Repeat the conversion with the number of columns set to 5. The output is now padded with zeros in the fifth column.

```
bb = de2bi([3 9],5)
```

```
bb = 2×5

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

Convert Decimals to Base-3 Numbers

Convert the decimals 1 through 6 to their base-3 equivalents. Set the leftmost bit as the most significant digit.

```
d = (1:6)';
t = de2bi(d,[],3,'left-msb');
[d t]
```

```
ans = 6×3

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

Convert Decimal to Binary

This example shows how to convert decimals to binary numbers in their base-2 equivalents.

```
d_array = [1 2 3 4];
```

Convert the decimal array to binary by using the `de2bi` function. Specify that the most significant digit is the leftmost element and set the number of desired columns to 5. The output becomes a 4-by-5 matrix where each row corresponds to a decimal value from the input. Because the largest decimal value in `d_array` can be expressed in 3 columns, the `de2bi` pads the matrix with two extra zero columns at the specified most-significant bit side. If you specify too few columns, the conversion will fail.

```
b_array = de2bi(d_array,5,'left-msb')
b_array = 4×5

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

```
b_array = de2bi(d_array,5,'right-msb')
```

```
b_array = 4×5

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

If you do not specify a number of columns, the number of columns is exactly what is needed to express the largest decimal of the input.

```
b_array = de2bi(d_array,'left-msb')
```

```
b_array = 4×3

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

The output rows for specifying a leftmost-significant bit correspond to:

$$1 = 0(2^2) + 0(2^1) + 1(2^0)$$

$$2 = 0(2^2) + 1(2^1) + 0(2^0)$$

$$3 = 0(2^2) + 1(2^1) + 1(2^0)$$

$$4 = 1(2^2) + 0(2^1) + 0(2^0)$$

```
b_array = de2bi(d_array,'right-msb')
```

```
b_array = 4×3

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

The output rows for specifying a rightmost-significant bit correspond to:

$$1 = 1(2^0) + 0(2^1) + 0(2^2)$$

$$2 = 0(2^0) + 1(2^1) + 0(2^2)$$

$3 = 1(2^0) + 1(2^1) + 0(2^2)$

$4 = 0(2^0) + 0(2^1) + 1(2^2)$

Input Arguments

d — Decimal input
nonnegative integer | vector | matrix

Decimal input, specified as a nonnegative integer, vector, or matrix. If d is a matrix, it is treated like the column vector `d(:)`.

Note To ensure an accurate conversion, d must be less than or equal to 2^{52}.

Data Types: `double` | `single` | `integer` | `fi`

n — Number of output columns
positive integer scalar

The number of output columns specified as a positive scalar. If necessary, the binary representation of d is padded with extra zeros.

Data Types: `double` | `single`

p — Base
2 (default) | positive integer scalar

Base of the output b, specified as an integer greater than or equal to 2.

- If d is a vector, the output b is a matrix in which each row is the base-p form of the corresponding element in d.
- If d is a matrix, `de2bi` treats it like the vector `d(:)`.

Data Types: `double` | `single`

`flg` — MSB flag
`'right-msb'` (default) | `'left-msb'`

MSB flag, specified as `'right-msb'` or `'left-msb'`.

- `'right-msb'` –- Indicates the right (or last) column of the binary output, b, as the most significant bit (or highest-order digit).
- `'left-msb'` –- Indicates the left (or first) column of the binary output, b, as the most significant bit (or highest-order digit).

Data Types: `char` | `string`

Output Arguments

b — Binary output
vector | matrix

Binary representation of d, returned as a row vector or matrix. The output is of the same data type as the input.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

bi2de

Introduced before R2006a

decode

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)
[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 encode reference page 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 polynomial character vector or 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 polynomial character vector or 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)` 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, `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` 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, a nonnegative integer in the rth row of `vec2matcerr` 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, `cerr = err`.

Examples

Encode and Decode Message with Hamming Code

Set the values of the codeword length and message length.

```
n = 15; % Codeword length
k = 11; % Message length
```

Create a random binary message with length equal to the message length.

```
data = randi([0 1],k,1);
```

Encode the message.

```
encData = encode(data,n,k,'hamming/binary');
```

Corrupt the encoded message sequence by introducing an error in the fourth bit.

```
encData(4) = ~encData(4);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'hamming/binary');
numerr = biterr(data,decData)
```

```
numerr = 0
```

Algorithms

Depending on the decoding method, decode relies on such lower-level functions as hammgen, syndtable, and cyclgen.

See Also

cyclpoly | encode | gen2par | syndtable

Topics

"Block Codes"

Introduced before R2006a

deintrlv

Restore ordering of symbols

Syntax

```
deintrlvd = deintrlv(data,elements)
```

Description

`deintrlvd = deintrlv(data,elements)` restores the original ordering of the elements of `data` by acting as an inverse of `intrlv`. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. To use this function as an inverse of the `intrlv` function, use the same `elements` input in both functions. In that case, the two functions are inverses in the sense that applying `intrlv` followed by `deintrlv` leaves data unchanged.

Examples

The code below illustrates the inverse relationship between `intrlv` and `deintrlv`.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p); % Rearrange [10 20 30 ... 100].
b = deintrlv(a,p) % Deinterleave a to restore ordering.
```

The output is

```
b =

    10    20    30    40    50    60    70    80    90   100
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`intrlv`

Topics

“Interleaving”

Introduced before R2006a

dfe

(To be removed) Construct decision-feedback equalizer object

Note will be removed in a future release. Use `comm.DecisionFeedback` instead.

Syntax

```
eqobj = dfe(nfwdweights,nfbkweights,alg)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst)
eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst,nsamp)
```

Description

The `dfe` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`eqobj = dfe(nfwdweights,nfbkweights,alg)` constructs a decision feedback equalizer object. The equalizer's feedforward and feedback filters have `nfwdweights` and `nfbkweights` symbol-spaced complex weights, respectively, which are initially all zeros. `alg` describes the adaptive algorithm that the equalizer uses; you should create `alg` using any of these functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`. The signal constellation of the desired output is `[-1 1]`, which corresponds to binary phase shift keying (BPSK).

`eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst)` specifies the signal constellation vector of the desired output.

`eqobj = dfe(nfwdweights,nfbkweights,alg,sigconst,nsamp)` constructs a DFE with a fractionally spaced forward filter. The forward filter has `nfwdweights` complex weights spaced at `T/nsamp`, where `T` is the symbol period and `nsamp` is a positive integer. `nsamp = 1` corresponds to a symbol-spaced forward filter.

Properties

The table below describes the properties of the decision feedback equalizer object. To learn how to view or change the values of a decision feedback equalizer object, see “Equalization”.

Note To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
`EqType`	Fixed value, `'Decision Feedback Equalizer'`
`AlgType`	Name of the adaptive algorithm represented by `alg`

Property	Description
nWeights	Number of weights in the forward filter and the feedback filter, in the format [nfwdweights, nfbkweights]. The number of weights in the forward filter must be at least 1.
nSampPerSym	Number of input samples per symbol (equivalent to nsamp input argument). This value relates to both the equalizer structure (see the use of K in “Equalization”) and an assumption about the signal to be equalized.
RefTap (except for CMA equalizers)	Reference tap index, between 1 and nfwdweights. Setting this to a value greater than 1 effectively delays the reference signal with respect to the equalizer's input signal.
SigConst	Signal constellation, a vector whose length is typically a power of 2.
Weights	Vector that concatenates the complex coefficients from the forward filter and the feedback filter. This is the set of w_i values in the schematic in “Equalization”.
WeightInputs	Vector that concatenates the tap weight inputs for the forward filter and the feedback filter. This is the set of u_i values in the schematic in “Equalization”.
ResetBeforeFiltering	If 1, each call to equalize resets the state of eqobj before equalizing. If 0, the equalization process maintains continuity from one call to the next.
NumSamplesProcessed	Number of samples the equalizer processed since the last reset. When you create or reset eqobj, this property value is 0.
Properties specific to the adaptive algorithm represented by alg	See reference page for the adaptive algorithm function that created alg: lms, signlms, normlms, varlms, rls, or cma.

Relationships Among Properties

If you change nWeights, MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

Property	Adjusted Value
Weights	zeros(1,sum(nWeights))
WeightInputs	zeros(1,sum(nWeights))
StepSize (Variable-step-size LMS equalizers)	InitStep*ones(1,sum(nWeights))
InvCorrMatrix (RLS equalizers)	InvCorrInit*eye(sum(nWeights))

Examples

Configuring Decision Feedback Equalizers

This example configures the recommended `comm.DecisionFeedback` System object™ and the legacy `dfe` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Decision Feedback Equalizer

Configure `dfe` and `comm.DecisionFeedback` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.DecisionFeedback` System object™ assumes that leakage factor is always 1.

```
eqOld = dfe(5,3,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Decision Feedback Equalizer'
  AlgType: 'LMS'
  nWeights: [5 3]
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.DecisionFeedback('NumForwardTaps',5,'NumFeedbackTaps',3, ...
    'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.DecisionFeedback with properties:
  Algorithm: 'LMS'
  NumForwardTaps: 5
  NumFeedbackTaps: 3
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r,x(1:100));
yNew = eqNew(r,x(1:100));
```

Use RLS Algorithm with Decision Feedback Equalizer

Configure `dfe` and `comm.DecisionFeedback` objects with comparable settings.

```
eqOld = dfe(5,3,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Decision Feedback  Equalizer'
  AlgType: 'RLS'
  nWeights: [5 3]
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  Forget Factor: 0.9500
  InvCorrInit: 0.1000
  InvCorrMatrix: [8×8 double]
  Weights: [0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.DecisionFeedback('NumForwardTaps',5,'NumFeedbackTaps',3,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.DecisionFeedback with properties:
  Algorithm: 'RLS'
  NumForwardTaps: 5
  NumFeedbackTaps: 3
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: 0.1000
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.DecisionFeedback` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Configure `dfe` and `comm.DecisionFeedback` objects with comparable settings. For the `comm.DecisionFeedback` object, set the initial inverse correlation matrix to `eye(5)*0.2`.

```
eqOld = dfe(5,3,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Decision Feedback Equalizer'
  AlgType: 'RLS'
  nWeights: [5 3]
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ForgetFactor: 0.9500
  InvCorrInit: 0.1000
  InvCorrMatrix: [8×8 double]
  Weights: [0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.DecisionFeedback('NumForwardTaps',5,'NumFeedbackTaps',3,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1, ...
    'InitialInverseCorrelationMatrix',eye(5)*0.2)
```

```
eqNew = comm.DecisionFeedback with properties:
  Algorithm: 'RLS'
  NumForwardTaps: 5
  NumFeedbackTaps: 3
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: [8×8 double]
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.DecisionFeedback` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Use Decision Feedback Equalizers Considering Signal Delays

Configure `dfe` and `comm.DecisionFeedback` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `dfe` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = dfe(filterDelay*sps+4,6,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Decision Feedback Equalizer'
  AlgType: 'LMS'
  nWeights: [16 6]
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.DecisionFeedback('NumForwardTaps',filterDelay*sps+4, ...
    'NumFeedbackTaps',6,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.DecisionFeedback with properties:
  Algorithm: 'LMS'
  NumForwardTaps: 16
  NumFeedbackTaps: 6
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.DecisionFeedback` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.DecisionFeedback` object, `InputDelay` is used to synchronize with the delayed signal. `NumForwardTaps`, `NumFeedbackTaps`, and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.DecisionFeedback('NumForwardTaps',11,'NumFeedbackTaps',4, ...
    'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.DecisionFeedback with properties:
  Algorithm: 'LMS'
  NumForwardTaps: 11
  NumFeedbackTaps: 4
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Compatibility Considerations

`dfe` will be removed

Warns starting in R2020a

dfe will be removed in a future release. Use `comm.DecisionFeedback` instead. For examples comparing setup of `comm.DecisionFeedback` to `dfe`, see “Configuring Decision Feedback Equalizers” on page 2-266.

See Also

Objects

`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics

“Equalization”

Introduced before R2006a

dftmtx

Discrete Fourier transform matrix in 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(randi([0 2^m-1],n,1),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.

Algorithms

The element `dm(a,b)` equals `alph^((a-1)*(b-1))`.

See Also

`fft` | `gf` | `ifft`

Topics

"Signal Processing Operations in Galois Fields"

Introduced before R2006a

distspec

Compute distance spectrum of convolutional code

Syntax

```
spect = distspec(trellis,n)
spect = distspec(trellis)
```

Description

`spect = distspec(trellis,n)` computes the free distance and the first `n` components of the weight and distance spectra of a linear convolutional code. Because convolutional codes do not have block boundaries, the weight spectrum and distance spectrum are semi-infinite and are most often approximated by the first few components. The input `trellis` is a valid MATLAB trellis structure, as described in “Trellis Description of a Convolutional Code”. The output, `spect`, is a structure with these fields:

Field	Meaning
`spect.dfree`	Free distance of the code. This is the minimum number of errors in the encoded sequence required to create an error event.
`spect.weight`	A length-`n` vector that lists the total number of information bit errors in the error events enumerated in `spect.event`.
`spect.event`	A length-`n` vector that lists the number of error events for each distance between `spect.dfree` and `spect.dfree+n-1`. The vector represents the first `n` components of the distance spectrum.

`spect = distspec(trellis)` is the same as `spect = distspec(trellis,1)`.

Examples

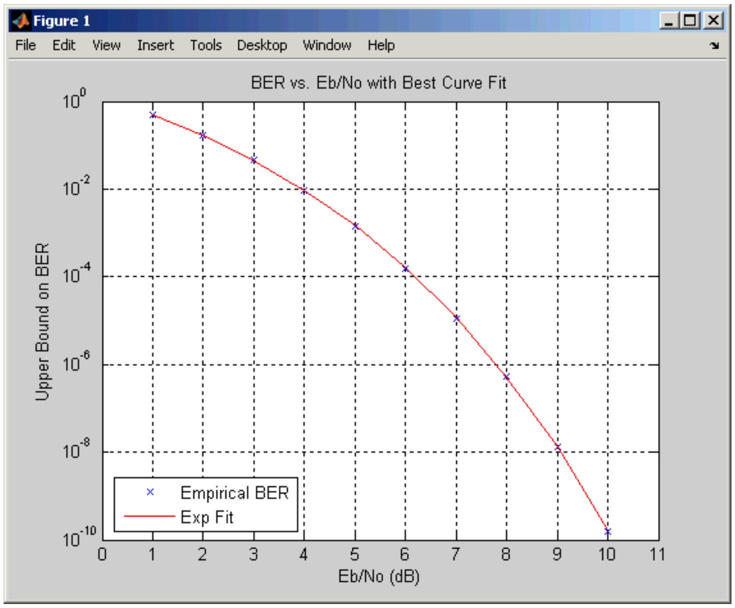
The example below performs these tasks:

- Computes the distance spectrum for the rate 2/3 convolutional code that is depicted on the reference page for the `poly2trellis` function
- Uses the output of `distspec` as an input to the `bercoding` function, to find a theoretical upper bound on the bit error rate for a system that uses this code with coherent BPSK modulation
- Plots the upper bound using the `berfit` function

```
trellis = poly2trellis([5 4],[23 35 0; 0 5 13])
spect = distspec(trellis,4)
berub = bercoding(1:10,'conv','hard',2/3,spect); % BER bound
berfit(1:10,berub); ylabel('Upper Bound on BER'); % Plot.
```

The output and plot are below.

```
trellis =

     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]


spect =

     dfree: 5
    weight: [1 6 28 142]
     event: [1 2 8 25]
```



Algorithms

The function uses a tree search algorithm implemented with a stack, as described in [2].

References

[1] Bocharova, I. E., and B. D. Kudryashov, “Rational Rate Punctured Convolutional Codes for Soft-Decision Viterbi Decoding,” *IEEE Transactions on Information Theory*, Vol. 43, No. 4, July 1997, pp. 1305–1313.

[2] Cedervall, M., and R. Johannesson, “A Fast Algorithm for Computing Distance Spectrum of Convolutional Codes,” *IEEE Transactions on Information Theory*, Vol. 35, No. 6, Nov. 1989, pp. 1146–1159.

[3] Chang, J., D. Hwang, and M. Lin, “Some Extended Results on the Search for Good Convolutional Codes,” *IEEE Transactions on Information Theory*, Vol. 43, No. 5, Sep. 1997, pp. 1682–1697.

[4] Frenger, P., P. Orten, and T. Ottosson, "Comments and Additions to Recent Papers on New Convolutional Codes," *IEEE Transactions on Information Theory*, Vol. 47, No. 3, March 2001, pp. 1199–1201.

See Also

`bercoding` | `iscatastrophic` | `istrellis` | `poly2trellis`

Introduced before R2006a

doppler

Construct Doppler spectrum structure

Syntax

```
s = doppler(specType)
s = doppler(specType, fieldValue)
s = doppler('BiGaussian', Name,Value)
```

Description

`s = doppler(specType)` constructs a Doppler spectrum structure of type `specType` for use with a fading channel System object. The returned structure, `s`, has default values for its dependent fields.

`s = doppler(specType, fieldValue)` constructs a Doppler spectrum structure of type `specType` for use with a fading channel System object. The returned structure, `s`, has its dependent field specified to `fieldValue`.

`s = doppler('BiGaussian', Name,Value)` constructs a BiGaussian Doppler spectrum structure for use with a fading channel System object. The returned structure, `s`, has dependent fields specified by `Name,Value` pair arguments.

Examples

Construct a Flat Doppler Spectrum Structure

Construct a flat Doppler structure variable for use with channel objects such as `comm.RayleighChannel`.

Invoke the `doppler` function to create a flat Doppler structure variable.

```
s = doppler('Flat')
```

```
s = struct with fields:
    SpectrumType: 'Flat'
```

Create a Bell Doppler Structure Variable

Use the `doppler` function to create a Doppler structure variable having the Bell spectrum.

```
s = doppler('Bell')
```

```
s = struct with fields:
    SpectrumType: 'Bell'
     Coefficient: 9
```

Construct a Rounded Doppler Spectrum Structure with Specified Polynomial

Specify the coefficients of the Doppler spectrum structure variable.

Construct a Rounded Doppler spectrum structure with coefficients `a0`, `a2`, and `a4` set to `2`, `6`, and `1`, respectively.

```
s = doppler('Rounded', [2, 6, 1])
```

```
s = struct with fields:
    SpectrumType: 'Rounded'
      Polynomial: [2 6 1]
```

Construct a BiGaussian Doppler Spectrum Structure with Specified Field Values

Use the `doppler` function to create a Doppler spectrum structure with the parameters specified for a BiGaussian spectrum.

```
s = doppler('BiGaussian','NormalizedCenterFrequencies', ...
    [.1 .85],'PowerGains',[1 2])
```

```
s = struct with fields:
                    SpectrumType: 'BiGaussian'
    NormalizedStandardDeviations: [0.7071 0.7071]
     NormalizedCenterFrequencies: [0.1000 0.8500]
                      PowerGains: [1 2]
```

The `NormalizedStandardDeviations` field is set to the default value. The `NormalizedCenterFrequencies`, and `PowerGains` fields are set to the values specified from the input arguments.

Input Arguments

`specType` — Spectrum type of Doppler spectrum structure for use with fading channel System object

`'Jakes'` | `'Flat'` | `'Rounded'` | `'Bell'` | `'Asymmetric Jakes'` | `'Restricted Jakes'` | `'Gaussian'` | `'BiGaussian'`

The spectrum type of a Doppler spectrum structure for use with a fading channel System object. Specify this value as a character vector.

The analytical expression for each Doppler spectrum type is described in the “Algorithms” on page 2-278 section.

Data Types: `char`

`fieldValue` — Value of dependent field of Doppler spectrum structure

scalar | vector

The value of the dependent field of the Doppler spectrum structure, specified as a scalar or vector of built-in data type. If you do not specify `fieldValue` , the dependent fields of the spectrum type use the default values.

Spectrum Type	Dependent Field	Description	Default Value
“Jakes” on page 2-279	—	—	—
“Flat” on page 2-279	—	—	—
“Rounded” on page 2-279	`Polynomial`	1-by-3 vector of real finite values, representing the polynomial coefficients, `a0, a2` and `a4`	`[1 -1.72 0.785]`
“Bell” on page 2-279	`Coefficient`	Nonnegative, finite, real scalar representing the Bell spectrum coefficient	`9`
“Asymmetric Jakes” on page 2-279	`NormalizedFrequencyInterval`	1-by-2 vector of real values between –1 and 1, inclusive, representing the minimum and maximum normalized Doppler shifts	`[0 1]`
“Restricted Jakes” on page 2-280	`NormalizedFrequencyInterval`	1-by-2 vector of real values between 0 and 1, inclusive, representing the minimum and maximum normalized Doppler shifts	`[0 1]`
“Gaussian” on page 2-280	`NormalizedStandardDeviation`	Normalized standard deviation of the Gaussian Doppler spectrum, specified as a positive, finite, real scalar	`0.7071`
“BiGaussian” on page 2-280	`NormalizedStandardDeviations`	Normalized standard deviations of the BiGaussian Doppler spectrum, specified as a positive, finite, real 1-by-2 vector	`[0.7071 0.7071]`
	`NormalizedCenterFreqencies`	Normalized center frequencies of the BiGaussian Doppler spectrum specified as a real 1-by-2 vector whose elements fall between –1 and 1	`[0 0]`

Spectrum Type	Dependent Field	Description	Default Value
	`PowerGains`	Linear power gains of the BiGaussian Doppler spectrum specified as a real nonnegative 1-by-2 vector	`[0.5 0.5]`

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `s=doppler('BiGaussian', 'NormalizedStandardDeviations', [.8 .75], 'NormalizedCenterFrequencies', [-.8 0], 'PowerGains', [.6 .6])`

`NormalizedStandardDeviations` — Normalized standard deviations of first and second Gaussian functions
`[1/sqrt(2) 1/sqrt(2)]` (default) | 1-by-2 positive numeric vector

The normalized standard deviation of the first and second Gaussian functions. You can specify this value as a 1-by-2 positive numeric vector, of built-in data types.

When you do not specify this dependent field, the default value is `[1/sqrt(2) 1/sqrt(2)]`.

Data Types: `double`

`NormalizedCenterFrequencies` — Normalized center frequencies of first and second Gaussian functions
`[0 0]` (default) | 1-by-2 numeric vector

The normalized center frequencies of the first and second Gaussian functions. You can specify this value as a 1-by-2 numeric vector of real values between –1 and 1, of built-in data types.

When you do not specify this dependent field, the default value is `[0 0]`.

Data Types: `double`

`PowerGains` — Power gains of first and second Gaussian functions
`[0.5 0.5]` (default) | 1-by-2 numeric vector

The power gains of the first and second Gaussian functions. You can specify this value as a 1-by-2 nonnegative numeric vector of built-in data types.

When you do not specify this dependent field, the default value is `[0.5 0.5]`.

Data Types: `double`

Algorithms

The following algorithms represent the analytical expressions for each Doppler spectrum type. In each case, f_d denotes the maximum Doppler shift (`MaximumDopplerShift` property) of the associated fading channel System object.

Jakes

The theoretical `Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{1}{\pi f_d \sqrt{1 - (f/f_d)^2}}, \ |f| \le f_d$$

Flat

The theoretical `Flat` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{1}{2f_d}, |f| \le f_d$$

Rounded

The theoretical `Rounded` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = C_r \left[a_0 + a_2 \left(\frac{f}{f_d} \right)^2 + a_4 \left(\frac{f}{f_d} \right)^4 \right], |f| \le f_d$$

where

$$C_r = \frac{1}{2f_d \left[a_0 + \frac{a_2}{3} + \frac{a_4}{5} \right]}$$

and you can specify $[a_0, \ a_2, \ a_4]$ in the dependent field, `polynomial`.

Bell

The theoretical `Bell` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{C_b}{1 + A \left(\frac{f}{f_d} \right)^2}$$

$$|f| \le f_d$$

where

$$C_b = \frac{\sqrt{A}}{\pi f_d}$$

You can specify *A* in the dependent field, `coefficient`.

Asymmetric Jakes

The theoretical `Asymmetric Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{A_a}{\pi f_d \sqrt{1 - (f/f_d)^2}}, \quad -f_d \le f_{\min} \le f \le f_{\max} \le f_d$$

$$A_a = \frac{1}{\frac{1}{\pi} \left[\sin^{-1} \left(\frac{f_{\max}}{f_d} \right) - \sin^{-1} \left(\frac{f_{\min}}{f_d} \right) \right]}$$

where you can specify $f_{\min}/\ f_d$ and$f_{\max}\ /f_d$ in the dependent field, `NormalizedFrequencyInterval`.

Restricted Jakes

The theoretical `Restricted Jakes` Doppler spectrum, *S(f)* has the analytic formula

$$S(f) = \frac{A_r}{\pi f_d \sqrt{1 - (f/f_d)^2}},\ 0 \le f_{\min} \le |f| \le f_{\max} \le f_d$$

where

$$A_r = \frac{1}{\frac{2}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

where you can specify $f_{\min}/\ f_d$ and$f_{\max}\ /f_d$ in the dependent field, `NormalizedFrequencyInterval`.

Gaussian

The theoretical `Gaussian` Doppler spectrum, *S(f)* has the analytic formula

$$S_G(f) = \frac{1}{\sqrt{2\pi\sigma_G^2}}\exp\left(-\frac{f^2}{2\sigma_G^2}\right)$$

You can specify σ_G/f_d in the dependent field, `NormalizedStandardDeviation`.

BiGaussian

The theoretical `BiGaussian` Doppler spectrum, *S(f)* has the analytic formula

$$S_G(f) = A_G\left[\frac{C_{G1}}{\sqrt{2\pi\sigma_{G1}^2}}\exp\left(-\frac{(f - f_{G1})^2}{2\sigma_{G1}^2}\right) + \frac{C_{G2}}{\sqrt{2\pi\sigma_{G2}^2}}\exp\left(-\frac{(f - f_{G2})^2}{2\sigma_{G2}^2}\right)\right]$$

where $A_G = \dfrac{1}{C_{G1} + C_{G2}}$ is a normalization coefficient.

You can specify σ_{G1}/f_d and σ_{G2}/f_d in the `NormalizedStandardDeviations` dependent field.

You can specify f_{G1}/f_d and f_{G2}/f_d in the `NormalizedCenterFrequencies` dependent field.

C_{G1} and C_{G2} are power gains that you can specify in the `PowerGains` dependent field.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

MIMO Channel | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2007a

doppler.ajakes

(Removed) Construct asymmetrical Doppler spectrum object

Note `doppler.ajakes` has been removed. Use `doppler('Asymmetric Jakes',...)` instead.

Syntax

```
dop = doppler.ajakes(freqminmaxajakes)
dop = doppler.ajakes
```

Description

The `doppler.ajakes` function creates an asymmetrical Jakes (AJakes) Doppler spectrum object. This object is to be used for the `DopplerSpectrum` property of a channel object created with the `rayleighchan` or the `ricianchan` functions.

`dop = doppler.ajakes(freqminmaxajakes)`, where `freqminmaxajakes` is a row vector of two finite real numbers between -1 and 1, creates a Jakes Doppler spectrum that is nonzero only for normalized (by the maximum Doppler shift f_d, in Hz) frequencies f_{norm} such that $-1 \leq f_{\min,\, norm} \leq f_{norm} \leq f_{\max,\, norm} \leq 1$, where $f_{\min,\, norm}$ is given by `freqminmaxajakes(1)` and $f_{\max,\, norm}$ is given by `freqminmaxajakes(2)`. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically: $f_{\min,\, norm} = f_{\min}/f_d$ and $f_{\max,\, norm} = f_{\max}/f_d$, where $f_{\min}$ is the minimum Doppler shift (in hertz) and $f_{\max}$ is the maximum Doppler shift (in hertz).

When `dop` is used as the `DopplerSpectrum` property of a channel object, space `freqminmaxajakes(1)` and `freqminmaxajakes(2)` by more than 1/50. Assigning a smaller spacing results in `freqminmaxarjakes` being reset to the default value of `[0 1]`.

`dop = doppler.ajakes` creates an asymmetrical Doppler spectrum object with a default `freqminmaxajakes = [0 1]`. This syntax is equivalent to constructing a Jakes Doppler spectrum that is nonzero only for positive frequencies.

Properties

The AJakes Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'AJakes'`
`FreqMinMaxAJakes`	Vector of minimum and maximum normalized Doppler shifts, two real finite numbers between -1 and 1

Theory and Applications

The Jakes power spectrum is based on the assumption that the angles of arrival at the mobile receiver are uniformly distributed [1]: the spectrum then covers the frequency range from $-f_d$ to f_d, f_d being the maximum Doppler shift. When the angles of arrival are not uniformly distributed, then the Jakes power spectrum does not cover the full Doppler bandwidth from $-f_d$ to f_d. The AJakes Doppler spectrum object covers the case of a power spectrum that is nonzero only for frequencies f such that $-f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$. It is an asymmetrical spectrum in the general case, but becomes a symmetrical spectrum if $f_{\min} = -f_{\max}$.

The normalized AJakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{A_a}{\pi f_d \sqrt{1-(f/f_d)^2}}, \quad -f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$$

$$A_a = \frac{1}{\frac{1}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

where $f_{\min}$ and $f_{\max}$ denote the minimum and maximum frequencies where the spectrum is nonzero. You can determine these values from the probability density function of the angles of arrival.

Examples

The following MATLAB code first creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$ Hz. It then creates an AJakes Doppler object with minimum normalized Doppler shift $f_{\min,\,norm} = -0.2$ and maximum normalized Doppler shift $f_{\max,\,norm} = 0.05$. The Doppler object is then assigned to the `DopplerSpectrum` property of the channel object. The channel then has a Doppler spectrum that is nonzero for frequencies f such that $-f_d \leq f_{\min} \leq f \leq f_{\max} \leq f_d$, where $f_{\min} = f_{\min,\,norm} \times f_d = -2$ Hz and $f_{\max} = f_{\max,\,norm} \times f_d = 0.5$ Hz.

```
chan = rayleighchan(1/1000, 10);
dop_ajakes = doppler.ajakes([-0.2 0.05]);
chan.DopplerSpectrum = dop_ajakes;
chan.DopplerSpectrum
```

This code returns:

```
        SpectrumType: 'AJakes'
    FreqMinMaxAJakes: [-0.2000 0.0500]
```

Compatibility Considerations

`doppler.ajakes` has been removed

Errors starting in R2020b

doppler.ajakes has been removed. Use `doppler('Asymmetric Jakes',...)` instead.

References

[1] Jakes, W. C., Ed., *Microwave Mobile Communications*, Wiley, 1974.

[2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007b

doppler.bell

(Removed) Construct bell-shaped Doppler spectrum object

Note `doppler.ajakes` has been removed. Use `doppler('Bell',...)` instead.

Syntax

```
doppler.bell
doppler.bell(coeffbell)
```

Description

doppler.bell creates a bell Doppler spectrum object. You can use this object with the `DopplerSpectrum` property of any channel object created with either the `rayleighchan` function, the `ricianchan` function, or `comm.MIMOChannel` System object.

dop = `doppler.bell` creates a bell Doppler spectrum object with default coefficient.

dop = `doppler.bell(coeffbell)` creates a bell Doppler spectrum object with coefficient given by `coeffbell`, where `coeffbell` is a positive, finite, real scalar.

Properties

The bell Doppler spectrum object has the following properties.

Property	Description
SpectrumType	Fixed value, 'Bell'
CoeffBell	Bell spectrum coefficient, positive real finite scalar.

Theory and Applications

A bell spectrum was proposed in [1] for the Doppler spectrum of indoor MIMO channels, for 802.11n channel modeling.

The normalized bell Doppler spectrum is given analytically by:

$$S(f) = \frac{C_b}{1 + A\left(\frac{f}{f_d}\right)^2}$$

where

$$|f| \leq f_d$$

and

$$C_b = \frac{\sqrt{A}}{\pi f_d}$$

f_d represents the maximum Doppler shift specified for the channel object, and A represents a positive real finite scalar (`CoeffBell`). The indoor MIMO channel model of IEEE 802.11n [1] uses the following parameter: $A = 9$. Since the channel is modeled as Rician fading with a fixed line-of-sight (LOS) component, a Dirac delta is also present in the Doppler spectrum at $f = 0$.

Examples

Construct a bell Doppler spectrum object with a coefficient of 8.5. Assign it to a Rayleigh channel object with one path.

```
dop = doppler.bell(8.5);
chan = rayleighchan(1e-5, 10);
chan.DopplerSpectrum = dop;
```

Compatibility Considerations

`doppler.bell` has been removed

Errors starting in R2020b

doppler.bell has been removed. Use `doppler('Bell',...)` instead.

References

[1] IEEE P802.11 Wireless LANs, “TGn Channel Models”, IEEE 802.1103/940r4, 2004-05-10.

See Also

`comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2009a

doppler.bigaussian

(Removed) Construct bi-Gaussian Doppler spectrum object

Note `doppler.bigaussian` has been removed. Use `doppler('BiGaussian',...)` instead.

Syntax

```
dop = doppler.bigaussian(property1,value1,...)
dop = doppler.bigaussian
```

Description

The `doppler.bigaussian` function creates a bi-Gaussian Doppler spectrum object to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` function or the `ricianchan` function).

`dop = doppler.bigaussian(property1,value1,...)` creates a bi-Gaussian Doppler spectrum object with properties as specified by the property/value pairs. If you do not specify a value for a property, the property is assigned a default value.

`dop = doppler.bigaussian` creates a bi-Gaussian Doppler spectrum object with default properties. The constructed Doppler spectrum object is equivalent to a single Gaussian Doppler spectrum centered at zero frequency. The equivalent command with property/value pairs is:

```
dop = doppler.bigaussian('SigmaGaussian1', 1/sqrt(2), ...
            'SigmaGaussian2', 1/sqrt(2), ...
            'CenterFreqGaussian1', 0, ...
            'CenterFreqGaussian2', 0, ...
            'GainGaussian1', 0.5, ...
            'GainGaussian2', 0.5)
```

Properties

The bi-Gaussian Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'BiGaussian'`
`SigmaGaussian1`	Normalized standard deviation of first Gaussian function (real positive finite scalar value)
`SigmaGaussian2`	Normalized standard deviation of second Gaussian function (real positive finite scalar value)
`CenterFreqGaussian1`	Normalized center frequency of first Gaussian function (real scalar value between -1 and 1)
`CenterFreqGaussian2`	Normalized center frequency of second Gaussian function (real scalar value between -1 and 1)

Property	Description
GainGaussian1	Power gain of first Gaussian function (linear scale, real nonnegative finite scalar value)
GainGaussian2	Power gain of second Gaussian function (linear scale, real nonnegative finite scalar value)

All properties are writable except for the SpectrumType property.

The properties SigmaGaussian1, SigmaGaussian2, GainGaussian1, and GainGaussian2 are normalized by the MaxDopplerShift property of the associated channel object.

Analytically, the normalized standard deviations of the first and second Gaussian functions are determined as $\sigma_{G1,\,norm} = \sigma_{G1}/f_d$ and $\sigma_{G2,\,norm} = \sigma_{G2}/f_d$, respectively, where σ_{G1} and σ_{G2} are the standard deviations of the first and second Gaussian functions, and f_d is the maximum Doppler shift, in hertz. Similarly, the normalized center frequencies of the first and second Gaussian functions are determined as $f_{G1,\,norm} = f_{G1}/f_d$ and $f_{G2,\,norm} = f_{G2}/f_d$, respectively, where f_{G1} and f_{G2} are the center frequencies of the first and second Gaussian functions. The properties GainGaussian1 and GainGaussian2 correspond to the power gains C_{G1} and C_{G2}, respectively, of the two Gaussian functions.

Theory and Applications

The bi-Gaussian power spectrum consists of two frequency-shifted Gaussian spectra. The COST207 channel models ([1], [2], [3]) specify two distinct bi-Gaussian Doppler spectra, GAUS1 and GAUS2, to be used in modeling long echos for urban and hilly terrain profiles.

The normalized bi-Gaussian Doppler spectrum is given analytically by:

$$S_G(f) = A_G\left[\frac{C_{G1}}{\sqrt{2\pi\sigma_{G1}^2}}\exp\left(-\frac{(f-f_{G1})^2}{2\sigma_{G1}^2}\right) + \frac{C_{G2}}{\sqrt{2\pi\sigma_{G2}^2}}\exp\left(-\frac{(f-f_{G2})^2}{2\sigma_{G2}^2}\right)\right]$$

where σ_{G1} and σ_{G2} are standard deviations, f_{G1} and f_{G2} are center frequencies, C_{G1} and C_{G2} are power gains, and $A_G = \dfrac{1}{C_{G1} + C_{G2}}$ is a normalization coefficient.

If either $f_{G1} = 0$ or $f_{G2} = 0$, a frequency-shifted Gaussian Doppler spectrum is obtained.

Examples

The following MATLAB code first creates a bi-Gaussian Doppler spectrum object with the same parameters as that of a COST 207 GAUS2 Doppler spectrum. It then creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 30$ and assigns the constructed Doppler spectrum object to its DopplerSpectrum property.

```
dop_bigaussian = doppler.bigaussian(‘SigmaGaussian1’, 0.1, ...
   ‘SigmaGaussian2’, 0.15, ‘CenterFreqGaussian1’, 0.7, ...
   ‘CenterFreqGaussian2’, -0.4, ‘GainGaussian1’, 1, ...
   ‘GainGaussian2’, 1/10^1.5)
chan = rayleighchan(1e-3, 30);
chan.DopplerSpectrum = dop_bigaussian;
```

Compatibility Considerations

`doppler.bigaussian` has been removed
Errors starting in R2020b

doppler.bigaussian has been removed. Use `doppler('BiGaussian',...)` instead.

References

[1] COST 207 WG1, *Proposal on channel transfer functions to be used in GSM tests late 1986*, COST 207 TD (86) 51 Rev. 3, Sept. 1986.

[2] COST 207, *Digital land mobile radio communications*, Office for Official Publications of the European Communities, Final report, Luxembourg, 1989.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics
"Fading Channels"

Introduced in R2007b

doppler.flat

(Removed) Construct flat Doppler spectrum object

Note `doppler.flat` has been removed. Use `doppler('Flat')` instead.

Syntax

```
dop = doppler.flat
```

Description

`dop = doppler.flat` creates a flat Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function). The maximum Doppler shift of the flat Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object.

Properties

The flat Doppler spectrum object contains only one property, `SpectrumType`, which is read-only and has a fixed value of `'Flat'`.

Theory and Applications

In a 3-D isotropic scattering environment, where the angles of arrival are uniformly distributed in the azimuth and elevation planes, the Doppler spectrum is found theoretically to be flat [2]. A flat Doppler spectrum is also specified in some cases of the ANSI J-STD-008 reference channel models for PCS, for both outdoor (pedestrian) and indoor (commercial) [1] applications.

The normalized flat Doppler power spectrum is given analytically by:

$$S(f) = \frac{1}{2f_d}, \ |f| \leq f_d$$

where f_d is the maximum Doppler frequency.

Examples

```
%% Create a Rayleigh Channel with Flat Doppler Spectrum
% This example shows how to create a Rayleigh channel
% object with a flat Doppler spectrum.
%%
% Set the sample time and maximum Doppler shift.
ts = 1e-6; % sec
fd = 50; % Hz
% Create the Rayleigh channel object.
chan = rayleighchan(ts,fd);
% Observe that the default Doppler spectrum
% property, SpectrumType, is 'Jakes'.
chan.DopplerSpectrum

ans =

    SpectrumType: 'Jakes'

% Change the Doppler spectrum property of the
% channel by using doppler.flat.
chan.DopplerSpectrum = doppler.flat
chan =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-06
         DopplerSpectrum: [1x1 doppler.flat]
         MaxDopplerShift: 50
              PathDelays: 0
           AvgPathGaindB: 0
      NormalizePathGains: 1
            StoreHistory: 0
          StorePathGains: 0
               PathGains: -0.6760 + 0.6319i
      ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```

Compatibility Considerations

`doppler.flat` has been removed
Errors starting in R2020b

doppler.flat has been removed. Use `doppler('Flat')` instead.

References

[1] ANSI J-STD-008, *Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems*, March 1995.

[2] Clarke, R. H., and Khoo, W. L., “3-D Mobile Radio Channel Statistics”, *IEEE Trans. Veh. Technol.*, Vol. 46, No. 3, pp. 798–799, August 1997.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics
“Fading Channels”

Introduced in R2007a

doppler.gaussian

(Removed) Construct Gaussian Doppler spectrum object

Note `doppler.gaussian` has been removed in a future release. Use `doppler ('Gaussian',...)` instead.

Syntax

```
dop = doppler.gaussian
dop = doppler.gaussian(sigmagaussian)
```

Description

The `doppler.gaussian` function creates a Gaussian Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.gaussian` creates a Gaussian Doppler spectrum object with a default standard deviation (normalized by the maximum Doppler shift f_d, in Hz) $\sigma_{G,\,norm} = 1/\sqrt{2}$. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically, $\sigma_{G,\,norm} = \sigma_G/f_d = 1/\sqrt{2}$, where σ_G is the standard deviation of the Gaussian Doppler spectrum.

`dop = doppler.gaussian(sigmagaussian)` creates a Gaussian Doppler spectrum object with a normalized f_d (by the maximum Doppler shift f_d, in Hz) $\sigma_{G,\,norm}$ of value `sigmagaussian`.

Properties

The Gaussian Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'Gaussian'`
`SigmaGaussian`	Normalized standard deviation of the Gaussian Doppler spectrum (a real positive number)

Theory and Applications

The Gaussian power spectrum is considered to be a good model for multipath components with long delays in UHF communications [3]. It is also proposed as a model for the aeronautical channel [2]. A Gaussian Doppler spectrum is also specified in some cases of the ANSI J-STD-008 reference channel models for PCS applications, for both outdoor (wireless loop) and indoor (residential, office) [1]. The normalized Gaussian Doppler power spectrum is given analytically by:

$$S_G(f) = \frac{1}{\sqrt{2\pi\sigma_G^2}}\exp\left(-\frac{f^2}{2\sigma_G^2}\right)$$

An alternate representation is [4]:

$$S_G(f) = \frac{1}{f_c}\sqrt{\frac{\ln 2}{\pi}}\exp\left(-(\ln 2)\left(\frac{f}{f_c}\right)^2\right)$$

where $f_c = \sigma_G\sqrt{2\ln 2}$ is the 3 dB cutoff frequency. If you set $f_c = f_d\sqrt{\ln 2}$, where f_d is the maximum Doppler shift, or equivalently $\sigma_G = f_d/\sqrt{2}$, the Doppler spread of the Gaussian power spectrum becomes equal to the Doppler spread of the Jakes power spectrum, where Doppler spread is defined as:

$$\sigma_D = \sqrt{\frac{\int_{-\infty}^{\infty} f^2 S(f)df}{\int_{-\infty}^{\infty} S(f)df}}$$

Examples

The following code creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$. It then creates a Gaussian Doppler spectrum object with a normalized standard deviation of $\sigma_{G,\mathrm{norm}} = 0.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = rayleighchan(1/1000,10);
dop_gaussian = doppler.gaussian(0.5);
chan.DopplerSpectrum = dop_gaussian;
```

Compatibility Considerations

`doppler.gaussian` has been removed

Errors starting in R2020b

doppler.gaussian has been removed. Use `doppler('Gaussian',...)` instead.

References

[1] ANSI J-STD-008, *Personal Station-Base Station Compatibility Requirements for 1.8 to 2.0 GHz Code Division Multiple Access (CDMA) Personal Communications Systems*, March 1995.

[2] Bello, P. A., “Aeronautical channel characterizations,” *IEEE Trans. Commun.*, Vol. 21, pp. 548–563, May 1973.

[3] Cox, D. C., “Delay Doppler characteristics of multipath propagation at 910 MHz in a suburban mobile radio environment,” *IEEE Transactions on Antennas and Propagation*, Vol. AP-20, No. 5, pp. 625–635, Sept. 1972.

[4] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007a

doppler.jakes

(Removed) Construct Jakes Doppler spectrum object

Note `doppler.jakes` has been removed. Use `doppler('Jakes')` instead.

Syntax

Description

`dop = doppler.jakes` creates a Jakes Doppler spectrum object that is to be used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function). The maximum Doppler shift of the Jakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object. By default, channel objects are created with a Jakes Doppler spectrum.

Properties

The Jakes Doppler spectrum object contains only one property, `SpectrumType`, which is read-only and has a fixed value of `'Jakes'`.

Theory and Applications

The Jakes Doppler power spectrum model is actually due to Gans [2], who analyzed the Clarke-Gilbert model ([1], [3], and [5]). The Clarke-Gilbert model is also called the *classical model*.

The Jakes Doppler power spectrum applies to a mobile receiver. It derives from the following assumptions [6]:

- The radio waves propagate horizontally.
- At the mobile receiver, the angles of arrival of the radio waves are uniformly distributed over $[-\pi, \pi]$.
- At the mobile receiver, the antenna is omnidirectional (i.e., the antenna pattern is circular-symmetrical).

The normalized Jakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{1}{\pi f_d \sqrt{1 - (f/f_d)^2}}, \ |f| \le f_d$$

where f_d is the maximum Doppler frequency.

Examples

Create a Rayleigh channel object with a maximum Doppler shift of fd=10 Hertz. Then, create a Jakes Doppler spectrum object and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = rayleighchan(1/1000,10);
dop_gaussian = doppler.jakes;
chan.DopplerSpectrum = dop_gaussian
```

Compatibility Considerations

`doppler.jakes` has been removed

Errors starting in R2020b

doppler.jakes has been removed. Use `doppler('Jakes')` instead.

References

[1] Clarke, R. H., “A Statistical Theory of Mobile-Radio Reception,” *Bell System Technical Journal*, Vol. 47, No. 6, pp. 957–1000, July-August 1968.

[2] Gans, M. J., “A Power-Spectral Theory of Propagation in the Mobile-Radio Environment,” *IEEE Trans. Veh. Technol.*, Vol. VT-21, No. 1, pp. 27–38, Feb. 1972.

[3] Gilbert, E. N., “Energy Reception for Mobile Radio,” *Bell System Technical Journal*, Vol. 44, No. 8, pp. 1779–1803, Oct. 1965.

[4] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.

[5] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[6] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2007a

doppler.rjakes

(Removed) Construct restricted Jakes Doppler spectrum object

Note `doppler.rjakes` has been removed. Use `doppler('Restricted Jakes',...)` instead.

Syntax

```
dop = doppler.rjakes
dop = doppler.rjakes(freqminmaxrjakes)
```

Description

The `doppler.rjakes` function creates a restricted Jakes (RJakes) Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rjakes` creates a Doppler spectrum object equivalent to the Jakes Doppler spectrum. The maximum Doppler shift of the RJakes Doppler spectrum object is specified by the `MaxDopplerShift` property of the channel object.

`dop = doppler.rjakes(freqminmaxrjakes)`, where `freqminmaxrjakes` is a row vector of two finite real numbers between 0 and 1, creates a Jakes Doppler spectrum. This spectrum is nonzero only for normalized frequencies (by the maximum Doppler shift, f_d, in Hertz), f_{norm}, such that $0 \le f_{\min,\,norm} \le |f_{norm}| \le f_{\max,\,norm} \le 1$, where $f_{\min,\,norm}$ is given by `freqminmaxrjakes(1)` and $f_{\max,\,norm}$ is given by `freqminmaxrjakes(2)`. The maximum Doppler shift f_d is specified by the `MaxDopplerShift` property of the channel object. Analytically, $f_{\min,\,norm} = f_{\min}/f_d$ and $f_{\max,\,norm} = f_{\max}/f_d$, where $f_{\min}$ is the minimum Doppler shift (in Hertz) and $f_{\max}$ is the maximum Doppler shift (in Hertz).

When `dop` is used as the `DopplerSpectrum` property of a channel object, `freqminmaxrjakes(1)` and `freqminmaxrjakes(2)` should be spaced by more than 1/50. Assigning a smaller spacing results in `freqminmaxrjakes` being reset to the default value of `[0 1]`.

Properties

The RJakes Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'RJakes'`
`FreqMinMaxRJakes`	Vector of minimum and maximum normalized Doppler shifts (two real finite numbers between 0 and 1)

Theory and Applications

The Jakes power spectrum is based on the assumption that the angles of arrival at the mobile receiver are uniformly distributed [1], where the spectrum covers the frequency range from $-f_d$ to f_d, f_d

being the maximum Doppler shift. When the angles of arrival are not uniformly distributed, the Jakes power spectrum does not cover the full Doppler bandwidth from $-f_d$ to f_d. This exception also applies to the case where the antenna pattern is directional. This type of spectrum is known as *restricted Jakes* [3]. The RJakes Doppler spectrum object covers only the case of a symmetrical power spectrum, which is nonzero only for frequencies f such that $0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$.

The normalized RJakes Doppler power spectrum is given analytically by:

$$S(f) = \frac{A_r}{\pi f_d \sqrt{1 - (f/f_d)^2}},\ 0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$$

where

$$A_r = \frac{1}{\frac{2}{\pi}\left[\sin^{-1}\left(\frac{f_{\max}}{f_d}\right) - \sin^{-1}\left(\frac{f_{\min}}{f_d}\right)\right]}$$

$f_{\min}$ and $f_{\max}$ denote the minimum and maximum frequencies where the spectrum is nonzero. They can be determined from the probability density function of the angles of arrival.

Examples

The following code first creates a Rayleigh channel object with a maximum Doppler shift of $f_d = 10$. It then creates an RJakes Doppler object with minimum normalized Doppler shift $f_{\min,\,norm} = 0.14$ and maximum normalized Doppler shift $f_{\max,\,norm} = 0.9$.

The Doppler object is assigned to the `DopplerSpectrum` property of the channel object. The channel then has a Doppler spectrum that is nonzero for frequencies f such that $0 \leq f_{\min} \leq |f| \leq f_{\max} \leq f_d$, where $f_{\min} = f_{\min,\,norm} \times f_d = 1.4$ Hz and $f_{\max} = f_{\max,\,norm} \times f_d = 9$ Hz.

```
chan = rayleighchan(1/1000, 10);
dop_rjakes = doppler.rjakes([0.14 0.9]);
chan.DopplerSpectrum = dop_rjakes;
chan.DopplerSpectrum
```

The output is:

```
             SpectrumType: 'RJakes'
    FreqMinMaxRJakes: [0.1400 0.9000]
```

Compatibility Considerations

`doppler.rjakes` has been removed

Errors starting in R2020b

doppler.rjakes has been removed. Use `doppler('Restricted Jakes',...)` instead.

References

[1] Jakes, W. C., Ed. *Microwave Mobile Communications*, Wiley, 1974.

[2] Lee, W. C. Y., *Mobile Communications Engineering: Theory and Applications*, 2nd Ed., McGraw-Hill, 1998.

[3] Pätzold, M., *Mobile Fading Channels*, Wiley, 2002.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

"Fading Channels"

Introduced in R2007a

doppler.rounded

(Removed) Construct rounded Doppler spectrum object

Note `doppler.rounded` has been removed. Use `doppler('Rounded',...)` instead.

Syntax

```
dop = doppler.rounded
dop = doppler.rounded(coeffrounded)
```

Description

The `doppler.rounded` function creates a rounded Doppler spectrum object that is used for the `DopplerSpectrum` property of a channel object (created with either the `rayleighchan` or the `ricianchan` function).

`dop = doppler.rounded` creates a rounded Doppler spectrum object with default polynomial coefficients $a_0 = 1$, $a_2 = -1.72$, $a_4 = 0.785$ (see “Theory and Applications” on page 2-300 for the meaning of these coefficients). The maximum Doppler shift f_d (in Hertz) is specified by the `MaxDopplerShift` property of the channel object.

`dop = doppler.rounded(coeffrounded)`, where `coeffrounded` is a row vector of three finite real numbers, creates a rounded Doppler spectrum object with polynomial coefficients, a_0, a_2, a_4, given by `coeffrounded(1)`, `coeffrounded(2)`, and `coeffrounded(3)`, respectively.

Properties

The rounded Doppler spectrum object contains the following properties.

Property	Description
`SpectrumType`	Fixed value, `'Rounded'`
`CoeffRounded`	Vector of three polynomial coefficients (real finite numbers)

Theory and Applications

A rounded spectrum is proposed as an approximation to the measured Doppler spectrum of the scatter component of fixed wireless channels at 2.5 GHz [1]. However, the shape of the spectrum is influenced by the center carrier frequency.

The normalized rounded Doppler spectrum is given analytically by a polynomial in f of order four, where only the even powers of f are retained:

$$S(f) = C_r\left[a_0 + a_2\left(\frac{f}{f_d}\right)^2 + a_4\left(\frac{f}{f_d}\right)^4\right],\ |f| \le f_d$$

where

$$C_r = \frac{1}{2f_d\left[a_0 + \frac{a_2}{3} + \frac{a_4}{5}\right]}$$

f_d is the maximum Doppler shift, and a_0, a_2, a_4 are real finite coefficients. The fixed wireless channel model of IEEE 802.16 [1] uses the following parameters: $a_0 = 1$, $a_2 = -1.72$, and $a_4 = 0.785$. Because the channel is modeled as Rician fading with a fixed line-of-sight (LOS) component, a Dirac delta is also present in the Doppler spectrum at $f = 0$.

Examples

The following code creates a Rician channel object with a maximum Doppler shift of $f_d = 10$. It then creates a rounded Doppler spectrum object with polynomial coefficients $a_0 = 1.0$, $a_2 = -0.5$, $a_4 = 1.5$, and assigns it to the `DopplerSpectrum` property of the channel object.

```
chan = ricianchan(1/1000,10,1);
dop_rounded = doppler.rounded([1.0 -0.5 1.5]);
chan.DopplerSpectrum = dop_rounded;
```

Compatibility Considerations

`doppler.rounded` has been removed

Errors starting in R2020b

doppler.rounded has been removed. Use `doppler('Rounded',...)` instead.

References

[1] IEEE 802.16 Broadband Wireless Access Working Group, “Channel models for fixed wireless applications,” *IEEE 802.16a-03/01*, 2003-06-27.

See Also

`comm.RayleighChannel` | `comm.RicianChannel` | `doppler` | `stdchan`

Topics

“Fading Channels”

Introduced in R2007a

dpcmdeco

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, `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 “Represent Partitions”, “Represent Codebooks”, or the `quantiz` reference page, 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” and “Example: Comparing Optimized and Nonoptimized DPCM Parameters” for examples that use `dpcmdeco`.

References

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

See Also

`dpcmenco` | `dpcmopt` | `quantiz`

Topics

“Differential Pulse Code Modulation”

Introduced before R2006a

dpcmenco

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, `predictor` has length M+1 and an initial entry of 0. The output vector `indx` is the quantization index.

See “Differential Pulse Code Modulation” for more about the format of `predictor`. See “Represent Partitions”, “Represent Partitions”, 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, the modulation is called a *delta modulation*.

Examples

See “Example: DPCM Encoding and Decoding” and “Example: Comparing Optimized and Nonoptimized DPCM Parameters” for examples that use `dpcmenco`.

References

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

See Also

`dpcmdeco` | `dpcmopt` | `quantiz`

Topics

“Differential Pulse Code Modulation”

Introduced before R2006a

dpcmopt

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,ini_cb)
```

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 "Represent Predictors" 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 "Represent Partitions", "Represent Codebooks", 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,ini_cb)` is the same as the first syntax, except that it also returns corresponding optimized codebook and partition vectors `codebook` and `partition`. `ini_cb`, 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 `ini_cb`. 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" for an example that uses `dpcmopt`.

See Also

`dpcmdeco` | `dpcmenco` | `lloyds` | `quantiz`

Topics

"Differential Pulse Code Modulation"

Introduced before R2006a

dpskdemod

Differential phase shift keying demodulation

Syntax

```
z = dpskdemod(y,M)
z = dpskdemod(y,M,phaserot)
z = dpskdemod(y,M,phaserot,symorder)
```

Description

`z = dpskdemod(y,M)` demodulates the complex envelope, `y`, of a DPSK-modulated signal having modulation order `M`.

`z = dpskdemod(y,M,phaserot)` specifies the phase rotation of the DPSK modulation.

`z = dpskdemod(y,M,phaserot,symorder)` also specifies the symbol order.

Examples

DPSK Demodulation

Demodulate DPSK data in a communication channel in which a phase shift is introduced.

Generate a 4-ary data vector and modulate it using DPSK.

```
M = 4;                         % Alphabet size
dataIn = randi([0 M-1],1000,1); % Random message
txSig = dpskmod(dataIn,M);      % Modulate
```

Apply the random phase shift resulting from the transmission process.

```
rxSig = txSig*exp(2i*pi*rand());
```

Demodulate the received signal.

```
dataOut = dpskdemod(rxSig,M);
```

The modulator and demodulator have the same initial condition. However, only the received signal experiences a phase shift. As a result, the first demodulated symbol is likely to be in error. Therefore, you should always discard the first symbol when using DPSK.

Find the number of symbol errors.

```
errs = symerr(dataIn,dataOut)
```

```
errs = 1
```

One symbol is in error. Repeat the error calculation after discarding the first symbol.

```
errs = symerr(dataIn(2:end),dataIn(2:end))
errs = 0
```

Input Arguments

y — DPSK-modulated input signal

vector | matrix

DPSK-modulated input signal, specified as a real or complex vector or matrix. If `y` is a matrix, the function processes the columns independently.

Data Types: `double`
Complex Number Support: Yes

`M` — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: `2` | `4` | `16`

Data Types: `double`

`phaserot` — Phase rotation

0 (default) | scalar | `[]`

Phase rotation of the DPSK modulation, specified in radians as a real scalar. The total phase shift per symbol is the sum of `phaserot` and the phase generated by the differential modulation.

If you specify `phaserot` as empty, then `dspkdemod` uses a phase rotation of 0 degrees.

Example: `pi/4`

Data Types: `double`

`symorder` — Symbol order

`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

Output Arguments

z — DPSK-demodulated output signal

vector | matrix

DPSK-demodulated output signal, returned as a vector or matrix having the same number of columns as input signal `y`.

Note The differential algorithm used in this function compares two successive elements of a modulated signal. To determine the first element of vector `z`, or the first row of matrix `z`, the function uses an initial phase rotation of `0`.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.DPSKDemodulator` | `dpskmod` | `pskdemod` | `pskmod`

Topics

"Phase Modulation"

Introduced before R2006a

dpskmod

Differential phase shift keying modulation

Syntax

```
y = dpskmod(x,M)
y = dpskmod(x,M,phaserot)
y = dpskmod(x,M,phaserot,symorder)
```

Description

`y = dpskmod(x,M)` modulates the input signal using differential phase shift keying (DPSK) with modulation order `M`.

`y = dpskmod(x,M,phaserot)` specifies the phase rotation of the DPSK modulation.

`y = dpskmod(x,M,phaserot,symorder)` also specifies the symbol order.

Examples

View Signal Trajectory of DPSK-Modulated Signal

Plot the output of the `dspkmod` function to view the possible transitions between DPSK symbols.

Set the modulation order to 4 to model DQPSK modulation.

```
M = 4;
```

Generate a sequence of 4-ary random symbols.

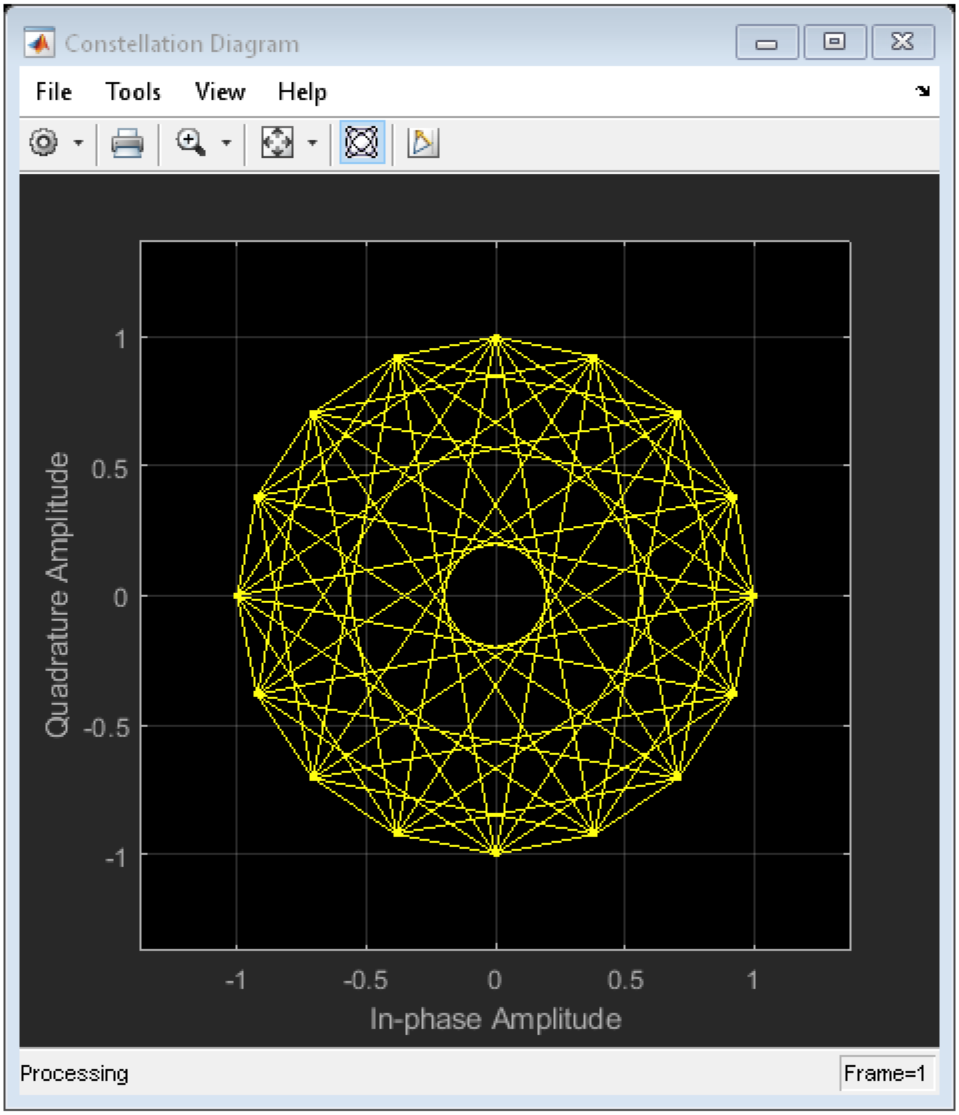
```
x = randi([0 M-1],500,1);
```

Apply DQPSK modulation to the input symbols.

```
y = dpskmod(x,M,pi/8);
```

Specify a constellation diagram object to display a signal trajectory diagram and without displaying the corresponding reference constellation. Display the trajectory.

```
cd = comm.ConstellationDiagram('ShowTrajectory',true,'ShowReferenceConstellation',false);
cd(y)
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of x must have values in the range of [0, M – 1].

Data Types: double

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double

phaserot — Phase rotation

0 (default) | scalar | [ ]

Phase rotation of the DPSK modulation, specified in radians as a real scalar. The total phase shift per symbol is the sum of phaserot and the phase generated by the differential modulation.

If you specify phaserot as empty, then dpskmod uses a phase rotation of 0 degrees.

Example: pi/4

Data Types: double

symorder — Symbol order

'bin' (default) | 'gray'

Symbol order, specified as 'bin' or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

- If symorder is 'bin', the function uses a natural binary-coded ordering.
- If symorder is 'gray', the function uses a Gray-coded ordering.

Data Types: char

Output Arguments

y — DPSK-modulated output signal

vector | matrix

Complex baseband representation of a DPSK-modulated output signal, returned as vector or matrix of complex values. The columns represent independent channels.

Note An initial phase rotation of 0 is used in determining the first element of the output y (or the first row of y if it is a matrix with multiple rows), because two successive elements are required for a differential algorithm.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

comm.DPSKModulator | dpskdemod | pskdemod | pskmod

Topics

“Phase Modulation”

Introduced before R2006a

dvbs2ldpc

Low-density parity-check (LDPC) codes from DVB-S.2 standard

Syntax

```
H = dvbs2ldpc(r)
H = dvbs2ldpc(r,outputFormat)
```

Description

`H = dvbs2ldpc(r)` returns the parity-check matrix `H` of the LDPC code with code rate `r` from the Digital Video Broadcasting standard DVB-S.2. The block length of the code is 64,800.

`H = dvbs2ldpc(r,outputFormat)` specifies the format for the output parity-check matrix.

Examples

Create LDPC Code Parity-Check Matrix from DVB-S.2 Standard

Create an LDPC parity check matrix for a code rate of 3/5 from the DVB-S.2 standard.

```
p = dvbs2ldpc(3/5);
```

Create an LDPC encoder object from the parity-check matrix `p`.

```
enc = comm.LDPCEncoder(p);
```

The parity-check matrix has dimensions of (*N*-*K*)-by-*N*. Calculate the length of the input message.

```
msgLength = size(p,2) - size(p,1)
```

```
msgLength = 38880
```

Input Arguments

`r` — Code rate
1/4 | 1/3 | 2/5 | 1/2 | ...

Code rate, specified as `1/4`, `1/3`, `2/5`, `1/2`, `3/5`, `2/3`, `3/4`, `4/5`, `5/6`, `8/9`, or `9/10`.

Data Types: `double`

`outputFormat` — Output format
`'sparse'` | `'indices'`

Output format for parity-check matrix `H`, specified as `'sparse'` or `'indices'`.

If you set this value to `'sparse'`, `H` is a sparse logical matrix. If you set this value to `'indices'`, `H` is a two-column matrix that defines the row and column indices of the 1s in `H`.

Data Types: `char` | `string`

Output Arguments

`H` — Parity-check matrix

matrix

Parity-check matrix, returned as a matrix.

The default parity-check matrix of size 32,400-by-64,800 corresponds to an irregular LDPC code with the structure shown in this table.

Row	Number of 1s per Row
1	6
2 to 32400	7

Column	Number of 1s per Column
1 to 12960	8
12961–32400	3

Columns from 32,401 to 64,800 form a lower triangular matrix. Only the elements on the main diagonal of the matrix and the subdiagonal immediately below the main diagonal are 1s. This LDPC code is used in conjunction with a BCH code in the DVB-S.2 standard to achieve a packet error rate below 10^{-7} at about 0.7 dB to 1 dB from the Shannon limit.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`comm.LDPCDecoder` | `comm.LDPCEncoder`

Introduced in R2007a

dvbsapskdemod

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) demodulation

Syntax

```
z = dvbsapskdemod(y,M,stdSuffix)
z = dvbsapskdemod(y,M,stdSuffix,codeIDF)
z = dvbsapskdemod(y,M,stdSuffix,codeIDF,frameLength)
z = dvbsapskdemod( ___ ,Name,Value)
```

Description

`z = dvbsapskdemod(y,M,stdSuffix)` demodulates an APSK input signal, `y`, that was modulated in accordance with the digital video broadcast (DVB) standard identified by `stdSuffix` and the modulation order, `M`. For a description of DVB-compliant APSK demodulation, see “DVB Compliant APSK Hard Demodulation” on page 2-319 and “DVB Compliant APSK Soft Demodulation” on page 2-319.

`z = dvbsapskdemod(y,M,stdSuffix,codeIDF)` specifies code identifier `codeIDF`, to use when selecting the demodulation parameters.

`z = dvbsapskdemod(y,M,stdSuffix,codeIDF,frameLength)` specifies `codeIDF` and `frameLength` to use when selecting the demodulation parameters.

`z = dvbsapskdemod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate DVB-S2X Specific 64-APSK Signal

Demodulate a 64-APSK signal that was modulated as specified in DVB-S2X. Compute hard decision integer output and verify that the output matches the input.

Set the modulation order and standard suffix. Generate random data.

```
M = 64;
std = 's2x';
x = randi([0 M-1],1000,1);
```

Modulate the data.

```
y = dvbsapskmod(x,M,std);
```

Demodulate the received signal. Compare the demodulated data to the original data.

```
z = dvbsapskdemod(y,M,std);
isequal(z,x)
```

```
ans = logical
   1
```

Demodulate DVB-S2 Specific 32-APSK Signal

Demodulate a 32-APSK signal that was modulated as specified in DVB-S2. Compute hard decision bit output and verify that the output matches the input.

Set the modulation order, standard suffix, and code identifier. Generate random bit data.

```
M = 32;
std = 's2';
codeIDF = '4/5';
numBitsPerSym = log2(M);
x = randi([0 1],100*numBitsPerSym,1,'uint32');
```

Modulate the data. Use a name-value pair to specify bit input data.

```
y = dvbsapskmod(x,M,std,codeIDF,'InputType','bit');
```

Demodulate the received signal. Compare the demodulated data to the original data.

```
z = dvbsapskdemod(y,M,std,'4/5','OutputType','bit', ...
    'OutputDataType','uint32');
isequal(z,x)
```

```
ans = logical
   1
```

Soft Bit Demodulate DVB-SH Specific 16-APSK Signal

Demodulate a DVB-SH compliant 16-APSK signal and calculate soft bits.

Set the modulation order and generate a random bit sequence.

```
M = 16;
std = 'sh';
numSym = 20000;
numBitsPerSym = log2(M);
x = randi([0 1],numSym*numBitsPerSym,1);
```

Modulate the data. Use a name-value pair to specify bit input data.

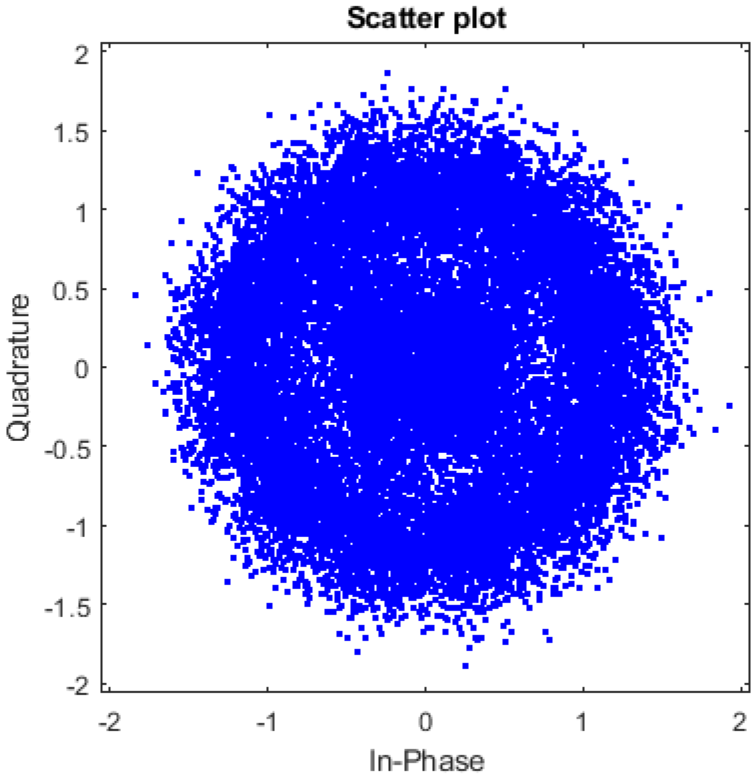
```
txSig = dvbsapskmod(x,M,std,'InputType','bit');
```

Pass the modulated signal through a noisy channel.

```
rxSig = awgn(txSig,10,'measured');
```

View the constellation of the received signal using a scatter plot.

```
scatterplot(rxSig)
```



DVB-SH compliant constellations have unit average power. Demodulate the signal, computing soft bits using the approximate LLR algorithm.

```
z = dvbsapskdemod(rxSig,M,std,'OutputType','approxllr', ...
     'NoiseVariance',0.1);
```

Input Arguments

y — APSK modulated signal

scalar | vector | matrix

APSK modulated signal, specified as a complex scalar, vector, or matrix. When y is a matrix, each column is treated as an independent channel.

y must be modulated in accordance with Digital Video Broadcasting (DVB) - Satellite Communications standard DVB-S2, DVB-S2X or DVB-SH. For more information, see [1], [2], and [3].

Data Types: single | double
Complex Number Support: Yes

M — Modulation order

integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Data Types: double

stdSuffix — Standard suffix

's2' | 's2x' | 's2h'

Standard suffix for DVBS modulation variant, specified as 's2', 's2x', or 's2h'.

Data Types: char | string

codeIDF — Code identifier

char | string

Code identifier, specified as a character vector or string. This table lists the acceptable codeIDF values.

Constellation Order (M)	Applicable Standard (stdSuffix)	Acceptable Code Identifier (CodeIDF) Values
16	's2' or 's2x'	'2/3', '3/4', '4/5', '5/6', '8/9', '9/10'
16	's2x'	'26/45', '3/5', '28/45', '23/36', '25/36', '13/18', '7/9', '77/90', '100/180', '96/180', '90/180', '18/30', '20/30'
32	's2' or 's2x'	'3/4', '4/5', '5/6', '8/9', '9/10'
32	's2x'	'32/45', '11/15', '7/9', '2/3'
64	's2x'	'11/15', '7/9', '4/5', '5/6', '128/180'
128	's2x'	'3/4', '7/9'
256	's2x'	'32/45', '3/4', '116/180', '20/30', '124/180', '22/30'

For more information, refer to Tables 9 and 10 in the DVB-S2 standard [1] and Table 17a in the DVB-S2X standard [2].

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

frameLength — Frame length

'normal' (default) | 'short'

Frame length, specified as 'normal' or 'short'. The function usesframeLength and codeIDF to select the modulation parameters.

Dependencies

This input argument applies only when stdSuffix is set to 's2' or 's2x'.

Data Types: char | string

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = dvbsapskdemod(x,M,stdSuffix,'InputType','bit','OutputDataType','single');`

`OutputType` — Output type

`'integer'` (default) | `'bit'` | `'llr'` | `'approxllr'`

Output type, specified as the comma-separated pair consisting of `OutputType` and `'integer'`, `'bit'`, `'llr'`, or `'approxllr'`. For a description of returned output, see `z`.

Data Types: `char` | `string`

`OutputDataType` — Output data type

`'double'` (default) | `...`

Output data type, specified as the comma-separated pair consisting of `OutputDataType` and one of the indicated data types. Acceptable values for `OutputDataType` depend on the `OutputType` value.

`OutputType` Value	Acceptable `OutputDataType` Values
`'integer'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, or `'uint32'`
`'bit'`	`'double'`, `'single'`, `'int8'`, `'int16'`, `'int32'`, `'uint8'`, `'uint16'`, `'uint32'`, or `'logical'`

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'integer'` or `'bit'`.

Data Types: `char` | `string`

`UnitAveragePower` — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `UnitAveragePower` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the function scales the constellation based on specifications in the relevant standard, as described in [1] and [2].

Note When `stdSuffix` is set to `'sh'`, the constellation always has unit average power.

Dependencies

This name-value pair argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `logical`

`NoiseVariance` — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of NoiseVariance and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “DVB Compliant APSK Soft Demodulation” on page 2-319 for algorithm selection considerations.

Dependencies

This name-value pair argument applies only when OutputType is set to 'llr' or 'approxllr'.

Data Types: double

PlotConstellation — Option to plot constellation

false (default) | true

Option to plot constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a logical scalar. To plot the constellation, set PlotConstellation to true.

Data Types: logical

Output Arguments

z — Demodulated signal

scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of the output vary depending on the specified OutputType value.

OutputType Value	Return Value of dvbsapskdemod	Dimensions of z
'integer'	Demodulated integer values from 0 to (M – 1)	z has the same dimensions as input y.
'bit'	Demodulated bits	The number of rows in z is $\log_2$(sum(M)) times the number of rows in y. Each demodulated symbol is mapped to a group of $\log_2$(sum(M)) elements in a column, where the first element represents the MSB and the last element represents the LSB.
'llr'	Log-likelihood ratio value for each bit	
'approxllr'	Approximate log-likelihood ratio value for each bit	

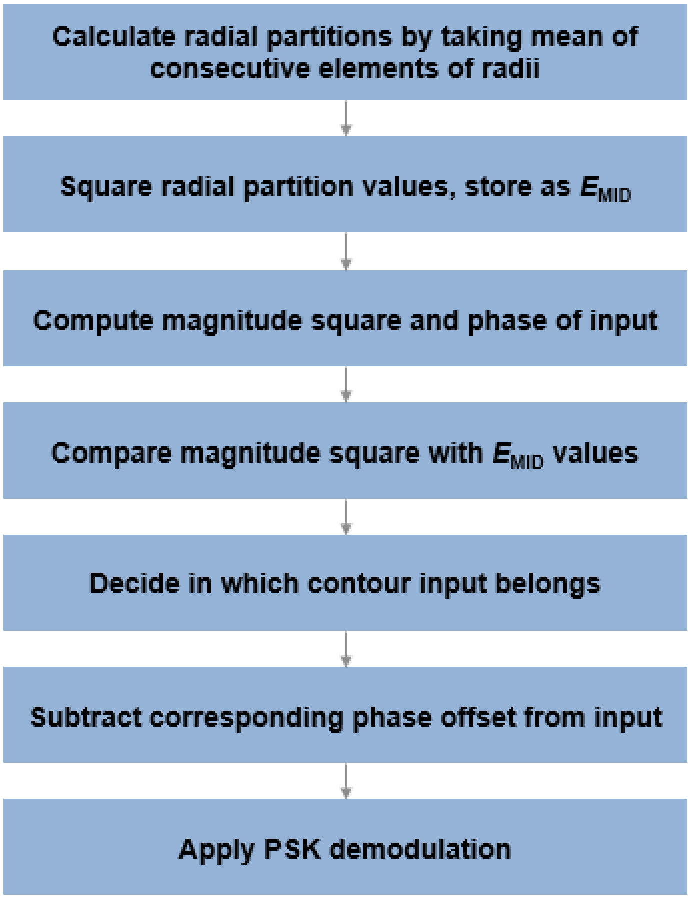
More About

DVB-S2/S2X/SH

Digital video broadcasting (DVB) standards specify S2, S2X, and SH standard-specific amplitude phase shift keying (APSK) modulation. For further information on the DVB-S2/S2X/SH standards, see specified in [1], [2], and [3], respectively.

DVB Compliant APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding as described in [4].


Calculate radial partitions by taking mean of consecutive elements of radii
Square radial partition values, store as E_{MID}
Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide in which contour input belongs
Subtract corresponding phase offset from input
Apply PSK demodulation

DVB Compliant APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value

- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

[4] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskmod` | `genqamdemod` | `mil188qamdemod` | `pskdemod` | `qamdemod`

Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

dvbsapskmod

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) modulation

Syntax

```
y = dvbsapskmod(x,M,stdSuffix)
y = dvbsapskmod(x,M,stdSuffix,codeIDF)
y = dvbsapskmod(x,M,stdSuffix,codeIDF,frameLength)
y = dvbsapskmod(___,Name,Value)
```

Description

`y = dvbsapskmod(x,M,stdSuffix)` performs APSK modulation on the input signal, `x`, in accordance with the digital video broadcast (DVB) standard identified by `stdSuffix` and the modulation order, `M`.

`y = dvbsapskmod(x,M,stdSuffix,codeIDF)` specifies the code identifier, `codeIDF`, to use when selecting the modulation parameters.

`y = dvbsapskmod(x,M,stdSuffix,codeIDF,frameLength)` specifies `codeIDF` and `frameLength` to use when selecting the modulation parameters.

`y = dvbsapskmod(___,Name,Value)` specifies options using one or more name-value pair arguments using any of the previous syntaxes. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply DVB-S2X 32-APSK Modulation to Data

Modulate data using the DVB-S2X standard specified 32-APSK modulation scheme. Display the result in a scatter plot.

Set the modulation order and the suffix identifying the DVB-S2X standard. Create a data vector with all possible symbols.

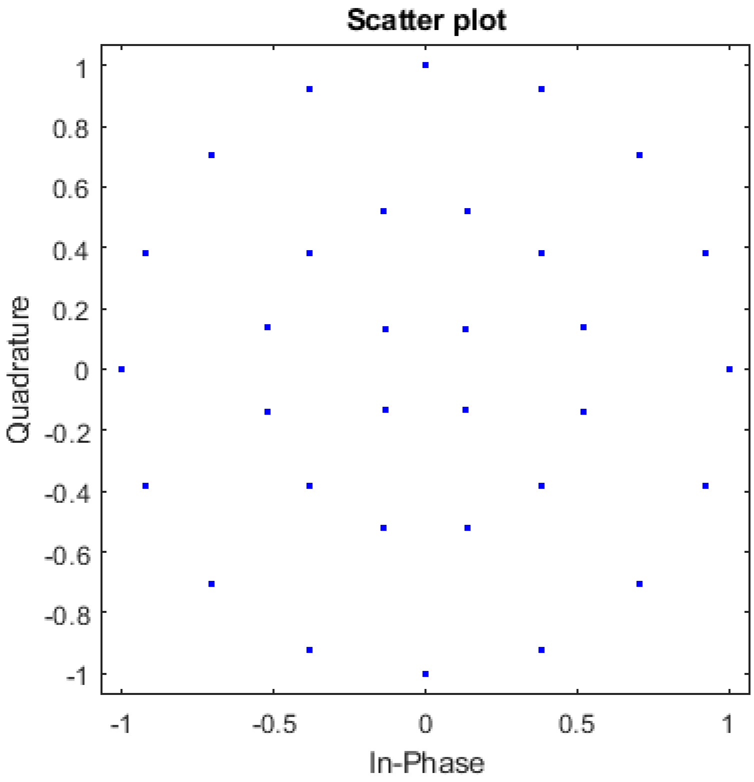
```
M = 32;
stdSuffix = 's2x';
x = (0:M-1);
```

Modulate the data.

```
y = dvbsapskmod(x,M,stdSuffix);
```

Display the constellation using a scatter plot.

```
scatterplot(y)
```



Apply DVB-S2X 64-APSK Modulation Specifying Code Identifier

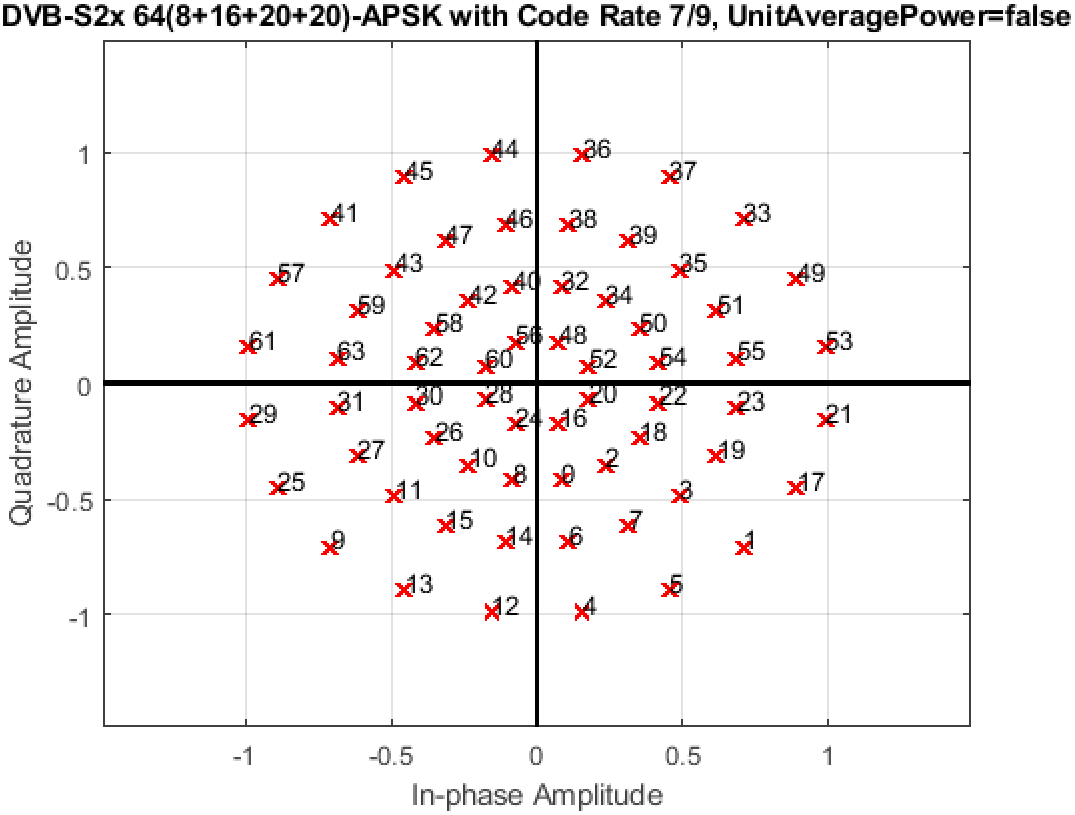
Modulate data using 64-APSK as specified in DVB-S2X standard. Plot constellation for different code identifiers.

Set the modulation order and standard suffix. Generate 1000 symbols of random data in one channel.

```
M = 64;
std = 's2x';
x = randi([0 M-1],1000,1);
```

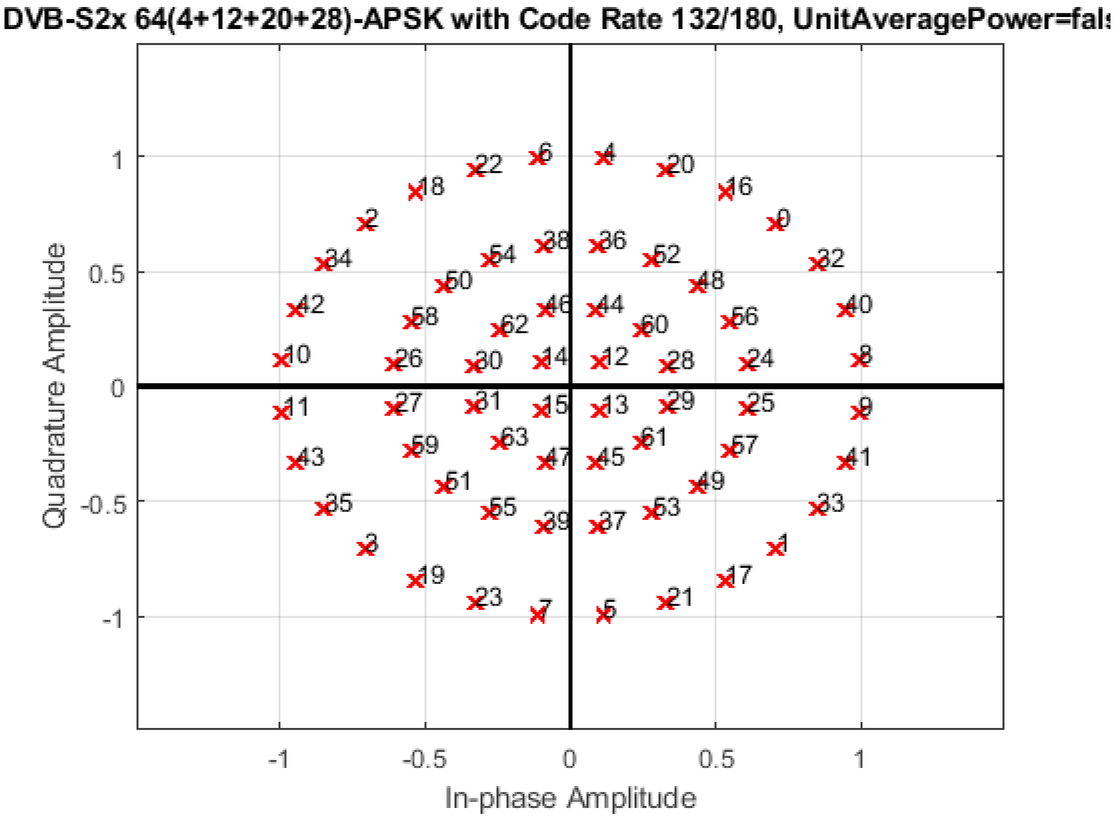
Modulate the data according to the 64-APSK constellation for the code identifier 7/9 and plot the reference constellation.

```
y1 = dvbsapskmod(x,M,std,'7/9','PlotConstellation',true);
```



Modulate setting the code identifier to 132/180 and observe the constellation structure differences.

```
y2 = dvbsapskmod(x,M,std,'132/180','PlotConstellation',true);
```



Apply DVB-S2 16-APSK Modulation Change Frame Length

Modulate data using 16-APSK as specified in DVB-S2 standard for normal and short frame lengths. Compute the output signal power.

Set the modulation order and the standard suffix. Generate random bit data for 1000 symbols in one channel.

```
M = 16;
std = 's2';
x = randi([0 1],1000*log2(M),1);
```

Set the input type to bit and modulate the data according to the 16-APSK constellation for code identifier 2/3. Use the default normal frame length.

```
y1 = dvbsapskmod(x,M,std,'2/3','InputType','bit');
```

Modulate the data using different settings, set the code-identifier to 8/9 and use a short frame length.

```
y2 = dvbsapskmod(x,M,std,'8/9','short','InputType','bit');
```

The average power of the modulated signal changes based on the code identifier. Compute the average power of the modulated signals.

```
y1avgPow = mean(abs(y1).^2)
```

```
y1avgPow = 0.7590
```

```
y2avgPow = mean(abs(y2).^2)
```

```
y2avgPow = 0.7716
```

Normalize 16-APSK Modulated DVB Signals by Average Power

Modulate data applying 16-APSK as specified in the DVB-SH and DVB-S2 standards. Normalize the modulator output so that it has an average signal power of 1 W.

Set the modulation order and generate all possible symbols.

```
M = 16;
x =  0:M-1;
```

Modulate the data applying 16-APSK as specified in DVB-SH. Use a name-value pair to specify single data type output.

```
y1 = dvbsapskmod(x,M,'sh','OutputDataType','single');
```

Modulate the data applying 16-APSK as specified in DVB-S2. Use a name-value pair to specify single data type output.

```
y2 = dvbsapskmod(x,M,'s2','OutputDataType','single');
```

Modulate the data applying 16-APSK as specified in DVB-S2. Use name-value pairs to set unit average power to true and to specify single data type output.

```
y3 = dvbsapskmod(x,M,'s2','UnitAveragePower',true,'OutputDataType','single');
```

Check which signals have unit average power.

```
y1avgPow = mean(abs(y1).^2)
```

```
y1avgPow = single
    1
```

```
y2avgPow = mean(abs(y2).^2)
```

```
y2avgPow = single
    0.7752
```

```
y3avgPow = mean(abs(y3).^2)
```

```
y3avgPow = single
    1.0000
```

Input Arguments

x — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers that range from 0 to (M – 1), where M is the modulation order.

Note To process the input signal as binary elements, set 'InputType' value to 'bit'. For binary inputs, the number of rows must be an integer multiple of $\log_2$(M). A group of $\log_2$(M) bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: double | single | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

M — Modulation order
integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Data Types: double

stdSuffix — Standard suffix
's2' | 's2x' | 'sh'

Standard suffix for DVBS modulation variant, specified as 's2', 's2x', or 'sh'.

Data Types: char | string

codeIDF — Code identifier
char | string

Code identifier, specified as a character vector or string. This table lists the acceptable codeIDF values.

Constellation Order (M)	Applicable Standard (stdSuffix)	Acceptable Code Identifier (CodeIDF) Values
16	's2' or 's2x'	'2/3', '3/4', '4/5', '5/6', '8/9', '9/10'
16	's2x'	'26/45', '3/5', '28/45', '23/36', '25/36', '13/18', '7/9', '77/90', '100/180', '96/180', '90/180', '18/30', '20/30'
32	's2' or 's2x'	'3/4', '4/5', '5/6', '8/9', '9/10'
32	's2x'	'32/45', '11/15', '7/9', '2/3'
64	's2x'	'11/15', '7/9', '4/5', '5/6', '128/180'
128	's2x'	'3/4', '7/9'
256	's2x'	'32/45', '3/4', '116/180', '20/30', '124/180', '22/30'

For more information, refer to Tables 9 and 10 in the DVB-S2 standard, [1], and Table 17a in the DVB-S2X standard, [2].

Dependencies

This input argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `char` | `string`

`frameLength` — Frame length

`'normal'` (default) | `'short'`

Frame length, specified as `'normal'` or `'short'`. `frameLength` and `codeIDF` are used to determine the modulation parameters.

Dependencies

This input argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = dvbsapskmod(x,M,std,'InputType','bit','OutputDataType','single');`

`InputType` — Input type

`'integer'` (default) | `'bit'`

Input type, specified as the comma-separated pair consisting of `'InputType'` and either `'integer'` or `'bit'`. To use `'integer'`, the input signal must consist of integer values from `0` to (`M` – 1). To use `'bit'`, the input signal must contain binary values and the number of rows must be an integer multiple of $\log_2(M)$.

Data Types: `char` | `string`

`UnitAveragePower` — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the function scales the constellation based on specifications in the relevant standard, as described in [1] and [2].

Note When `stdSuffix` is set to `'sh'`, the constellation always has unit average power.

Dependencies

This name-value pair argument applies only when `stdSuffix` is set to `'s2'` or `'s2x'`.

Data Types: `logical`

`OutputDataType` — Output data type

`'double'` (default) | `'single'`

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and either `'double'` or `'single'`.

Data Types: `char` | `string`

`PlotConstellation` — Option to plot constellation
`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

y — Modulated signal
scalar | vector | matrix

Modulated signal, returned as a complex scalar, vector, or matrix. The dimensions of `y` depend on the specified `'InputType'` value.

`'InputType'` Value	Dimensions of `y`
`'integer'`	`y` has the same dimensions as input `x`.
`'bit'`	The number of rows in `y` equals the number of rows in `x` divided by $\log_2$(`M`).

Data Types: `double` | `single`

More About

DVB-S2/S2X/SH

Digital video broadcasting (DVB) standards specify S2, S2X, and SH standard-specific amplitude phase shift keying (APSK) modulation. For further information on the DVB-S2/S2X/SH standards, see specified in [1], [2], and [3], respectively.

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskmod` | `dvbsapskdemod` | `genqammod` | `mil188qammod` | `pskmod` | `qammod`

Objects

`comm.GeneralQAMModulator` | `comm.PSKModulator`

Introduced in R2018a

encode

Block encoder

Syntax

```
code = encode(msg,n,k)
code = encode(msg,n,k,codingMethod,prim_poly)
code = encode(msg,n,k,codingMethod,genmat)
code = encode(msg,n,k,codingMethod,genpoly)
[code,added] = encode( ___ )
```

Description

`code = encode(msg,n,k)` encodes message, `msg`, using the Hamming encoding method with codeword length, `n`, and message length, `k`. The value of `n` must be calculated for an integer, *m*, such that $m \geq 2$. The values of `n` and `k` are calculated as 2^m–1 and `n`–*m*, respectively.

`code = encode(msg,n,k,codingMethod,prim_poly)` encodes `msg` using `codingMethod` as the Hamming encoding method, and `prim_poly` as the primitive polynomial. The value of `n` must be calculated for an integer, m≥2.

`code = encode(msg,n,k,codingMethod,genmat)` encodes `msg` using `codingMethod` as the linear block encoding method and `genmat` as the generator matrix. The value of `n` must be calculated for an integer, m≥2.

`code = encode(msg,n,k,codingMethod,genpoly)` encodes `msg` using `codingMethod` as the systematic cyclic code and `genpoly`, as the generator polynomial. The value of `n` must be calculated for an integer, m≥2.

`[code,added] = encode( ___ )` returns the additional variable `added`. `added` denotes the number of zeros appended at the end of the message matrix before encoding. You can specify any of the input argument combinations from the previous syntaxes.

Examples

Encode and Decode Message with Hamming Code

Set the values of the codeword length and message length.

```
n = 15; % Codeword length
k = 11; % Message length
```

Create a random binary message with length equal to the message length.

```
data = randi([0 1],k,1);
```

Encode the message.

```
encData = encode(data,n,k,'hamming/binary');
```

Corrupt the encoded message sequence by introducing an error in the fourth bit.

```
encData(4) = ~encData(4);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'hamming/binary');
numerr = biterr(data,decData)
```

```
numerr = 0
```

Encode and Decode Message with Linear Block Code

Set the values of codeword length and message length.

```
n = 7; % Codeword length
k = 3; % Message length
```

Create a random binary message with length equal to the message length.

```
data = randi([0 1],k,1);
```

Create a cyclic generator polynomial. Then, create a parity-check matrix and convert it into a generator matrix.

```
pol = cyclpoly(n,k);
parmat = cyclgen(n,pol);
genmat = gen2par(parmat);
```

Encode the message sequence by using the generator matrix.

```
encData = encode(data,n,k,'linear/binary',genmat);
```

Corrupt the encoded message sequence by introducing an error in the third bit.

```
encData(3) = ~encData(3);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'linear/binary',genmat);
```

```
Single-error patterns loaded in decoding table.  8 rows remaining.
2-error patterns loaded.  1 rows remaining.
3-error patterns loaded.  0 rows remaining.
```

```
numerr = biterr(data,decData)
```

```
numerr = 0
```

Encode and Decode Message with Cyclic Block Code

Set the values of the codeword length and message length.

```
n = 15; % Codeword length
k = 5; % Message length
```

Create a random binary message with length equal to the message length.

```
data = randi([0 1],k,1);
```

Create a generator polynomial for a cyclic code. Create a parity-check matrix by using the generator polynomial.

```
genpoly = cyclpoly(n,k);
parmat = cyclgen(n,genpoly);
```

Create a syndrome decoding table by using the parity-check matrix.

```
trt = syndtable(parmat);
```

```
Single-error patterns loaded in decoding table.  1008 rows remaining.
2-error patterns loaded.  918 rows remaining.
3-error patterns loaded.  648 rows remaining.
4-error patterns loaded.  243 rows remaining.
5-error patterns loaded.  0 rows remaining.
```

Encode the data by using the generator polynomial.

```
encData = encode(data,n,k,'cyclic/binary',genpoly);
```

Corrupt the encoded message sequence by introducing errors in the first, second, fourth and seventh bits.

```
encData(1) = ~encData(1);
encData(2) = ~encData(2);
encData(4) = ~encData(4);
```

Decode the corrupted sequence. Observe that the decoder has correctly recovered the message.

```
decData = decode(encData,n,k,'cyclic/binary',genpoly,trt);
numerr = biterr(data,decData)
```

```
numerr = 0
```

Input Arguments

`msg` — Input messages

binary column or row vector | binary matrix with k columns | column or row vector of integers in the range [0, 2^{k}–1]

Input messages, specified as one of these options:

- Binary column or row vector with k columns
- Binary matrix with k columns
- Column or row vector of k columns and having integers in the range [0, 2^{k}–1]

Example: `msg = [0 1 1 0, 0 1 0 1, 1 0 0 1]` specifies a binary row vector for k=4.

Example: `msg = [0 1 1 0; 0 1 0 1; 1 0 0 1]` specifies a binary matrix for k=4.

Example: `msg = [6, 10, 9]` specifies a row vector of integers for k=4.

Data Types: `double`

n — Codeword length

positive integer

Codeword length, specified as a positive integer. The function calculates this value as 2^*m*–1, where *m* must be greater than or equal to 2.

Data Types: double

k — Message length

positive integer

Message length, specified as a positive integer. The function calculates this value as n–*m*, where *m* must be greater than or equal to 2.

Data Types: double

codingMethod — Error coding method and format

'hamming/binary' (default) | 'hamming/decimal' | 'linear/binary' | ...

Error coding method and format, specified as one of these:

- 'hamming/binary'
- 'hamming/decimal'
- 'linear/binary'
- 'linear/decimal'
- 'cyclic/binary'
- 'cyclic/decimal'

Data Types: char | string

prim_poly — Primitive polynomial

gfprimdf(n-k) (default) | binary row vector | character vector | string scalar | positive integer

Primitive polynomial, specified as one of these options:

- Binary row vector — This vector gives coefficients of prim_poly in the order of ascending powers.
- Character vector or a string scalar — This value defines prim_poly in textual representation. For more information, see polynomial character vector.
- Positive integer — This value defines prim_poly in the range [$2^m + 1$, $2^{m+1} - 1$].

For more information about default primitive polynomials, see “Default Primitive Polynomials” on page 2-400. For more information about the representation of primitive polynomials, see “Primitive Polynomials and Element Representations”.

Data Types: double | char | string

genmat — Generator matrix

k-by-n numeric matrix

Generator matrix, specified as a k-by-n numeric matrix.

Data Types: double

genpoly — Generator polynomial
cyclpoly(n-k) (default) | binary row vector | character vector | string scalar

Generator polynomial, specified as a polynomial character vector or a row vector that gives the coefficients in order of ascending powers of the binary generator polynomial. The value of genpoly for an [n, k] cyclic code must have degree n–k and divide x^n-1, where x is an identifier.

Data Types: char | string

Output Arguments

code — Output code
binary column or row vector | binary matrix with n columns | column or row vector of integers in the range [0, 2^n–1].

Output code, returned as one of the options in this table. The value and dimension of code depends on the value and dimension of the “msg” on page 2-0 and the input message format according to this table:

msg Value	Input Message Format	code Value
Binary column or row vector	binary	Binary column or row vector
Binary matrix with k columns	binary	Binary matrix with n columns
Column or row vector of integers in the range [0, 2^k–1]	decimal	Column or row vector of integers in the range [0, 2^n–1]

added — Additional variable
nonnegative integer

Additional variable, returned as the number of zeros that were appended at the end of the message matrix before encoding for the matrix to have the appropriate size. The size of the message matrix depends on the n, k, and msg and the encoding method.

Algorithms

Depending on the error-correction coding method, the encode function relies on lower-level functions such as hammgen and cyclgen.

See Also

cyclgen | cyclpoly | decode | hammgen

Topics
“Block Codes”

Introduced before R2006a

equalize

(To be removed) Equalize signal using equalizer object

Note will be removed in a future release. Use comm.LinearEqualizer or comm.DecisionFeedback instead.

Syntax

```
y = equalize(eqobj,x)
y = equalize(eqobj,x,trainsig)
[y,yd] = equalize(...)
[y,yd,e] = equalize(...)
```

Description

y = equalize(eqobj,x) processes the baseband signal vector x with equalizer object eqobj and returns the equalized signal vector y. At the end of the process, eqobj contains updated state information such as equalizer weight values and input buffer values. To construct eqobj, use the lineareq or dfe function. The equalize function assumes that the signal x is sampled at nsamp samples per symbol, where nsamp is the value of the nSampPerSym property of eqobj. For adaptive algorithms other than CMA, the equalizer adapts in decision-directed mode using a detector specified by the SigConst property of eqobj. The delay of the equalizer is (eqobj.RefTap-1)/eqobj.nSampPerSym.

Note that (eqobj.RefTap-1) must be an integer multiple of nSampPerSym. For a fractionally-spaced equalizer, the taps are spaced at fractions of a symbol period. The reference tap pertains to training symbols, and thus, must coincide with a whole number of symbols (i.e., an integer number of samples per symbol). eqobj.RefTap=1 corresponds to the first symbol, eqobj.RefTap=nSampPerSym+1 to the second, and so on. Therefore (eqobj.RefTap-1) must be an integer multiple of nSampPerSym.

If eqobj.ResetBeforeFiltering is 0, equalize uses the existing state information in eqobj when starting the equalization operation. As a result, equalize(eqobj,[x1 x2]) is equivalent to [equalize(eqobj,x1) equalize(eqobj,x2)]. To reset eqobj manually, apply the reset function to eqobj.

If eqobj.ResetBeforeFiltering is 1, equalize resets eqobj before starting the equalization operation, overwriting any previous state information in eqobj.

y = equalize(eqobj,x,trainsig) initially uses a training sequence to adapt the equalizer. After processing the training sequence, the equalizer adapts in decision-directed mode. The vector length of trainsig must be less than or equal to length(x)-(eqobj.RefTap-1)/eqobj.nSampPerSym.

[y,yd] = equalize(...) returns the vector yd of detected data symbols.

[y,yd,e] = equalize(...) returns the result of the error calculation. For adaptive algorithms other than CMA, e is the vector of errors between y and the reference signal, where the reference signal consists of the training sequence or detected symbols.

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r,x(1:100));
yNew = eqNew(r,x(1:100));
```

Use RLS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: 0.1000
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. For the `comm.LinearEqualizer` object, set the initial inverse correlation matrix to `eye(5)*0.2`.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1, ...
    'InitialInverseCorrelationMatrix',eye(5)*0.2)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: [5×5 double]
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Use CMA Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,cma(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'Constant Modulus'
  nWeights: 5
  nSampPerSym: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [1 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','CMA','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'CMA'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputSamplesPerSymbol: 1
  AdaptWeightsSource: 'Property'
  AdaptWeights: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Compatibility Considerations

`equalize` will be removed

Warns starting in R2020a

equalize will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead. For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-336.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

evdoForwardReferenceChannels

Define 1xEV-DO forward reference channel

Syntax

```
cfg = evdoForwardReferenceChannels(wv)
cfg = evdoForwardReferenceChannels(wv,numpackets)
```

Description

`cfg = evdoForwardReferenceChannels(wv)` returns a structure, `cfg`, that defines 1xEV-DO forward link parameters given the input waveform identifier, `wv`. Pass this structure to the `evdoForwardWaveformGenerator` function to generate a forward link reference channel waveform.

For all syntaxes, `evdoForwardReferenceChannels` creates a configuration structure that is compliant with the cdma2000 high data rate packet specification, [1].

`cfg = evdoForwardReferenceChannels(wv,numpackets)` specifies the number of packets to be generated.

Examples

Generate 1xEV-DO Release 0 Forward Link Waveform

Create a configuration structure for a Release 0 channel having a 921.6 kbps data rate and transmitted over two slots.

```
config = evdoForwardReferenceChannels('Rel0-921600-2');
```

Display the number of slots and the data rate.

```
config.PacketSequence
```

```
ans = struct with fields:
    MACIndex: 0
    DataRate: 921600
    NumSlots: 2
```

Generate the complex waveform using the associated waveform generator function, `evdoForwardWaveformGenerator`.

```
wv = evdoForwardWaveformGenerator(config);
```

Generate 1xEV-DO Revision A Forward Link Waveform

Create a structure to transmit a Revision A 1xEV-DO channel consisting of three 1024-bit packets transmitted over 2 slots with a 64-bit preamble length.

```
config = evdoForwardReferenceChannels('RevA-1024-2-64',3);
```

Verify that the function created a 1-by-3 structure array. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans=1×3 struct array with fields:
    MACIndex
    PacketSize
    NumSlots
    PreambleLength
```

Examine the first structure element to verify the packet size, number of slots, and preamble length match what you specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
          MACIndex: 0
        PacketSize: 1024
          NumSlots: 2
    PreambleLength: 64
```

Generate the waveform.

```
wv = evdoForwardWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector.

Parameter Field	Values	Description
wv	'Rel0-38400-16' | 'Rel0-76800-8' | 'Rel0-153600-4' | 'Rel0-307200-2' | 'Rel0-307200-4' | 'Rel0-614400-1' | 'Rel0-614400-2' | 'Rel0-921600-2' | 'Rel0-1228800-1' | 'Rel0-1228800-2' | 'Rel0-1843200-1' | 'Rel0-2457600-1'	Character vector representing the 1xEV-DO Release 0 reference channel with data rate in bps and number of slots. For example, you can specify 'Rel0-153600-4' to create a structure that represents a reference channel with a 153,600 bps data rate and uses four slots.

Parameter Field	Values	Description
	`'RevA-128-1-64'` | `'RevA-128-2-128'` | `'RevA-128-4-256'` | `'RevA-128-4-1024'` | `'RevA-128-8-512'` | `'RevA-256-1-64'` | `'RevA-256-2-128'` | `'RevA-256-4-256'` | `'RevA-256-4-1024'` | `'RevA-256-8-512'` | `'RevA-256-16-1024'` | `'RevA-512-1-64'` | `'RevA-512-2-64'` | `'RevA-512-2-128'` | `'RevA-512-4-128'` | `'RevA-512-4-256'` | `'RevA-512-4-1024'` | `'RevA-512-8-512'` | `'RevA-512-16-1024'` | `'RevA-1024-1-64'` | `'RevA-1024-2-64'` | `'RevA-1024-2-128'` | `'RevA-1024-4-128'` | `'RevA-1024-4-256'` | `'RevA-1024-8-512'` | `'RevA-1024-16-1024'` | `'RevA-2048-1-64'` | `'RevA-2048-2-64'` | `'RevA-2048-4-128'` | `'RevA-3072-1-64'` | `'RevA-3072-2-64'` `'RevA-4096-1-64'` | `'RevA-4096-2-64'` | `'RevA-5120-1-64'` | `'RevA-5120-2-64'`	Character vector representing the 1xEV-DO Revision A reference channel with the packet size in bits, the number of slots, and the preamble length in chips. For example, you can specify `'RevA-256-1-64'` to create a reference channel having a 256-bit packet, transmitted in one slot, with a 64-bit preamble length.

Example: `'Rel0-614400-2'`

Example: `'RevA-4096-2-64'`

Data Types: `char`

`numpackets` — Number of packets
`1` (default) | positive integer scalar

Number of packets, specified as a positive integer.

Example: 4

Data Types: `double`

Output Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator
structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO
PNOffset	Nonnegative scalar integer [0, 511]	PN offset of the base station
IdleSlotsWithControl	'Off' | 'On'	Include idle slots with control channels
EnableControl	'Off' | 'On'	Enable control signaling
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer [1, 8]	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000short' | 'Custom' | 'Off'	Select filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when the FilterType field is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
PacketSequence	Structure	See **PacketSequence substructure**.
PacketDataSources	Structure	See **PacketDataSources substructure**.

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet
	Release 0	
DataRate	38400 | 76800 | 153600 | 307200 | 614400 | 921600 | 1228800 | 1843200 | 2457600	Data rate (bps)
NumSlots	Positive scalar integer	Number of slots
	Revision A	
PacketSize	128 | 256 | 512 | 1024 | 2048 | 3072 | 4096 | 5120	Packet size (bits)
NumSlots	1 | 2 | 4 | 8 | 16	Number of slots
PreambleLength	64 | 128 | 256 | 512 | 1024	Preamble length (chips)

PacketDataSources Substructure

Include a `PacketDataSources` substructure in the `cfg` structure to define a set of matching data sources for each MAC index. The `PacketDataSources` substructure contains these fields.

Parameter Field	Values	Description
`MACIndex`	Positive scalar integer	MAC index associated with the packet
`DataSource`	Cell array, `{'PN Type', RN Seed}` or binary vector. Standard PN sequence options are `'PN9'`, `'PN15'`, `'PN23'`, `'PN9-ITU'`, and `'PN11'`.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
`EnableCoding`	`'Off'` | `'On'`	Enable error correction coding

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*.

See Also

`evdoForwardWaveformGenerator` | `evdoReverseReferenceChannels`

Introduced in R2015b

evdoForwardWaveformGenerator

Generate 1xEV-DO forward link waveform

Syntax

```
waveform = evdoForwardWaveformGenerator(cfg)
```

Description

`waveform = evdoForwardWaveformGenerator(cfg)` returns the 1xEV-DO forward link `waveform` as defined by the parameter configuration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties the function uses to generate a 1xEV-DO waveform. You can generate `cfg` by using the `evdoForwardReferenceChannels` function.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all parameter combinations are supported. To ensure that the input argument is valid, use the `evdoForwardReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate 1xEV-DO Revision A Forward Link Waveform

Create a structure to transmit a Revision A 1xEV-DO channel consisting of three 1024-bit packets transmitted over 2 slots with a 64-bit preamble length.

```
config = evdoForwardReferenceChannels('RevA-1024-2-64',3);
```

Verify that the function created a 1-by-3 structure array. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans=1×3 struct array with fields:
    MACIndex
    PacketSize
    NumSlots
    PreambleLength
```

Examine the first structure element to verify the packet size, number of slots, and preamble length match what you specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
          MACIndex: 0
```

```
        PacketSize: 1024
          NumSlots: 2
    PreambleLength: 64
```

Generate the waveform.

```
wv = evdoForwardWaveformGenerator(config);
```

Generate 1xEV-DO Forward Link Waveform with Custom Filter

Create a structure to generate two packets of a 1.8 Mbps Release 0 channel.

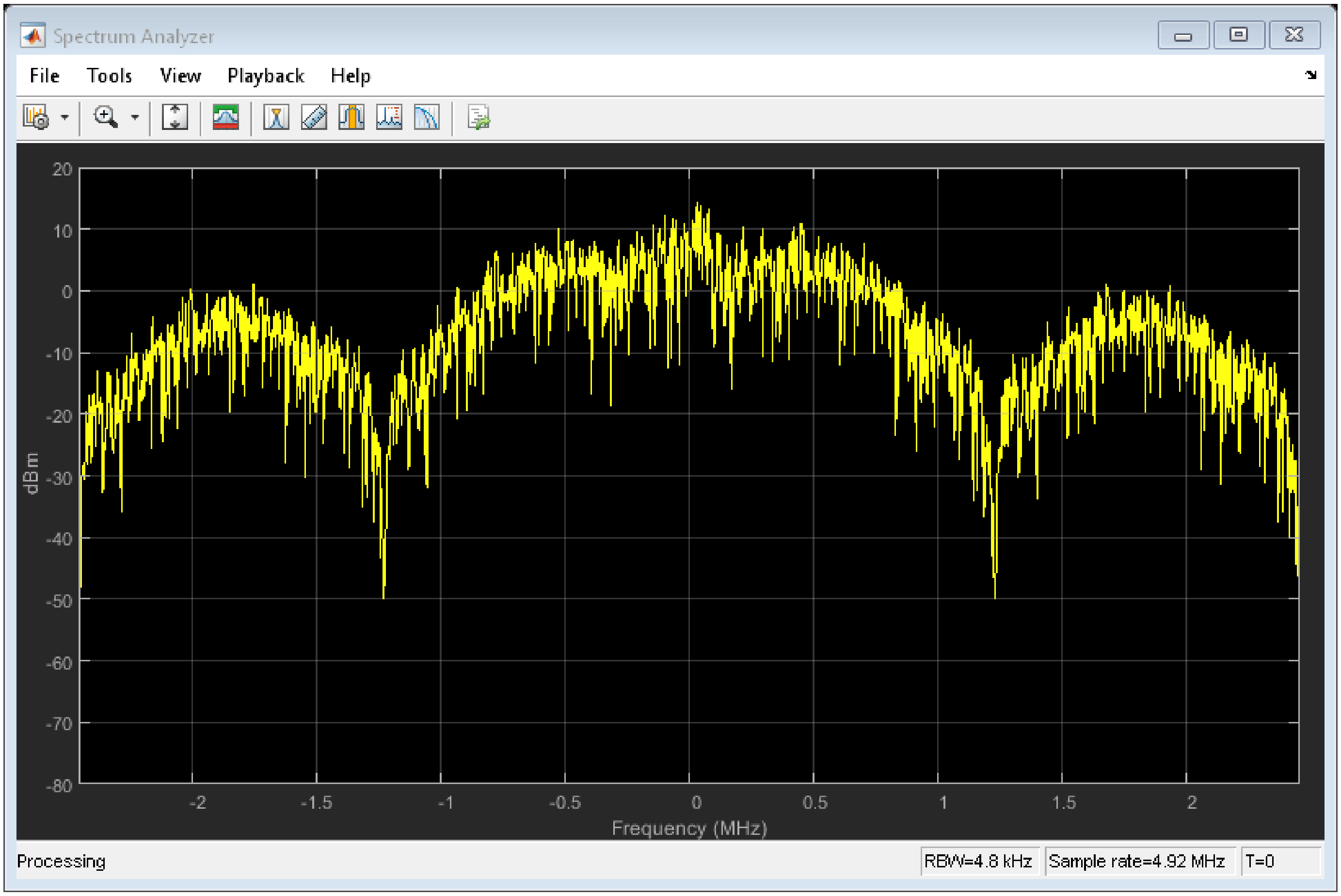
```
config = evdoForwardReferenceChannels('Rel0-1843200-1',2);
```

Calculate the sample rate of the waveform.

```
fs = 1.2288e6 * config.OversamplingRatio;
```

Disable the internal filter of the `evdoForwardWaveformGenerator` function. Generate the 1xEV-DO waveform. Plot the spectrum of the waveform.

```
config.FilterType = 'off';
wv = evdoForwardWaveformGenerator(config);

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Create a lowpass FIR filter with a 500 kHz passband, a 750 kHz stopband, and a stopband attenuation of 60 dB.

```
d = designfilt('lowpassfir', ...
    'PassbandFrequency',500e3, ...
    'StopbandFrequency',750e3, ...
    'StopbandAttenuation',60, ...
    'SampleRate',fs);
```

Change the filter type to `'Custom'` and specify the coefficients from the digital filter, `d`.

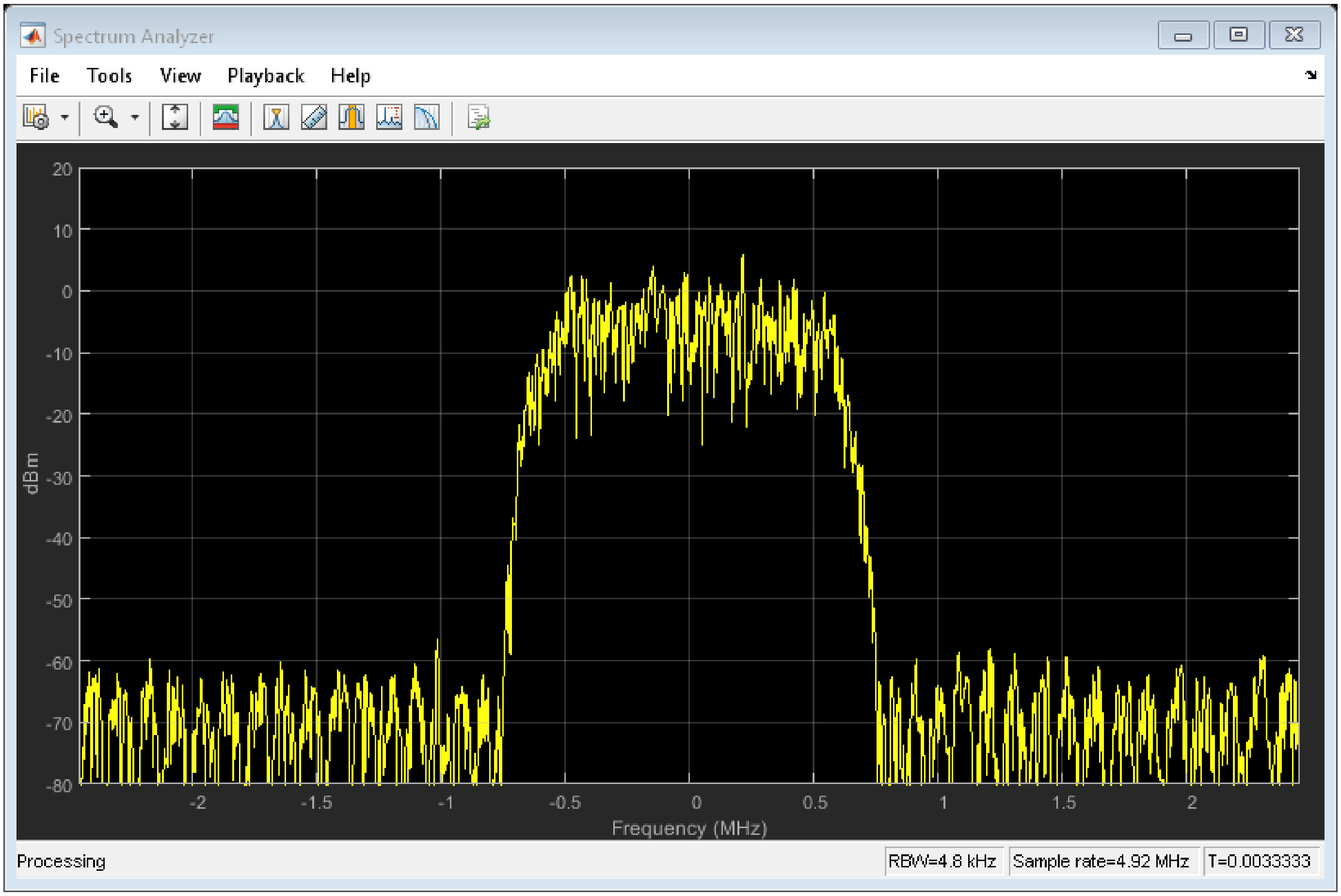
```
config.FilterType = 'Custom';
config.CustomFilterCoefficients = d.Coefficients;
```

Generate the waveform using the custom filter coefficients.

```
wv = evdoForwardWaveformGenerator(config);
```

Plot the spectrum of the filtered 1xEV-DO waveform.

```
step(sa,wv)
```



The filter attenuates the waveform by 60 dB for frequencies outside of ± 750 kHz.

Input Arguments

`cfg` — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
`Release`	`'Release0'` | `'RevisionA'`	1xEV-DO
`PNOffset`	Nonnegative scalar integer [0, 511]	PN offset of the base station
`IdleSlotsWithControl`	`'Off'` | `'On'`	Include idle slots with control channels
`EnableControl`	`'Off'` | `'On'`	Enable control signaling
`NumChips`	Positive scalar integer	Number of chips in the waveform
`OversamplingRatio`	Positive scalar integer [1, 8]	Oversampling ratio at output

Parameter Field	Values	Description
FilterType	'cdma2000Long' | 'cdma2000short' | 'Custom' | 'Off'	Select filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when the FilterType field is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
PacketSequence	Structure	See **PacketSequence substructure**.
PacketDataSources	Structure	See **PacketDataSources substructure**.

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet
	Release 0	
DataRate	38400 | 76800 | 153600 | 307200 | 614400 | 921600 | 1228800 | 1843200 | 2457600	Data rate (bps)
NumSlots	Positive scalar integer	Number of slots
	Revision A	
PacketSize	128 | 256 | 512 | 1024 | 2048 | 3072 | 4096 | 5120	Packet size (bits)
NumSlots	1 | 2 | 4 | 8 | 16	Number of slots
PreambleLength	64 | 128 | 256 | 512 | 1024	Preamble length (chips)

PacketDataSources Substructure

Include a PacketDataSources substructure in the cfg structure to define a set of matching data sources for each MAC index. The PacketDataSources substructure contains these fields.

Parameter Field	Values	Description
MACIndex	Positive scalar integer	MAC index associated with the packet
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

Parameter Field	Values	Description
EnableCoding	'Off' | 'On'	Enable error correction coding

Output Arguments

waveform — Modulated baseband waveform comprising the primary physical channels
complex vector array

Modulated baseband waveform comprising the primary cdma2000 physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*.

See Also

cdma2000ForwardReferenceChannels | cdma2000ReverseWaveformGenerator

Introduced in R2015b

evdoReverseReferenceChannels

Define 1xEV-DO reverse reference channel

Syntax

```
cfg = evdoReverseReferenceChannels(wv)
cfg = evdoReverseReferenceChannels(wv,numpackets)
```

Description

`cfg = evdoReverseReferenceChannels(wv)` returns a structure, `cfg`, that defines 1xEV-DO reverse link parameters given the input waveform identifier, `wv`. Pass this structure to the `evdoReverseWaveformGenerator` function to generate a reverse link reference channel waveform.

For all syntaxes, `evdoReverseReferenceChannels` creates a structure that is compliant with the cdma2000 high data rate packet specification,[1].

`cfg = evdoReverseReferenceChannels(wv,numpackets)` specifies the number of packets to be generated.

Examples

Generate 1xEV-DO Reverse Channel Waveform

Create a structure to generate a Release 0, 1xEV-DO waveform having a 19.2 kbps data rate.

```
config = evdoReverseReferenceChannels('Rel0-19200');
```

Verify that the packet has a data rate of 19.2 kbps.

```
config.PacketSequence.DataRate
```

```
ans = 19200
```

Generate the complex waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Generate 1xEV-DO Revision A Reverse Link Waveform

Create a structure for a Revision A 1xEV-DO channel having 2048-bit packets, transmitted in 12 slots. Specify that five packets are transmitted.

```
config = evdoReverseReferenceChannels('RevA-2048-12',5);
```

Verify that a 1-by-5 structure array is created. Each element in the structure array corresponds to a data packet.

```
config.PacketSequence
```

```
ans=1×5 struct array with fields:
    Power
    DataSource
    EnableCoding
    PayloadSize
    NumSlots
    DataRate
```

Examine the first structure element to verify the packet size and the number of slots are as specified in the function call.

```
config.PacketSequence(1)
```

```
ans = struct with fields:
           Power: 0
      DataSource: {'PN9'  [1]}
    EnableCoding: 'On'
     PayloadSize: 2048
        NumSlots: 12
        DataRate: 102400
```

Generate the waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Input Arguments

wv — Waveform identification

character vector

Waveform identification of the reference channel, specified as a character vector.

Parameter Field	Values	Description
wv	'Rel0-9600' | 'Rel0-19200' | 'Rel0-38400' | 'Rel0-76800' | 'Rel0-153600'	Character vector representing the 1xEV-DO Release 0 data rate in bps. For example, you can specify 'Rel0-153600' to create a structure corresponding to a Release 0 reference channel having a 153,600 bps data rate.

Parameter Field	Values	Description
	'RevA-128-4' | 'RevA-128-8' | 'RevA-128-12' | 'RevA-128-16' | 'RevA-256-4' | 'RevA-256-8' | 'RevA-256-12' | 'RevA-256-16' | 'RevA-512-4' | 'RevA-512-8' | 'RevA-512-12' | 'RevA-512-16' | 'RevA-768-4' | 'RevA-768-8' | 'RevA-768-12' | 'RevA-768-16' | 'RevA-1024-4' | 'RevA-1024-8' | 'RevA-1024-12' | 'RevA-1024-16' | 'RevA-1536-4' | 'RevA-1536-8' | 'RevA-1536-12' | 'RevA-1536-16' | 'RevA-2048-4' | 'RevA-2048-8' | 'RevA-2048-12' | 'RevA-2048-16' | 'RevA-3072-4' | 'RevA-3072-8' | 'RevA-3072-12' | 'RevA-3072-16' | 'RevA-4096-4' | 'RevA-4096-8' | 'RevA-4096-12' | 'RevA-4096-16' | 'RevA-6144-4' | 'RevA-6144-8' | 'RevA-6144-12' | 'RevA-6144-16' | 'RevA-8192-4' | 'RevA-8192-8' | 'RevA-8192-12' | 'RevA-8192-16' | 'RevA-12288-4' | 'RevA-12288-8' | 'RevA-12288-12' | 'RevA-12288-16'	Character vector representing the 1xEV-DO Revision A packet size in bits and the number of slots. For example, you can specify 'RevA-256-4' to create a structure corresponding to a Revision A reference channel having 256-bit packets and transmitted in four slots.

Example: 'Rel0-38400'

Example: 'RevA-3072-12'

Data Types: char

numpackets — Number of packets

1 (default) | positive integer scalar

Number of packets, specified as a positive integer.

Example: 2

Data Types: double

Output Arguments

cfg — Configuration of the parameters and channels used by the waveform generator

structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO applicable standard
LongCodeMaskI	42-bit binary number	Long code identifier for in-phase channel
LongCodeMaskQ	42-bit binary number	Long code identifier for quadrature channel
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Custom' | 'Off'	Specify the filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when FilterType is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
ACKChannel	Structure	See **ACKChannel substructure**.
PilotChannel	Structure	See **PilotChannel substructure**.
AuxPilotChannel	Not present or structure	See **AuxPilotChannel substructure**.
PacketSequence	Structure	See **PacketSequence substructure**.

ACKChannel Substructure

Include the ACKChannel substructure in the cfg structure to specify the acknowledgment channel. The ACKChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

PilotChannel Substructure

Include the PilotChannel substructure in the cfg structure to specify the pilot channel. The PilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

AuxPilotChannel Substructure

Include the AuxPilotChannel substructure in the cfg structure to specify the auxiliary pilot channel, which is available only for Revision A. The AuxPilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
Power	Real scalar	MAC index associated with the packet
EnableCoding	'Off' | 'On'	Enable error correction coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
Release 0		
DataRate	9600 | 19200 | 38400 | 76800 | 153600	Data rate (bps)
Revision A		
PacketSize	128 | 256 | 512 | 768 | 1024 | 1536 | 2048 | 3072 | 4096 | 6144 | 8192 | 12288	Packet size (bits)
NumSlots	4 | 8 | 12 | 16	Number of slots

Data Types: struct

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*.

See Also

evdoForwardReferenceChannels | evdoReverseWaveformGenerator

Introduced in R2015b

evdoReverseWaveformGenerator

Generate 1xEV-DO reverse link waveform

Syntax

```
waveform = evdoReverseWaveformGenerator(cfg)
```

Description

`waveform = evdoReverseWaveformGenerator(cfg)` returns the 1xEV-DO reverse link `waveform` as defined by the parameter configuration structure, `cfg`.

The top-level parameters and lower-level substructures of `cfg` specify the waveform and channel properties used by the function to generate a 1xEV-DO waveform. You can generate `cfg` by using the `evdoReverseReferenceChannels` function.

Note The tables herein list the allowable values for the top-level parameters and substructure fields. However, not all parameter combinations are supported. To ensure that the input argument is valid, use the `evdoReverseReferenceChannels` function. If you input the structure fields manually, consult [1] to ensure that the input parameter combinations are permitted.

Examples

Generate 1xEV-DO Reverse Channel Waveform

Create a structure to generate a Release 0, 1xEV-DO waveform having a 19.2 kbps data rate.

```
config = evdoReverseReferenceChannels('Rel0-19200');
```

Verify that the packet has a data rate of 19.2 kbps.

```
config.PacketSequence.DataRate
```

```
ans = 19200
```

Generate the complex waveform.

```
wv = evdoReverseWaveformGenerator(config);
```

Generate 1xEV-DO Reverse Link Waveform with Custom Filter

Create a structure to generate four packets of a Revision A channel having 768-bit packets transmitted over eight slots.

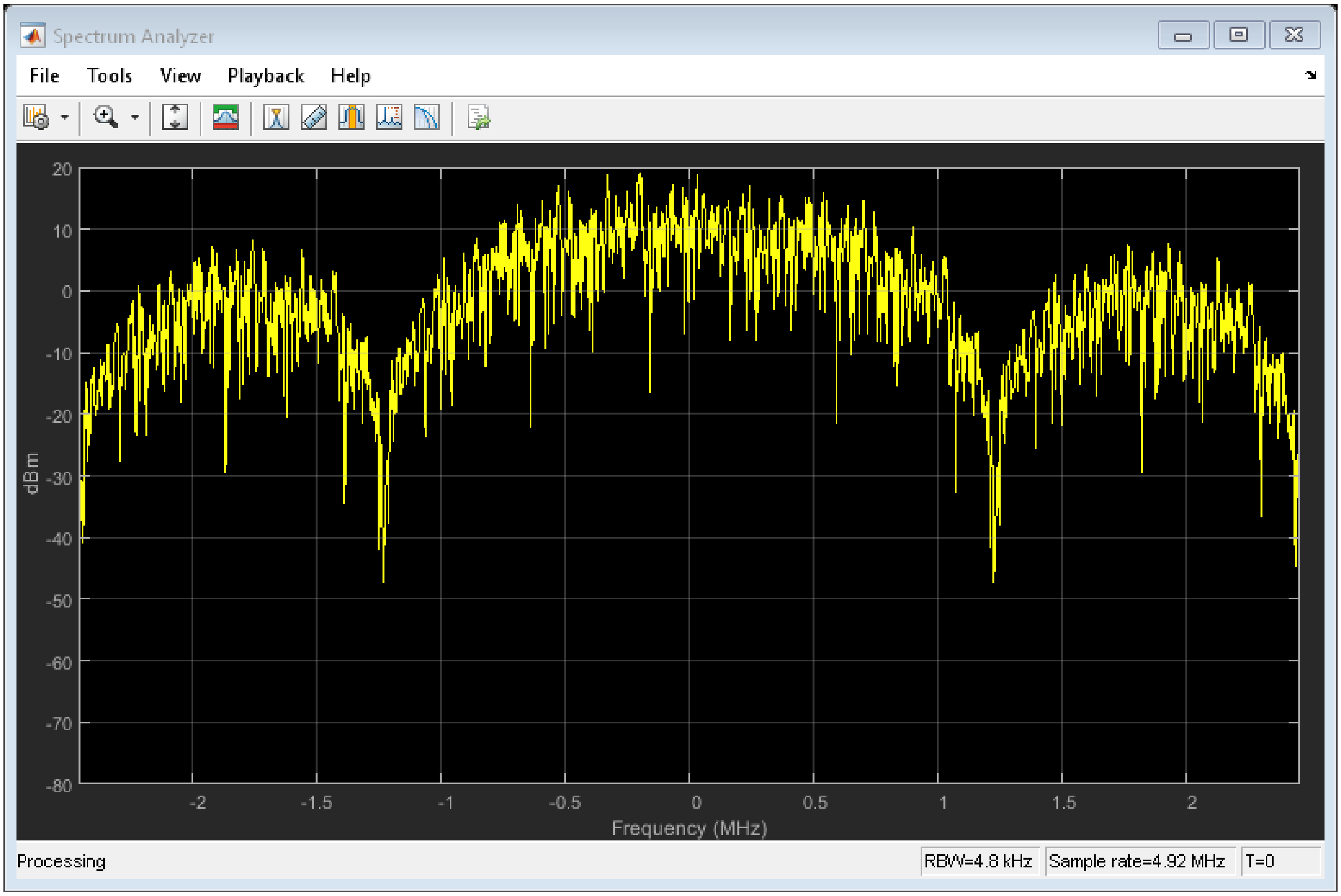
```
config = evdoReverseReferenceChannels('RevA-768-8',4);
```

Calculate the sample rate of the waveform.

```
fs = 1.2288e6 * config.OversamplingRatio;
```

Disable the internal filter of the `evdoReverseWaveformGenerator`. Generate the 1xEV-DO waveform. Plot the spectrum of the waveform.

```
config.FilterType = 'off';
wv = evdoReverseWaveformGenerator(config);

sa = dsp.SpectrumAnalyzer('SampleRate',fs);
step(sa,wv)
```



Create a lowpass FIR filter with a 500 kHz passband, a 750 kHz stopband, and a stopband attenuation of 60 dB.

```
d = designfilt('lowpassfir', ...
    'PassbandFrequency',500e3, ...
    'StopbandFrequency',750e3, ...
    'StopbandAttenuation',60, ...
    'SampleRate',fs);
```

Change the filter type to `'Custom'` and specify the coefficients from the digital filter, `d`.

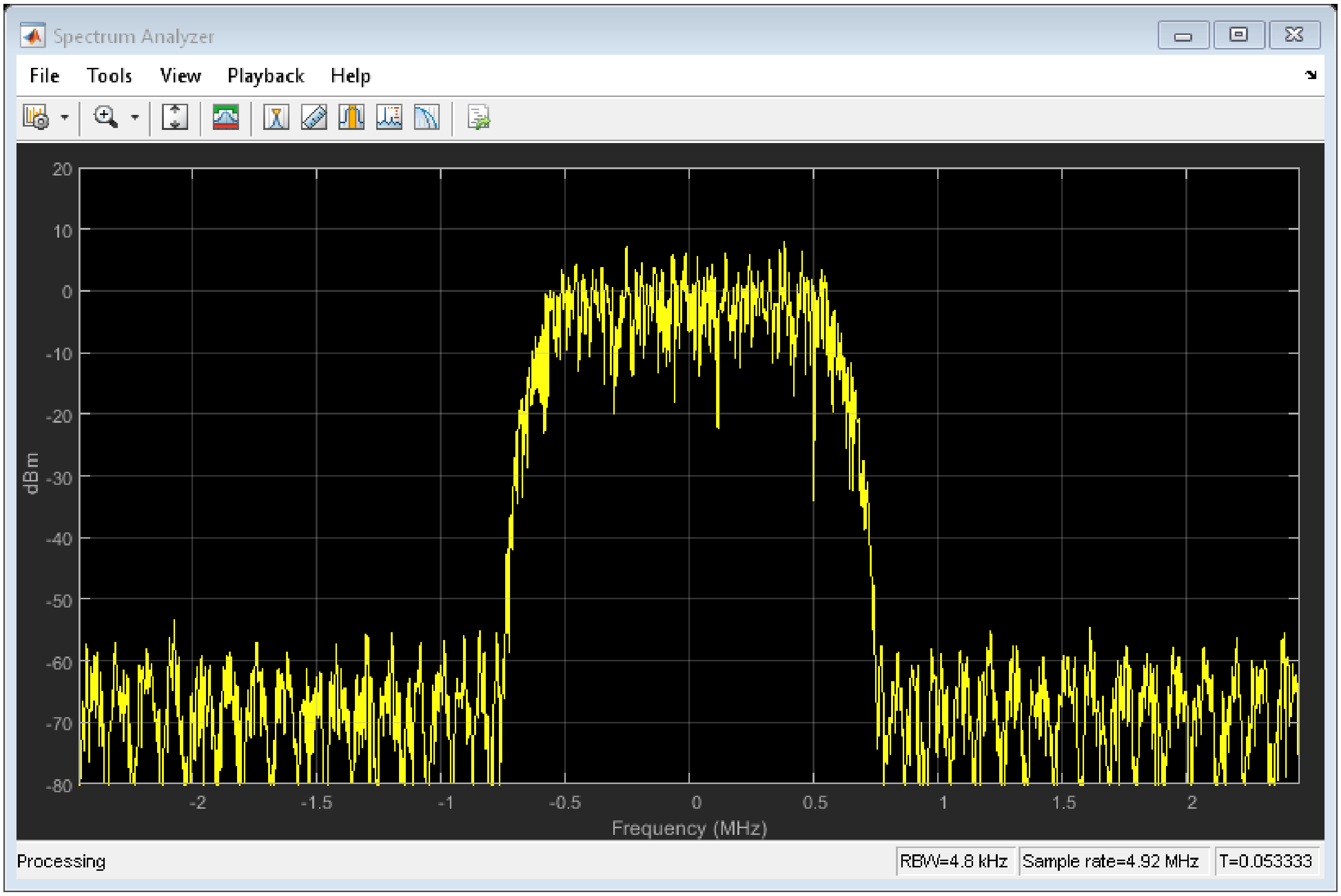
```
config.FilterType = 'Custom';
config.CustomFilterCoefficients = d.Coefficients;
```

Generate the waveform using the custom filter coefficients.

```
wv = evdoReverseWaveformGenerator(config);
```

Plot the spectrum of the filtered 1xEV-DO waveform.

```
step(sa,wv)
```



The filter attenuates the waveform by 60 dB for frequencies outside of ± 750 kHz.

Input Arguments

cfg — Configuration of the parameters and channels used by the waveform generator
structure

Configuration of the parameters and channels used by the waveform generator. The configuration structure is defined in these tables.

Top-Level Parameters and Substructures

Parameter Field	Values	Description
Release	'Release0' | 'RevisionA'	1xEV-DO applicable standard
LongCodeMaskI	42-bit binary number	Long code identifier for in-phase channel
LongCodeMaskQ	42-bit binary number	Long code identifier for quadrature channel

Parameter Field	Values	Description
NumChips	Positive scalar integer	Number of chips in the waveform
OversamplingRatio	Positive scalar integer	Oversampling ratio at output
FilterType	'cdma2000Long' | 'cdma2000Short' | 'Custom' | 'Off'	Specify the filter type or disable filtering
CustomFilterCoefficients	Real vector	Custom filter coefficients (applies when FilterType is set to 'Custom')
InvertQ	'Off' | 'On'	Negate the quadrature output
EnableModulation	'Off' | 'On'	Enable carrier modulation
ModulationFrequency	Nonnegative scalar integer	Carrier modulation frequency (applies when EnableModulation is 'On')
ACKChannel	Structure	See **ACKChannel substructure**.
PilotChannel	Structure	See **PilotChannel substructure**.
AuxPilotChannel	Not present or structure	See **AuxPilotChannel substructure**.
PacketSequence	Structure	See **PacketSequence substructure**.

ACKChannel Substructure

Include the ACKChannel substructure in the cfg structure to specify the acknowledgment channel. The ACKChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.

PilotChannel Substructure

Include the PilotChannel substructure in the cfg structure to specify the pilot channel. The PilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)

Parameter Fields	Values	Description
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

AuxPilotChannel Substructure

Include the AuxPilotChannel substructure in the cfg structure to specify the auxiliary pilot channel, which is available only for Revision A. The AuxPilotChannel substructure contains these fields.

Parameter Fields	Values	Description
Enable	'On' | 'Off'	Character vector to enable or disable the channel
Power	Real scalar	Channel power (dBW)
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
EnableCoding	'On' | 'Off'	Enable channel coding

PacketSequence Substructure

Include the PacketSequence substructure in the cfg structure to define a sequence of data packets for consecutive transmission. The PacketSequence substructure contains these fields.

Parameter Field	Values	Description
Power	Real scalar	MAC index associated with the packet
EnableCoding	'Off' | 'On'	Enable error correction coding
DataSource	Cell array, {'PN Type', RN Seed} or binary vector. Standard PN sequence options are 'PN9', 'PN15', 'PN23', 'PN9-ITU', and 'PN11'.	Data source. Specify a standard PN sequence with a random number seed or a custom vector.
Release 0		
DataRate	9600 | 19200 | 38400 | 76800 | 153600	Data rate (bps)
Revision A		

Parameter Field	Values	Description
`PacketSize`	`128` | `256` | `512` | `768` | `1024` | `1536` | `2048` | `3072` | `4096` | `6144` | `8192` | `12288`	Packet size (bits)
`NumSlots`	`4` | `8` | `12` | `16`	Number of slots

Output Arguments

`waveform` — Modulated baseband waveform comprising the physical channels
complex vector array

Modulated baseband waveform comprising the 1xEV-DO physical channels, returned as a complex vector array.

References

[1] 3GPP2 C.S0024–A v3.0. “cdma2000 High Rate Packet Data Air Interface Specification.” *3rd Generation Partnership Project 2*.

See Also

`evdoForwardWaveformGenerator` | `evdoReverseReferenceChannels`

Introduced in R2015b

eyediagram

Generate 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)` generates an eye diagram for signal `x`, plotting `n` samples in each trace. 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.

`eyediagram(x,n,period)` sets the labels on the horizontal axis to the range between –`period`/2 to `period`/2.

`eyediagram(x,n,period,offset)` specifies the offset for the eye diagram. The function assumes that the (`offset` + 1)th value of the signal and every `n`th value thereafter, occur at times that are integer multiples of `period`.

`eyediagram(x,n,period,offset,plotstring)` specifies plot attributes for the eye diagram.

`eyediagram(x,n,period,offset,plotstring,h)` generates the eye diagram in an existing figure whose handle is `h`.

Note Use of `hold on` to plot multiple signals in the same figure is not supported.

`h = eyediagram( ___ )` returns the handle to the figure that contains the eye diagram. You can specify any of the input argument combinations from the previous syntaxes.

Examples

Generate Eye Diagram of Filtered QPSK Signal

Generate an eyediagram of a filtered QPSK signal.

Generate random symbols. Apply QPSK modulation to get a modulated signal.

```
data = randi([0 3],1000,1);
modSig = pskmod(data,4,pi/4);
```

Specify the number of output samples per symbol parameter. Create a transmit filter object, `txfilter`.

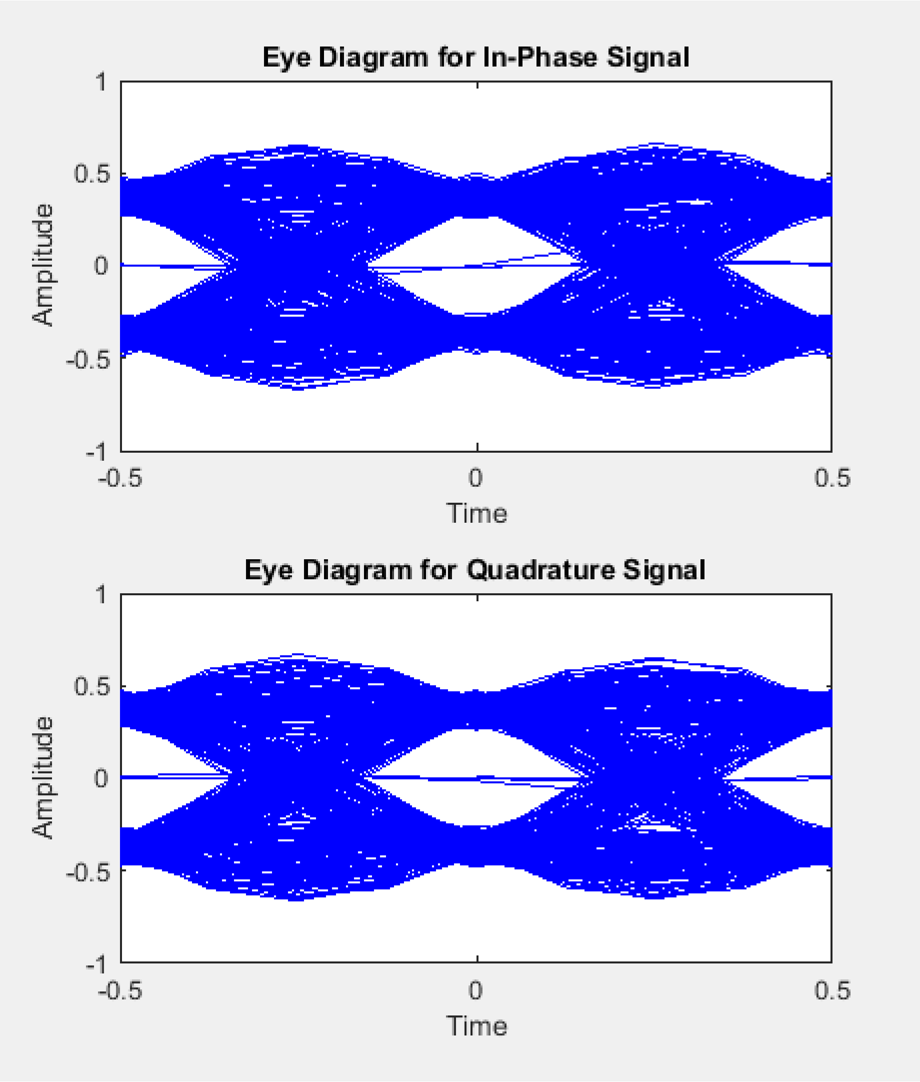
```
sps=4;
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps);
```

Filter the modulated signal `modSig`.

```
txSig = txfilter(modSig);
```

Display the eye diagram.

```
eyediagram(txSig,2*sps)
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix.

The interpretation of x and the number of plots depend on the shape and complexity of x.

- If x is a real-valued two-column matrix, the function 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-valued vector, the function 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-valued vector, the function interprets the vector as a real signal. The figure window contains a single plot.

Data Types: double
Complex Number Support: Yes

n — Number of samples per trace
integer greater than 1

Number of samples per trace, specified as a integer greater than 1.

Data Types: double

period — Trace period
1 (default) | positive scalar

Trace period, specified as a positive scalar. The labels on the horizontal axis of the eye diagram range between -period/2 to period/2.

Data Types: double

offset — Offset value
0 (default) | integer in the range from 0 to (n–1)

Offset value, specified as an integer in the range 0 to (n–1). The function assumes that the (offset + 1)th value of the signal and every nth value thereafter, occur at times that are integer multiples of the input period.

Data Types: double

plotstring — Plot attributes
'b-' (default) | character vector | string scalar

Plot attributes, specified as a character vector or string scalar containing symbols.

This argument sets the plotting symbol, line type, and color for the eye diagram. The format and meaning of the symbols are the same as in the plot function. For example, the default value 'b-' produces a solid blue line.

Data Types: char | string

h — Figure handle
Figure object

Figure handle to an existing figure that contains an eye diagram, specified as a Figure object. h must be a handle to a figure that the eyediagram function previously generated.

Output Arguments

h — Figure handle
`Figure` object

Figure handle, returned as a `Figure` object. To modify properties of this object, see `Figure Properties`.

See Also

Functions
`plot` | `scatterplot`

Objects
`comm.ConstellationDiagram`

Topics
"Eye Diagram Analysis"

Introduced before R2006a

EyeScope

(Removed) Launch eye diagram scope for eye diagram object H

Note `eyescope` has been removed. Use `eyediagram` instead.

Syntax

```
eyescope
eyescope(h)
```

Description

Eye Diagram Scope is a graphical user interface (GUI) that enables you to visualize and measure the effects that various impairments, such as noise, jitter, and filtering, have on a modulated signal. The scope performs a probability density function (pdf) analysis on the signal to illustrate its trajectory in time, and to calculate such quantities as eye SNR, RMS jitter, rise time, and fall time. The scope also enables you to import and compare measurement results for eye diagrams of multiple signals.

There are two ways to call EyeScope:

- `eyescope` calls an empty scope
- `eyescope(h)` calls the scope and displays object `h`

Note You can call EyeScope with an eye diagram object as the input argument. EyeScope uses the `inputname` function to resolve the caller’s work space name for the argument. If the `inputname` function cannot resolve the caller’s work space name, then EyeScope uses a default name. To learn about the cases when EyeScope cannot determine the work space name, type `help inputname` at the MATLAB command line.

Compatibility Considerations

EyeScope has been removed
Errors starting in R2020a

`EyeScope` has been removed. Use `eyediagram` instead.

See Also

Functions
`eyediagram`

Introduced in R2008b

fft

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), the length of `x` must be 2^m-1.

Examples

Discrete Fourier Transform of Galois Vector

Set the order of the Galois field. Because `x` is in the Galois field (2^4), the length of `x` must be $2^m - 1$.

```
m = 4;
n = 2^m-1;
```

Generate a random GF vector.

```
x = gf(randi([0 2^m-1],n,1),m);
```

Perform the Fourier transform.

```
y = fft(x);
```

Invert the transform.

```
z = ifft(y);
```

Confirm that the inverse transform `z` = `x`.

```
isequal(z,x)
```

```
ans = logical
   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.

Algorithms

If x is a column vector, fft applies dftmtx to the primitive element of the Galois field and multiplies the resulting matrix by x.

See Also

dftmtx | gf | ifft

Topics

“Signal Processing Operations in Galois Fields”

Introduced before R2006a

filter (channel)

(To be removed) Filter signal with channel object

Note `filter` has been removed. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

Syntax

```
y = filter(chan,x)
```

Description

`y = filter(chan,x)` processes the baseband signal vector `x` with the channel object `chan`. The result is the signal vector `y`. The final state of the channel is stored in `chan`. You can construct `chan` using either `rayleighchan` or `ricianchan`. The `filter` function assumes `x` is sampled at frequency `1/ts`, where `ts` equals the `InputSamplePeriod` property of `chan`.

If `chan.ResetBeforeFiltering` is `0`, `filter` uses the existing state information in `chan` when starting the filtering operation. As a result, `filter(chan,[x1 x2])` is equivalent to `[filter(chan,x1) filter(chan,x2)]`. To reset `chan` manually, apply the `reset` function to `chan`.

If `chan.ResetBeforeFiltering` is `1`, `filter` resets `chan` before starting the filtering operation, overwriting any previous state information in `chan`.

Compatibility Considerations

`filter` has been removed

Errors starting in R2020b

filter (channel) has been removed. Use `comm.RicianChannel` instead.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`comm.RayleighChannel` | `comm.RicianChannel`

Topics

"Fading Channels"

Introduced in R2007a

filter (Galois field)

1-D digital filter over 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 shown here:

```
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

Filter a Galois Field

When using the Galois 1-D digital filter function, the data is normalized by the first element of the denominator coefficient vector.

```
a = gf([2 3 5 7],3);
b = gf([1 3],3);
x = gf(randi([0,7],10,1),3);
filt_x = filter(b,a,x)
```

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

Array elements =

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

The first coefficient of the denominator coefficient vector, a(1) = 2. To confirm the function normalizes the data, manually normalize the filtered data. Use `isequal` to compare the outputs, we see they are equal.

```
filt_x2 = a(1) * filter(b/a(1),a,x)
```

```
filt_x2 = GF(2^3) array. Primitive polynomial = D^3+D+1 (11 decimal)

Array elements =

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

```
isequal(filt_x,filt_x2)
```

```
ans = logical
   1
```

See Also

`gf`

Introduced before R2006a

fmdemod

Frequency demodulation

Syntax

```
z = fmdemod(y,Fc,Fs,freqdev)
z = fmdemod(y,Fc,Fs,freqdev,ini_phase)
```

Description

`z = fmdemod(y,Fc,Fs,freqdev)` returns a demodulated signal `z`, given the input frequency modulated (FM) signal `y`, where the carrier signal has frequency `Fc` and sampling rate `Fs`. `freqdev` is the frequency deviation of the modulated signal.

Note

- The value of `Fs` must satisfy `Fs` ≥ 2`Fc`.
- The value of `freqdev` must satisfy `freqdev` < `Fc`.

`z = fmdemod(y,Fc,Fs,freqdev,ini_phase)` specifies the initial phase of the modulated signal in radians.

Examples

FM Modulate and Demodulate Sinusoidal Signal

Set the sampling frequency to 1kHz and carrier frequency to 200 Hz. Generate a time vector having a duration of 0.2 s.

```
fs = 1000;
fc = 200;
t = (0:1/fs:0.2)';
```

Create two-tone sinusoidal signal with frequencies 30 and 60 Hz.

```
x = sin(2*pi*30*t)+2*sin(2*pi*60*t);
```

Set the frequency deviation to 50 Hz.

```
fDev = 50;
```

Frequency modulate `x`.

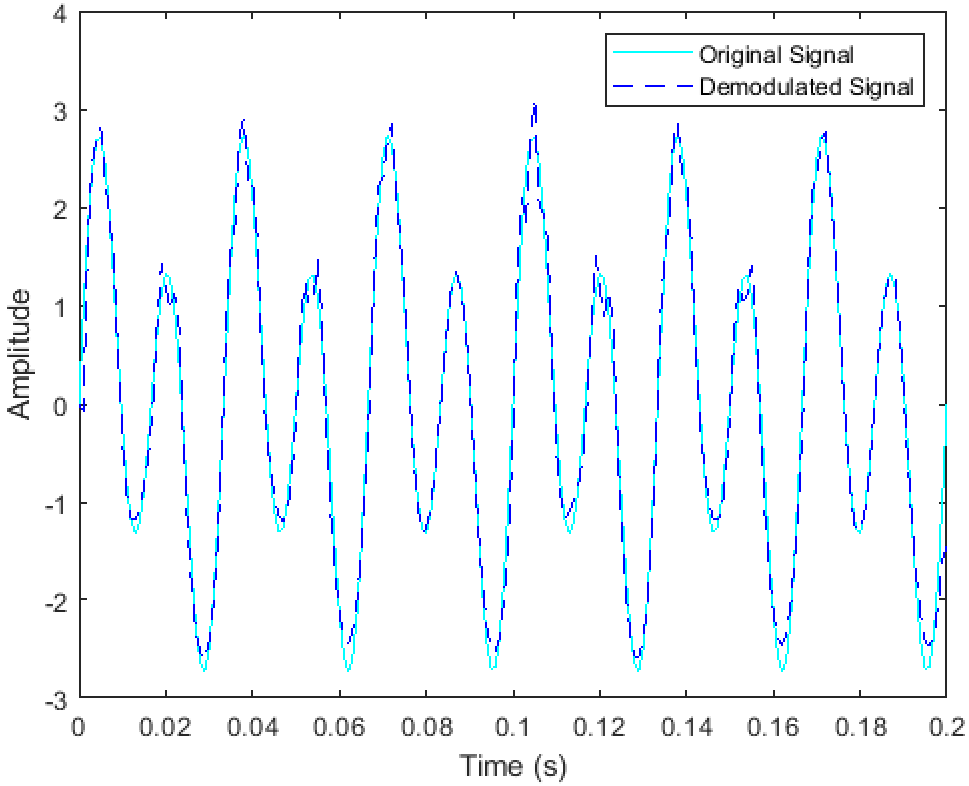
```
y = fmmod(x,fc,fs,fDev);
```

Demodulate z.

```
z = fmdemod(y,fc,fs,fDev);
```

Plot the original and demodulated signals.

```
plot(t,x,'c',t,z,'b--');
xlabel('Time (s)')
ylabel('Amplitude')
legend('Original Signal','Demodulated Signal')
```



The demodulated signal closely approximates the original.

Input Arguments

y — Frequency modulated input signal

scalar | vector | matrix | 3-D array

Frequency modulated input signal, specified as a scalar, vector, matrix, or 3-D array. Each element of y must be real.

Data Types: double | single

Fc — Carrier frequency

positive real scalar

Carrier frequency in hertz (Hz), specified as a positive real scalar.

Data Types: double

`Fs` — Sampling rate
positive scalar

Sampling rate in hertz (Hz), specified as a positive scalar.

Data Types: `double`

`freqdev` — Frequency deviation
positive scalar

Frequency deviation of the modulated signal in hertz (Hz), specified as a positive scalar.

Data Types: `double`

`ini_phase` — Initial phase
scalar

Initial phase of the modulated signal in radians, specified as a scalar.

Data Types: `double`

Output Arguments

`z` — Frequency demodulated output signal
scalar | vector | matrix | 3-D array

Frequency demodulated signal, returned as a scalar, vector, matrix, or 3-D array.

See Also

Functions
`amdemod` | `fmmod` | `pmdemod`

Objects
`comm.FMBroadcastDemodulator` | `comm.FMDemodulator`

Topics
"Analog Passband Modulation"

Introduced before R2006a

fmmod

Frequency modulation

Syntax

```
y = fmmod(x,Fc,Fs,freqdev)
y = fmmod(x,Fc,Fs,freqdev,ini_phase)
```

Description

`y = fmmod(x,Fc,Fs,freqdev)` returns a frequency modulated (FM) signal `y`, given the input message signal `x`, where the carrier signal has frequency `Fc` and sampling rate `Fs`. `freqdev` is the frequency deviation of the modulated signal.

Note

- The value of `Fs` must satisfy `Fs` ≥ 2`Fc`.
- The value of `freqdev` must satisfy `freqdev` < `Fc`.

`y = fmmod(x,Fc,Fs,freqdev,ini_phase)` specifies the initial phase of the modulated signal.

Examples

FM Modulate a Sinusoidal Signal

Set the sampling frequency to 1kHz and carrier frequency to 200 Hz. Generate a time vector having a duration of 0.2 s.

```
fs = 1000;
fc = 200;
t = (0:1/fs:0.2)';
```

Create two tone sinusoidal signal with frequencies 30 and 60 Hz.

```
x = sin(2*pi*30*t)+2*sin(2*pi*60*t);
```

Set the frequency deviation to 50 Hz.

```
fDev = 50;
```

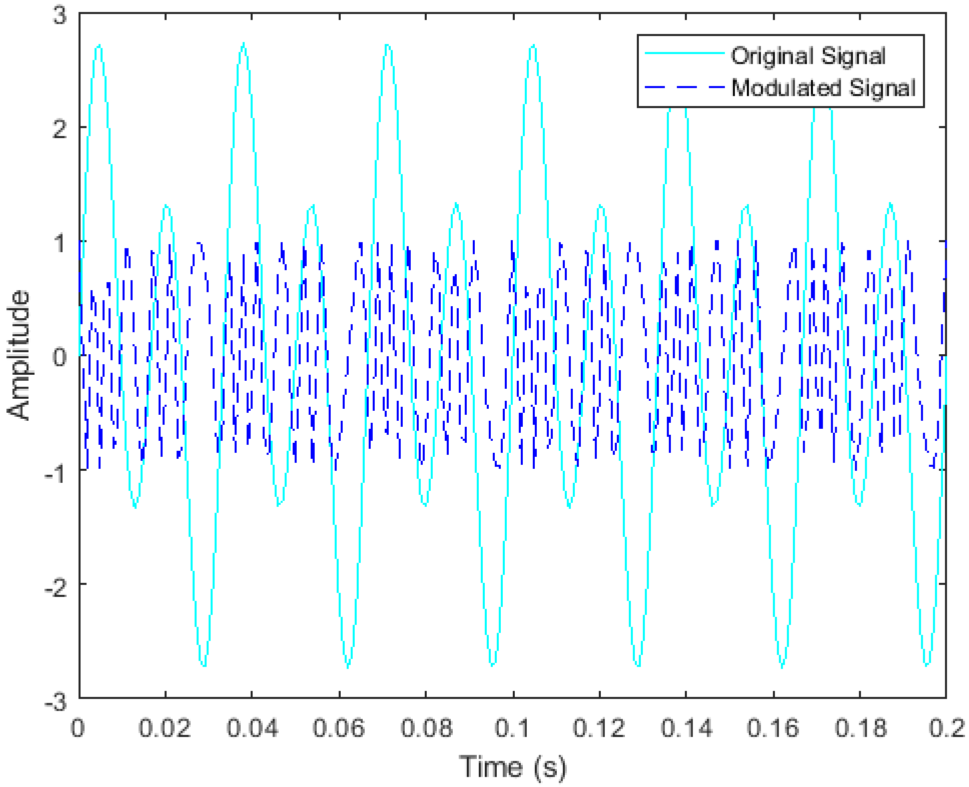
Frequency modulate `x`.

```
y = fmmod(x,fc,fs,fDev);
```

Plot the original and modulated signals.

```
plot(t,x,'c',t,y,'b--')
xlabel('Time (s)')
```

```
ylabel('Amplitude')
legend('Original Signal','Modulated Signal')
```



Input Arguments

x — Input message signal

scalar | vector | matrix | 3-D array

Input message signal, specified as a scalar, vector, matrix, or a 3-D array. Each element of x must be real.

Data Types: single | double

Fc — Carrier frequency

positive real scalar

Carrier frequency in hertz (Hz), specified as a positive real scalar.

Data Types: single | double

Fs — Sampling rate

positive real scalar

Sampling rate in hertz (Hz), specified as a positive real scalar.

Data Types: single | double

`freqdev` — Frequency deviation

positive real scalar

Frequency deviation of the modulated signal in hertz (Hz), specified as a positive real scalar.

Data Types: `single` | `double`

`ini_phase` — Initial phase

real scalar

Initial phase of the modulated signal in radians, specified as a real scalar.

Data Types: `single` | `double`

Output Arguments

y — Frequency modulated output signal

scalar | vector | matrix | 3-D array

Frequency modulated signal, returned as a scalar, vector, matrix, or 3-D array.

See Also

Functions
`ammod` | `fmdemod` | `pmmod`

Objects
`comm.FMModulator` | `comm.FMBroadcastModulator`

Topics
“Analog Passband Modulation”

Introduced before R2006a

fskdemod

Frequency shift keying demodulation

Syntax

```
z = fskdemod(y,M,freq_sep,nsamp)
z = fskdemod(y,M,freq_sep,nsamp,Fs)
z = fskdemod(y,M,freq_sep,nsamp,Fs,symorder)
```

Description

`z = fskdemod(y,M,freq_sep,nsamp)` noncoherently demodulates the complex envelope `y` of a signal using the frequency shift key method.

`z = fskdemod(y,M,freq_sep,nsamp,Fs)` specifies the sampling frequency in Hz.

`z = fskdemod(y,M,freq_sep,nsamp,Fs,symorder)` specifies how the function assigns binary words to corresponding integers.

Examples

Modulation and Demodulation of an FSK Signal in AWGN

Pass an FSK signal through an AWGN channel and estimate the resulting bit error rate (BER). Compare the estimated BER to the theoretical value.

Set the simulation parameters.

```
M = 2;          % Modulation order
k = log2(M);    % Bits per symbol
EbNo = 5;       % Eb/No (dB)
Fs = 16;        % Sample rate (Hz)
nsamp = 8;      % Number of samples per symbol
freqsep = 10;   % Frequency separation (Hz)
```

Generate random data symbols.

```
data = randi([0 M-1],5000,1);
```

Apply FSK modulation.

```
txsig = fskmod(data,M,freqsep,nsamp,Fs);
```

Pass the signal through an AWGN channel

```
rxSig  = awgn(txsig,EbNo+10*log10(k)-10*log10(nsamp),...
    'measured',[],'dB');
```

Demodulate the received signal.

```
dataOut = fskdemod(rxSig,M,freqsep,nsamp,Fs);
```

Calculate the bit error rate.

```
[num,BER] = biterr(data,dataOut);
```

Determine the theoretical BER and compare it to the estimated BER. Your BER value might vary because the example uses random numbers.

```
BER_theory = berawgn(EbNo,'fsk',M,'noncoherent');
[BER BER_theory]
```

```
ans = 1×2

    0.0958    0.1029
```

Input Arguments

y — FSK-modulated output signal

vector | matrix

Complex baseband representation of a FSK-modulated signal, specfied as vector or matrix of complex values. If `y` is a matrix with multiple rows and columns, fskdemod processes the columns independently.

Data Types: `double` | `single`

`M` — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: `double`

`symorder` — Symbol order

`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

`freq_sep` — Desired separation between frequencies

positive scalar

Desired separation between frequencies, specified in Hz. By the Nyquist sampling theorem, `freq_sep` and `M` must satisfy (`M`-1)*`freq_sep` <= 1.

Data Types: `double`

`nsamp` — Number of samples per output symbol

positive scalar greater than 1

Number of samples per output symbol, specified as a positive scalar greater than 1.

Data Types: `double`

`Fs` — Sample rate
1 Hz (default) | positive scalar

Sample rate, specified as a positive scalar.

Data Types: `double`

Output Arguments

`z` — Output signal
vector | matrix

Output signal, returned as a vector or matrix of positive integers. The elements of `z` have values in the range of [0, `M` – 1].

Example: `randi([0 3],100,1)`

Data Types: `double`

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications.* Upper Saddle River, NJ: Prentice-Hall, 2001.

See Also

`fskmod` | `pskdemod` | `pskmod`

Topics
"Digital Modulation"

Introduced before R2006a

fskmod

Frequency shift keying modulation

Syntax

```
y = fskmod(x,M,freq_sep,nsamp)
y = fskmod(x,M,freq_sep,nsamp,Fs)
y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)
y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont,symorder)
```

Description

`y = fskmod(x,M,freq_sep,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using frequency shift keying modulation.

`y = fskmod(x,M,freq_sep,nsamp,Fs)` specifies the sampling rate of `y`.

`y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont)` specifies the phase continuity.

`y = fskmod(x,M,freq_sep,nsamp,Fs,phase_cont,symorder)` specifies how the function assigns binary words to corresponding integers.

Examples

FSK Signal Spectrum Plot

Generate an FSK modulated signal and display its spectral characteristics.

Set the function parameters.

```
M = 4;          % Modulation order
freqsep = 8;    % Frequency separation (Hz)
nsamp = 8;      % Number of samples per symbol
Fs = 32;        % Sample rate (Hz)
```

Generate random M-ary symbols.

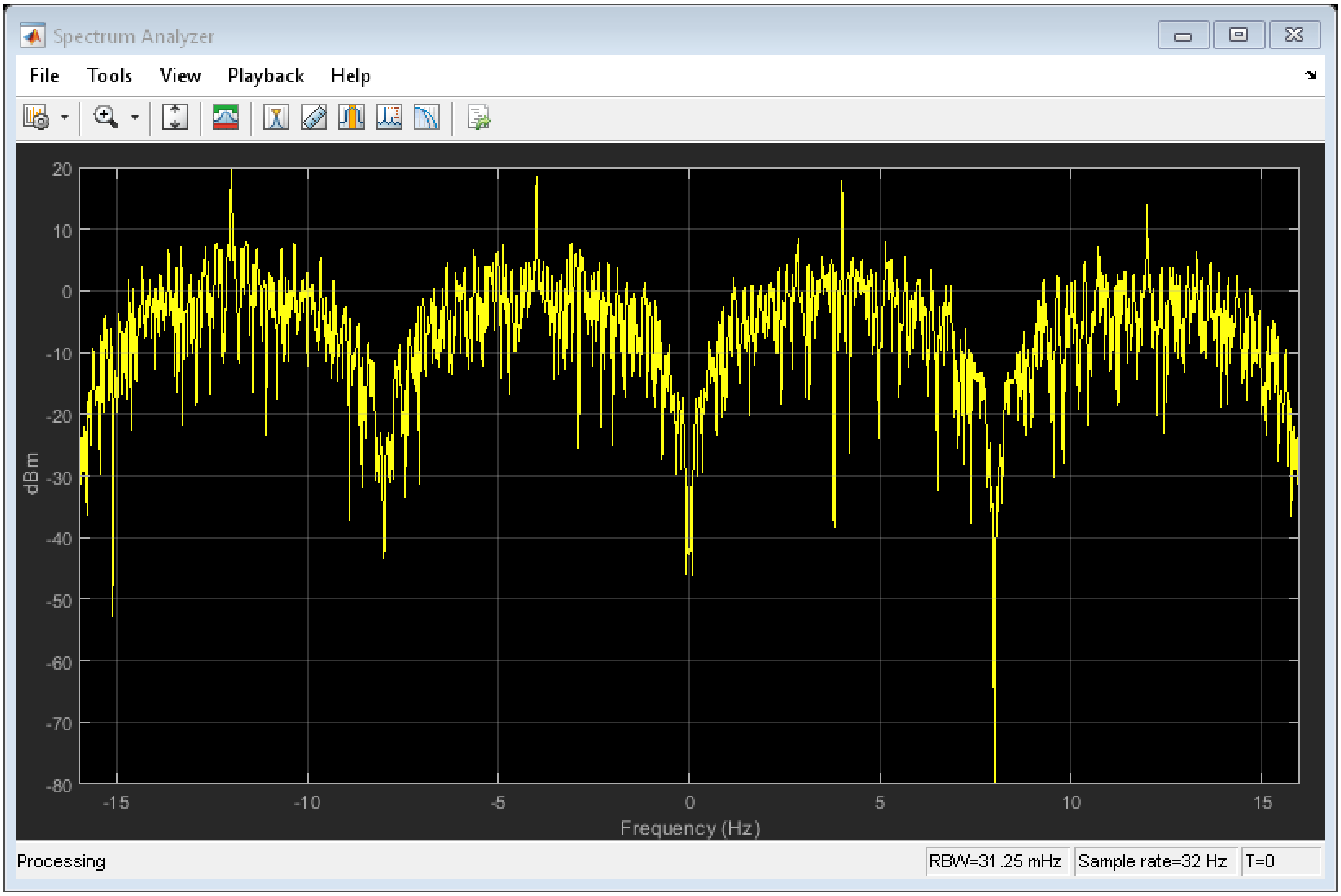
```
x = randi([0 M-1],1000,1);
```

Apply FSK modulation.

```
y = fskmod(x,M,freqsep,nsamp,Fs);
```

Create a spectrum analyzer System object? and use its `step` method to display a plot of the signal spectrum.

```
h = dsp.SpectrumAnalyzer('SampleRate',Fs);
step(h,y)
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of x must have values in the range of [0, M – 1]. If x is a matrix, fskmod processes the columns independently.

Example: randi([0 3],100,1)

Data Types: double

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double

symorder — Symbol order

'bin' (default) | 'gray'

Symbol order, specified as 'bin' or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

- If symorder is 'bin', the function uses a natural binary-coded ordering.
- If symorder is 'gray', the function uses a Gray-coded ordering.

Data Types: char

freq_sep — Desired separation between frequencies

positive scalar

Desired separation between frequencies, specified in Hz. By the Nyquist sampling theorem, freq_sep and M must satisfy (M-1)*freq_sep <= 1.

Data Types: double

nsamp — Number of samples per output symbol

positive scalar greater than 1

Number of samples per output symbol, specified as a positive scalar greater than 1.

Data Types: double

Fs — Sample rate

1 Hz (default) | positive scalar

Sample rate, specified as a positive scalar.

Data Types: double

phase_cont — Phase continuity

'cont' (default) | 'discont'

Phase continuity, specified as either 'cont' or 'discont'. Set phase_cont to 'cont' to force phase continuity across symbol boundaries in y, or 'discont' to avoid forcing phase continuity.

Data Types: char

Output Arguments

y — FSK-modulated output signal

vector | matrix

Complex baseband representation of a FSK-modulated signal, returned as vector or matrix of complex values. The columns of y represent independent channels.

Data Types: double | single

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications.* Upper Saddle River, NJ: Prentice-Hall, 2001.

See Also

fskdemod | pskdemod | pskmod

Topics

“Digital Modulation”

Introduced before R2006a

gen2par

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

Convert Parity-Check Matrix for a Hamming Code to Generator Matrix

Convert the parity-check matrix for a Hamming code into the corresponding generator matrix and back again.

Create the parity-check matrix.

```
parmat = hammgen(3)
```

```
parmat = 3×7

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

Convert the parity-check matrix into the corresponding generator matrix.

```
genmat = gen2par(parmat)
genmat = 4×7

     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
```

Convert the generator matrix back again. The output, `parmat2`, should be the same as the original matrix, `parmat`.

```
parmat2 = gen2par(genmat)
```

```
parmat2 = 3×7

     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`

Topics

"Block Codes"

Introduced before R2006a

genqamdemod

General quadrature amplitude demodulation

Syntax

```
z = genqamdemod(y,const)
```

Description

`z = genqamdemod(y,const)` demodulates the complex envelope, `y`, of a quadrature amplitude modulated signal using the signal mapping specified in `const`.

Examples

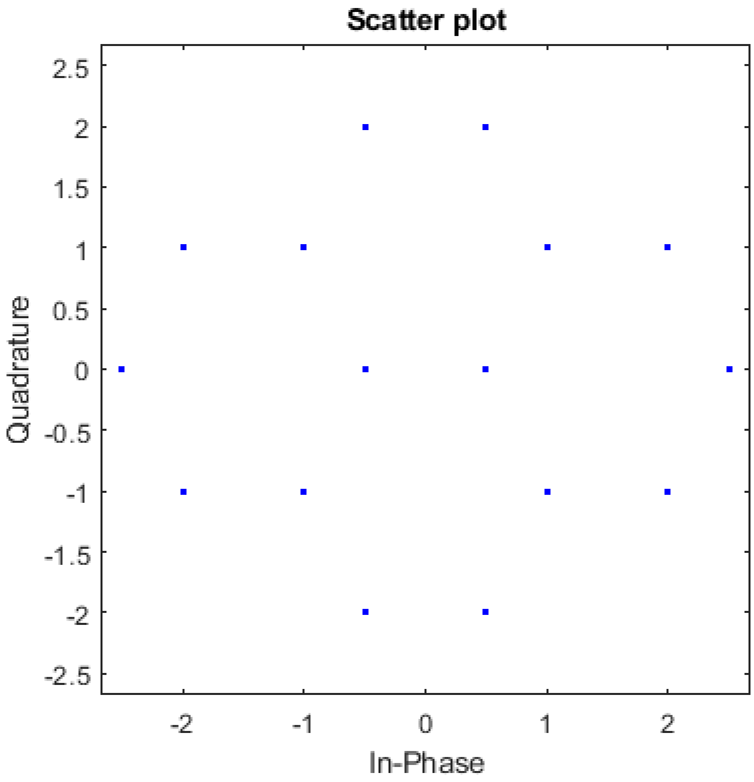
General QAM Modulation and Demodulation

Create the points that describe 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(:);
const = inphase + 1i*quadr;
```

Plot the constellation.

```
h = scatterplot(const);
```



Generate input data symbols. Modulate the symbols using this constellation.

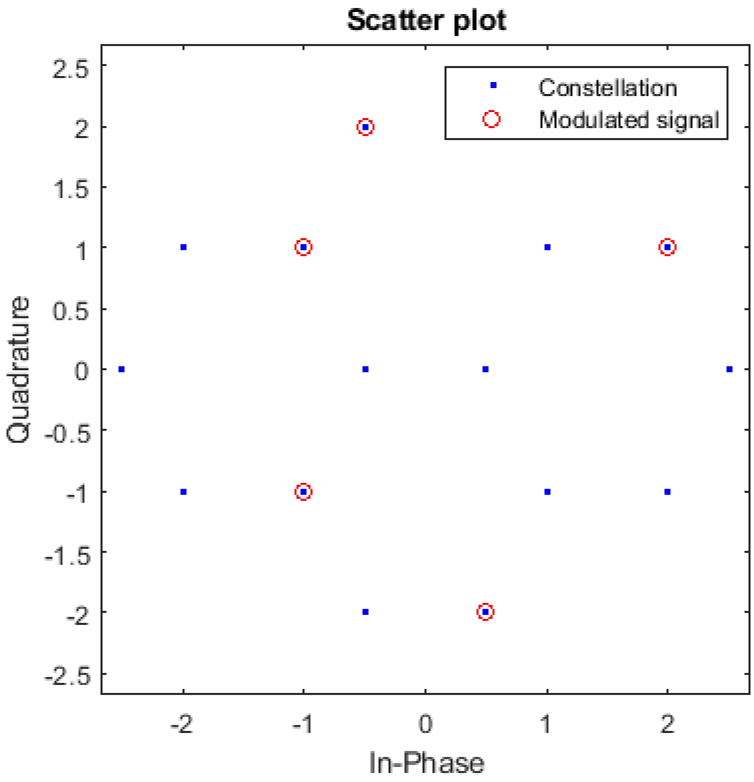
```
x = [3 8 5 10 7];
y = genqammod(x,const);
```

Demodulate the modulated signal, y.

```
z = genqamdemod(y,const);
```

Plot the modulated signal in same figure.

```
hold on;
scatterplot(y,1,0,'ro',h);
legend('Constellation','Modulated signal');
```



Determine the number of symbol errors between the demodulated data to the original sequence.

```
numErrs = symerr(x,z)
```

```
numErrs = 0
```

Input Arguments

y — Complex envelope

scalar | vector | matrix | 3-D array

Complex envelope, specified as a scalar, vector, matrix, or 3-D array of numeric values. If `y` is a matrix with multiple rows, the function processes the rows independently.

`const` — Signal mapping

complex vector

Signal mapping, specified as a complex vector.

Data Types: `double` | `single`

Output Arguments

z — Message signal

scalar | vector | matrix | 3-D array

Message signal, returned as a scalar, vector, matrix, or 3-D array of numeric values. The message signal consists of integers between 0 and `length(const)`–1. The datatype of `z` is the same as the data type of input `x`.

Data Types: `double` | `single`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`genqammod` | `pamdemod` | `pammod` | `qamdemod` | `qammod`

Topics
"Digital Modulation"

Introduced before R2006a

genqammod

General quadrature amplitude modulation (QAM)

Syntax

```
y = genqammod(x,const)
```

Description

`y = genqammod(x,const)` returns the complex envelop of the QAM for message signal `x`. Input `const` specifies the signal mapping for the modulation.

Examples

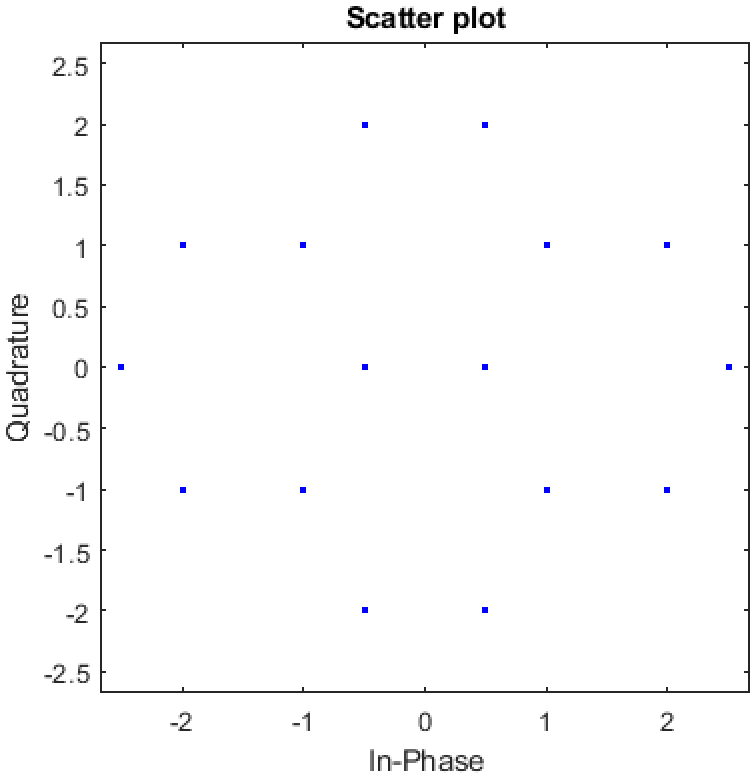
General QAM Modulation and Demodulation

Create the points that describe 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(:);
const = inphase + 1i*quadr;
```

Plot the constellation.

```
h = scatterplot(const);
```



Generate input data symbols. Modulate the symbols using this constellation.

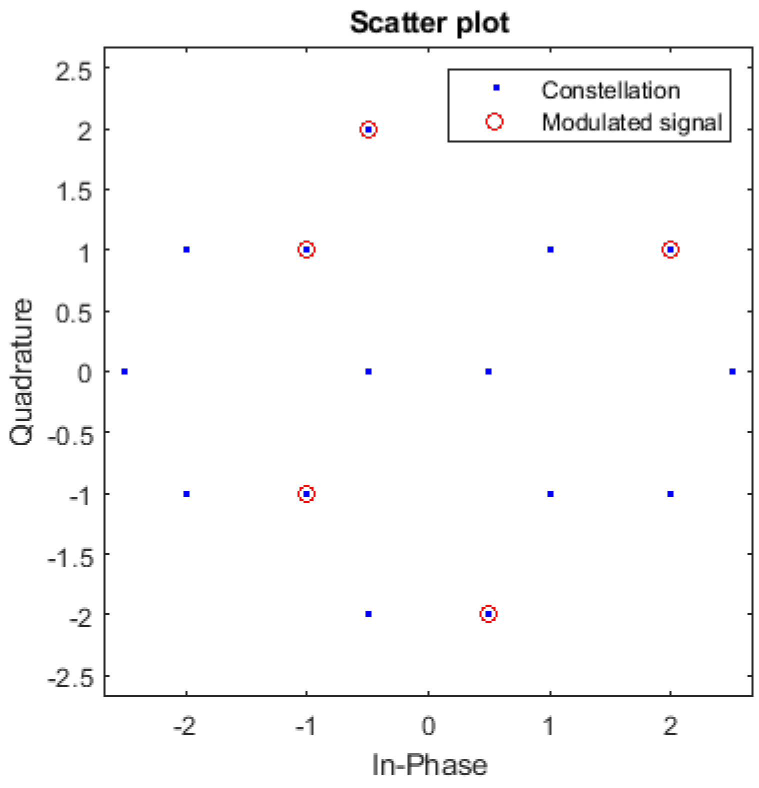
```
x = [3 8 5 10 7];
y = genqammod(x,const);
```

Demodulate the modulated signal, y.

```
z = genqamdemod(y,const);
```

Plot the modulated signal in same figure.

```
hold on;
scatterplot(y,1,0,'ro',h);
legend('Constellation','Modulated signal');
```



Determine the number of symbol errors between the demodulated data to the original sequence.

```
numErrs = symerr(x,z)
```

```
numErrs = 0
```

Input Arguments

x — Message signal

scalar | vector | matrix | 3-D array

Message signal, specified as a scalar, vector, matrix, or 3-D array of numeric values. The message signal must consist of integers from 0 and `length(const)`–1. If `x` is a matrix with multiple rows, the function processes the columns independently.

Data Types: `double` | `single` | `fi` | `int8` | `int16` | `uint8` | `uint16`

`const` — Signal mapping

complex vector

Signal mapping, specified as a complex vector.

Data Types: `double` | `single` | `fi` | `int8` | `int16` | `uint8` | `uint16`

Output Arguments

y — Complex envelope
scalar | vector | matrix | 3-D array

Complex envelope, returned as a scalar, vector, matrix, or 3-D array of numeric values. The length of `y` is the same as the length of input `x`.

Data Types: `double` | `single` | `fi` | `int8` | `int16` | `uint8` | `uint16`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`genqamdemod` | `pamdemod` | `pammod` | `qamdemod` | `qammod`

Topics
"Digital Modulation"

Introduced before R2006a

gf

Galois field array

Syntax

```
x_gf = gf(x)
x_gf = gf(x,m)
x_gf = gf(x,m,prim_poly)
```

Description

`x_gf = gf(x)` creates a Galois field (GF) array, GF(2), from matrix `x`.

`x_gf = gf(x,m)` creates a Galois field array from matrix `x`. The Galois field has 2^m elements, where `m` is an integer from 1 through 16.

`x_gf = gf(x,m,prim_poly)` creates a Galois field array from matrix `x` by using the primitive polynomial `prim_poly`.

Examples

Create GF(2) Array from Specified Matrix

Specify a matrix of `0`s and `1`s.

```
x = [0 1 1; 0 1 0; 1 1 1];
```

Create a GF(2) array from `x`.

```
x_gf = gf(x)
```

```
x_gf = GF(2) array.

Array elements =

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

Create Sequence of GF(16) Elements

Set the order of the Galois field to 16, where the order equals 2^m. Specify a matrix of elements that range from 0 to $2^m - 1$. Create the Galois field array.

```
m = 4;
x = [3 2 9; 1 2 1];
y = gf(x,m)
```

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

Array elements = 

   3   2   9
   1   2   1
```

Create GF Sequence with Specified Primitive Polynomial

Create a sequence of integers. Create a Galois field array in GF(2^5).

```
x = [17 8 11 27];
y = gf(x,5)
```

```
y = GF(2^5) array. Primitive polynomial = D^5+D^2+1 (37 decimal)

Array elements = 

   17    8   11   27
```

Determine all possible primitive polynomials for GF(2^5).

```
pp = primpoly(5,'all')
```

```
Primitive polynomial(s) = 

D^5+D^2+1
D^5+D^3+1
D^5+D^3+D^2+D^1+1
D^5+D^4+D^2+D^1+1
D^5+D^4+D^3+D^1+1
D^5+D^4+D^3+D^2+1

pp = 6×1

    37
    41
    47
    55
    59
    61
```

Create a Galois field array using the primitive polynomial that has a decimal equivalent of 59.

```
z = gf(x,5,'D5+D4+D3+D+1')
```

```
z = GF(2^5) array. Primitive polynomial = D^5+D^4+D^3+D+1 (59 decimal)

Array elements = 

   17    8   11   27
```

Input Arguments

x — Input matrix

matrix with all values greater than or equal to zero

Input matrix, specified as a matrix with values greater than or equal to zero. The function uses this value to create a GF array.

- If you do not specify the `prim_poly` input argument, each element of `x` must be an integer in the range [0, 2^m–1].
- If you specify `prim_poly` input argument, each element of `x` must be `0` or `1`.

Data Types: `double`

m — Order of primitive polynomial

positive integer

Order of primitive polynomial, specified as a positive integer from 1 through 16. The function uses this value to calculate the distinct number of elements in the GF.

Data Types: `double`

prim_poly — Primitive polynomial

primitive polynomial in GF(2^m) (default) | binary row vector | character vector | string scalar | positive integer

Primitive polynomial, specified as one of these options:

- Binary row vector — This vector specifies coefficients of `prim_poly` in the order of ascending powers.
- Character vector or a string scalar — This value defines `prim_poly` in a textual representation. For more details, refer to polynomial character vector.
- Positive integer — This value defines `prim_poly` in the range [(2^m + 1), (2^{m+1} – 1)].

If `prim_poly` is not specified, see “Default Primitive Polynomials” on page 2-400 for the list of default primitive polynomial used for each Galois field array GF(2^m).

Data Types: `double` | `char` | `string`

Output Arguments

x_gf — Galois field array

variable that MATLAB recognizes as a Galois field array

Galois field array, returned as 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 the variable specifies. For example, if you apply the `log` function to a Galois array, MATLAB computes the logarithm in the Galois field for that Galois array and not in the field of real or complex numbers.

More About

Default Primitive Polynomials

This table lists the default primitive polynomial used for each Galois field array GF(2^m). To use a different primitive polynomial, specify `prim_poly` as an input argument. `prim_poly` must be in the range [(2^m + 1), (2^{m+1} – 1)] and must indicate an irreducible polynomial. For more information, see “Primitive Polynomials and Element Representations”.

Value of 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

Galois Computations

This table lists the operations supported for Galois field arrays.

Operation	Description
+ -	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

Operation	Description
conv	Convolution of Galois vectors
convmtx	Convolution matrix of Galois field vector
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 (gf)	One-dimensional digital filter over a Galois field
ifft	Inverse discrete Fourier transform
inv	Inverse of Galois matrix
length	Length of Galois vector
log	Logarithm in a Galois field
lu	Lower-Upper triangular factorization of Galois array
minpol	Find the minimal polynomial for a Galois element
mldivide	Matrix left division \ of Galois arrays
polyval	Evaluate polynomial in 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

See Also

Functions

cosets | gftable | isprimitive | primpoly

Topics

"Galois Field Computations"
"Error Detection and Correction"
"ElGamal Public Key Cryptosystem"

Introduced before R2006a

gfadd

Add polynomials over Galois field

Syntax

```
c = gfadd(a,b)
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 prime. To work in GF(2^m), apply the + operator to Galois arrays of equal size. For details, see “Example: Addition and Subtraction”.

`c = gfadd(a,b)` adds two GF(2) polynomials, `a` and `b`, which can be either polynomial character vectors or numeric vectors. If `a` and `b` are vectors of the same orientation but different lengths, then the shorter vector is zero-padded. If `a` and `b` are matrices they must be of the same size.

`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, 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), extra zeros are added at the end; if it has more than `len` entries, 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” for an explanation of these formats. If `a` and `b` are matrices of the same size, the function treats each element independently.

Examples

Add Two GF Arrays

Sum $2 + 3x + x^2$ and $4 + 2x + 3x^2$ over GF(5).

```
x = gfadd([2 3 1],[4 2 3],5)
```

```
x = 1×3

     1     0     4
```

Add the two polynomials and display the first two elements.

```
y = gfadd([2 3 1],[4 2 3],5,2)
```

```
y = 1×2

     1     0
```

For prime number p and exponent m, create a matrix listing all elements of GF(p^m) given primitive polynomial $2 + 2x + x^2$.

```
p = 3;
m = 2;
primpoly = [2 2 1];
field = gftuple((-1:p^m-2)',primpoly,p);
```

Sum A^2 and A^4. The result is A.

```
g = gfadd(2,4,field)
```

```
g = 1
```

See Also

gfconv | gfdeconv | gfdiv | gfmul | gfsub | gftuple

Topics

"Arithmetic in Galois Fields"

Introduced before R2006a

gfconv

Multiply polynomials over Galois field

Syntax

```
c = gfconv(a,b)
c = gfconv(a,b,p)
c = gfconv(a,b,field)
```

```
c = gfconv(polys)
c = gfconv(polys,p)
c = gfconv(polys,field)
```

Description

`c = gfconv(a,b)` returns a row vector that specifies the GF(2) polynomial coefficients in order of ascending powers. The returned vector results from the multiplication of GF(2) polynomials `a` and `b`. The polynomial degree of the resulting GF(2) polynomial `c` equals the degree of `a` plus the degree of `b`.

For additional information, see “Tips” on page 2-408.

`c = gfconv(a,b,p)` multiplies two GF(`p`) polynomials, where `p` is a prime number. `a`, `b`, and `c` are in the same Galois field. `a`, `b`, and `c` are polynomials with coefficients in order of ascending powers. Each coefficient is in the range [0, `p`–1].

`c = gfconv(a,b,field)` multiplies two GF(p^m) polynomials, where `field` is a matrix containing the *m*-tuple of all elements in GF(p^m). *p* is a prime number, and m is a positive integer. `a`, `b`, and `c` are in the same Galois field.

In this syntax, each coefficient is specified in exponential format, specifically [-Inf, 0, 1, 2, ...]. The elements in exponential format represent the `field` elements [0, 1, α, α^2, ...] relative to some primitive element α of GF(p^m).

`c = gfconv(polys)` returns a row vector that specifies the GF(2) polynomial coefficients in order of ascending powers. The returned vector results from the multiplication of the GF(2) polynomials specified in `polys`. The polynomial degree of the resulting GF(2) polynomial `c` equals the sum of the degrees of the polynomials contained in `polys`. Use this syntax when `polys` specifies polynomials as a cell array of character vectors or as a string array.

`c = gfconv(polys,p)` multiplies the GF(`p`) polynomials specified in `polys`, where `p` is a prime number. `polys` and `c` are polynomials with coefficients in order of ascending powers. Each coefficient is in the range [0, `p`–1]. `a`, `b`, and `c` are in the same Galois field.

`c = gfconv(polys,field)` multiplies the GF(p^m) polynomials in `polys`, where `field` is a matrix containing the *m*-tuple of all elements in GF(p^m). *p* is a prime number, and *m* is a positive integer. `a`, `b`, and `c` are in the same Galois field.

In this syntax, each coefficient is specified in exponential format, specifically [-Inf, 0, 1, 2, ...]. The elements in exponential format represent the `field` elements [0, 1, α, α^2, ...] relative to some primitive element α of GF(p^m).

Examples

Multiply GF(2) Polynomials

Multiply $1 + 2x + 3x^2 + 4x^3$ and $1 + x$ three times. Represent the polynomials as row vectors, character vectors, and strings.

```
c_rv = gfconv([1 1 0 1],[1 1])
```

```
c_rv = 1×5

     1     0     1     1     1
```

```
c_cv = gfconv('1 + x + x^3','1 + x')
```

```
c_cv = 1×5

     1     0     1     1     1
```

```
c_s = gfconv("1 + x + x^3","1 + x")
```

```
c_s = 1×5

     1     0     1     1     1
```

The results corresponds to $1 + x^2 + x^3 + x^4$.

Multiply Polynomials Over GF(3)

Multiply $1 + x + x^4$ and $x + x^2$ over the Galois field GF(3).

```
gfc = gfconv([1 1 0 0 1],[0 1 1],3)
```

```
gfc = 1×7

     0     1     2     1     0     1     1
```

The result corresponds to $x + 2x^2 + x^3 + x^5 + x^6$.

Multiply Polynomials Over GF(2^4) Using Field Input

Multiply $1 + 2x + 3x^2 + 4x^3 + 5x^4$ and $1 + x$ in the Galois field GF(2^4).

```
field = gftuple([-1:2^4-2]',4,2);
c = gfconv('1 + 2x + 3x^2 + 4x^3 + 5x^4','1 + x',field)
```

```
c = 1×6
```

```
     2     6     7     8     9     6
```

Use the `gfpretty` function to display the result in polynomial form.

```
gfpretty(c)
```

```
                                     2       3       4       5
                      2 + 6 X + 7 X  + 8 X  + 9 X  + 6 X
```

Multiply GF(2) Polynomials Specified As Cell Array

Create a cell array containing three polynomials that result in the DVB-S2 generator polynomial for *t* = 3 when multiplied together.

```
polyCell = {'1 + x + x3 + x5 + x14', ...
    '1 + x6 + x8 + x11 + x14','1 + x + x2 + x6 + x9 + x10 + x14'};
gp = gfconv(polyCell); % DVB-S2 for t=3
```

Use the `gfpretty` function to display the result in polynomial form.

```
gfpretty(gp)
```

```
     4    6    8    10    11    13    16    17    20    24    25    26    27
1 + X  + X  + X  + X   + X   + X   + X   + X   + X   + X   + X   + X   + X

              30    31    32    33    34    35    36    37    38    39    42
           + X   + X   + X   + X   + X   + X   + X   + X   + X   + X   + X
```

Multiply Polynomials Expressed As Strings in GF(2^4) Using Field Input

Multiply $1 + 2x + 3x^2 + 4x^3 + 5x^4$, $1 + x$, and $1 + x^3$ in the Galois field GF(2^4).

```
field = gftuple((-1:2^4-2)', 4, 2);
c = gfconv(["1 + 2x + 3x^2 + 4x^3 + 5x^4","1 + x","1 + x3"],field)
```

```
c = 1×9

     4    13    14     9     2     1     7     8     8
```

Use the `gfpretty` function to display the result in polynomial form.

```
gfpretty(c)
```

```
                          2      3      4    5      6      7      8
           4 + 13 X + 14 X  + 9 X  + 2 X  + X  + 7 X  + 8 X  + 8 X
```

Input Arguments

a — Galois field polynomial

row vector | character vector | string

Galois field polynomial, specified as a row vector, character vector, or string. a can be either a "Character Representation of Polynomials" or numeric vector.

a and b must both be GF(*p*) polynomials or GF(p^m) polynomials, where *p* is prime. The value of *p* is as specified when included, 2 when omitted, or implied when `field` is specified.

Example: `[1 2 3 4]` is the polynomial $1+2x+3x^2+4x^3$ in GF(5) expressed as a row vector.

Data Types: `double` | `char` | `string`

b — Galois field polynomial

row vector | character vector | string

Galois field polynomial, specified as a row vector, character vector, or string. b can be either a "Character Representation of Polynomials" or numeric vector.

a and b must both be GF(*p*) polynomials or GF(p^m) polynomials, where *p* is prime. The value of *p* is as specified when included, 2 when omitted, or implied when `field` is specified.

Example: `'1 + x'` is a polynomial in GF(2^4) expressed as a character vector.

Data Types: `double` | `char` | `string`

p — Prime number

2 (default) | prime number

Prime number, specified as a prime number.

Data Types: `double`

`field` — *m*-tuple of all elements in GF(p^m)

matrix

m-tuple of all elements in GF(p^m), specified as a matrix. `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. To generate the *m*-tuple of all elements in GF(p^m), use

```
field =gftuple([-1:p^m-2]',m,p)
```

The coefficients, specified in exponential format, represent the field elements in GF(p^m). For an explanation of these formats, see "Representing Elements of Galois Fields".

Data Types: `double`

`polys` — Galois field polynomial list

cell array of character vectors | string array

Galois field polynomial list, specified as a cell array of character vectors or a string array.

Example: `["1+x+x3+x5+x14","1+x6+x8+x11+x14"]` is a string array of polynomials.

Data Types: `cell` | `string`

Output Arguments

c — Galois field polynomial

row vector

Galois field polynomial, returned as a row vector of the polynomial coefficients in order of ascending powers. The polynomial degree of the resulting GF(p^m) polynomial `c` equals the sum of the degrees of the input polynomials. `c` is in the same Galois field as the input polynomials.

Tips

- The gfconv function performs computations in GF(p^m), where p is prime, and m is a positive integer. It multiplies polynomials over a Galois field. To work in GF(2^m), you can also use the `conv` function of the `gf` object with Galois arrays. For details, see “Multiplication and Division of Polynomials”.
- To multiply elements of a Galois field, use `gfmul` instead of `gfconv`. Algebraically, multiplying polynomials over a Galois field is equivalent to convolving vectors containing the coefficients of the polynomials. This convolution operation uses arithmetic over the same Galois field.

See Also

Functions
`gfadd` | `gfdeconv` | `gfmul` | `gfpretty` | `gfsub` | `gftuple`

Topics
“Character Representation of Polynomials”
“Representing Elements of Galois Fields”
“Multiplication and Division of Polynomials”

Introduced before R2006a

gfcosets

Produce cyclotomic cosets for Galois field

Syntax

```
c = gfcosets(m)
c = gfcosets(m,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `cosets` function.

`c = gfcosets(m)` produces cyclotomic cosets mod(2^m - 1). Each row of the output GFCS contains one cyclotomic coset.

`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”.

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” on page 2-410.

Examples

The command below finds the cyclotomic cosets for GF(9).

```
c = gfcosets(2,3)
```

The output is

```
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.
```

The output is

```
m =

     1     0     1
     1     0     1
```

References

[1] Blahut, Richard E., *Theory and Practice of Error Control Codes*, Reading, MA, Addison-Wesley, 1983, p. 105.

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

See Also

`gfminpol` | `gfprimdf` | `gfroots`

Introduced before R2006a

gfdeconv

Divide polynomials over Galois field

Syntax

```
[q,r] = gfdeconv(b,a)
[q,r] = gfdeconv(b,a,p)
[q,r] = gfdeconv(b,a,field)
```

Description

`[q,r] = gfdeconv(b,a)` returns the quotient `q` and remainder `r` as row vectors that specify GF(2) polynomial coefficients in order of ascending powers. The returned vectors result from the division `b` by `a`. `a`, `b`, and `q` are in GF(2).

For additional information, see “Tips” on page 2-414.

`[q,r] = gfdeconv(b,a,p)` divides two GF(`p`) polynomials, where `p` is a prime number. `b`, `a`, and `q` are in the same Galois field. `b`, `a`, `q`, and `r` are polynomials with coefficients in order of ascending powers. Each coefficient is in the range [0, `p`–1].

`[q,r] = gfdeconv(b,a,field)` divides two GF(p^m) polynomials, where `field` is a matrix containing the *m*-tuple of all elements in GF(p^m). *p* is a prime number, and *m* is a positive integer. `b`, `a`, and `q` are in the same Galois field.

In this syntax, each coefficient is specified in exponential format, specifically [-Inf, 0, 1, 2, ...]. The elements in exponential format represent the `field` elements [0, 1, α, α^2, ...] relative to some primitive element α of GF(p^m).

Examples

Divide Polynomials in GF(3)

Divide $x + x^3 + x^4$ by $1 + x$ in the Galois field GF(3) three times. Represent the polynomials as row vectors, character vectors, and strings.

```
p = 3;
```

Represent the polynomials using row vectors and divide them in GF(3).

```
b = [0 1 0 1 1];
a = [1 1];
[q_rv,r_rv] = gfdeconv(b,a,p)
```

```
q_rv = 1×4

     1     0     0     1


r_rv = 2
```

To confirm the output, compare the original Galois field polynomials to the result of adding the remainder to the product of the quotient and the divisor.

```
bnew = gfadd(gfconv(q_rv,a,p),r_rv,p);
isequal(b,bnew)
```

```
ans = logical
   1
```

Represent the polynomials using character vectors and divide them in GF(3).

```
b = 'x + x^3 + x^4';
a = '1 + x';
[q_cv,r_cv] = gfdeconv(b,a,p)
```

```
q_cv = 1×4

     1     0     0     1


r_cv = 2
```

Represent the polynomials using strings and divide them in GF(3) .

```
b = "x + x^3 + x^4";
a = "1 + x";
[q_s,r_s] = gfdeconv(b,a,p)
```

```
q_s = 1×4

     1     0     0     1


r_s = 2
```

Use the `gfpretty` function to display the result without the remainder in polynomial form.

```
gfpretty(q_s)
```

```
                                     3
                                1 + X
```

Check for Irreducibility and Primitiveness Over GF(3^k)

In the Galois field GF(3), output polynomials of the form $x^k - 1$ for k in the range [2, 8] that are evenly divisible by $1 + x^2$. An irreducible polynomial over GF(p) of degree at least 2 is primitive if and only if it does not divide $-1 + x^k$ evenly for any positive integer k less than $p^m - 1$. For more information, see the `gfprimck` function.

The irreducibility of $1 + x^2$ over GF(3), along with the polynomials that are output, indicates that $1 + x^2$ is not primitive for GF(3^2).

```
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
      multiple{ii}=b;
      gfpretty(b)
   end
end
```

```
     4
2 + X

     8
2 + X
```

Input Arguments

b — Galois field polynomial

row vector | character vector | string

Galois field polynomial, specified as a row vector, character vector, or string. b can be either a “Character Representation of Polynomials” or numeric vector.

a and b must both be GF(p) polynomials or GF(p^m) polynomials, where p is prime. The value of p is as specified when included, 2 when omitted, or implied when field is specified.

Example: '1 + x' is a polynomial in GF(2^4) expressed as a character vector.

Data Types: double | char | string

a — Galois field polynomial

row vector | character vector | string

Galois field polynomial, specified as a row vector, character vector, or string. a can be either a “Character Representation of Polynomials” or numeric vector.

a and b must both be GF(p) polynomials or GF(p^m) polynomials, where p is prime. The value of p is as specified when included, 2 when omitted, or implied when field is specified.

Example: [1 2 3 4] is the polynomial $1+2x+3x^2+4x^3$ in GF(5) expressed as a row vector.

Data Types: double | char | string

p — Prime number

2 (default) | prime number

Prime number, specified as a prime number.

Data Types: double

field — *m*-tuple of all elements in GF(p^m)

matrix

m-tuple of all elements in GF(p^m), specified as a matrix. `field` is the matrix listing all elements of GF(p^m), arranged relative to the same primitive element. To generate the *m*-tuple of all elements in GF(p^m), use

```
field =gftuple([-1:p^m-2]',m,p)
```

The coefficients, specified in exponential format, represent the field elements in GF(p^m). For an explanation of these formats, see “Representing Elements of Galois Fields”.

Data Types: `double`

Output Arguments

q — Galois field polynomial
row vector

Galois field polynomial, returned as a row vector of the polynomial coefficients in order of ascending powers. `q` is the quotient from the division of `b` by `a` and is in the same Galois field as the input polynomials.

r — Division remainder
scalar | row vector

Division remainder, returned as a scalar or a row vector of the polynomial coefficients in order of ascending powers. `r` is the remainder resulting from the division of `b` by `a`.

Tips

- The gfdeconv function performs computations in GF(p^m), where *p* is prime, and *m* is a positive integer. It divides polynomials over a Galois field. To work in GF(2^m), use the `deconv` function of the `gf` object with Galois arrays. For details, see “Multiplication and Division of Polynomials”.
- To divide elements of a Galois field, you can also use `gfdiv` instead of `gfdeconv`. Algebraically, dividing polynomials over a Galois field is equivalent to deconvolving vectors containing the coefficients of the polynomials. This deconvolution operation uses arithmetic over the same Galois field.

See Also

Functions
`gfadd` | `gfconv` | `gfdiv` | `gfsub` | `gftuple`

Topics
“Tips” on page 2-408
“Character Representation of Polynomials”
“Representing Elements of Galois Fields”
“Multiplication and Division of Polynomials”

Introduced before R2006a

gfdiv

Divide elements of Galois field

Syntax

```
quot = gfdiv(b,a)
quot = gfdiv(b,a,p)
quot = gfdiv(b,a,field)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), apply the `./` operator to Galois arrays. For details, see “Example: Division”.

The `gfdiv` function divides elements of a Galois field. (To divide polynomials over a Galois field, use `gfdeconv` instead.)

`quot = gfdiv(b,a)` divides `b` by `a` in GF(2) element-by-element. `a` and `b` are scalars, vectors or matrices of the same size. Each entry in `a` and `b` represents an element of GF(2). The entries of `a` and `b` are either 0 or 1.

`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, 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” 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

gfconv | gfdeconv | gfmul | gftuple

Introduced before R2006a

gffilter (prime Galois field)

Filter data using polynomials over prime Galois field

Syntax

```
y = gffilter(b,a,x)
y = gffilter(b,a,x,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `filter` function with Galois arrays. For details, see “Filtering”.

`y = gffilter(b,a,x)` filters the data in vector `x` with the filter described by vectors `b` and `a`. The vectors `b`, `a` and `x` must be in GF(2), that is, be binary and `y` is also in GF(2).

`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`.

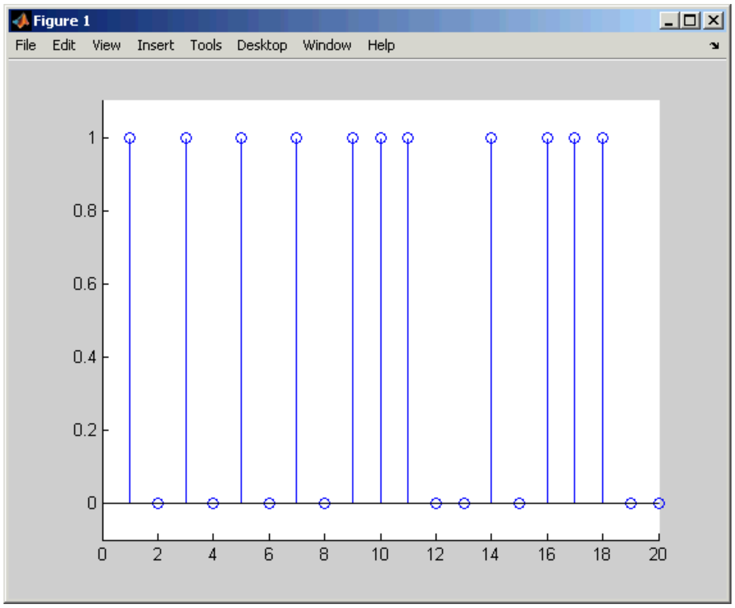
The vector `a` represents the degree-n_a polynomial

```
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

`filter` | `gfadd` | `gfconv`

Introduced before R2006a

gflineq

Find particular solution of Ax = b over prime Galois field

Syntax

```
x = gflineq(A,b)
x = gflineq(A,b,p)
[x,vld] = gflineq(...)
```

Description

Note This function performs computations in GF(p), where p is prime. To work in GF(2^m), apply the \ or / operator to Galois arrays. For details, see “Solving Linear Equations”.

x = gflineq(A,b) outputs a particular solution of the linear equation A x = b in GF(2). The elements in a, b and x are either 0 or 1. If the equation has no solution, then x is empty.

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, 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, x is empty.

[x,vld] = gflineq(...) returns a flag vld that indicates the existence of a solution. If vld = 1, the solution x exists and is valid; if vld = 0, 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)
```

The output is below.

```
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)
```

The output is below.

```
This linear equation has no solution.

x2 =

     []


vld2 =

     0
```

Algorithms

`gflineq` uses Gaussian elimination.

See Also

`conv` | `gfadd` | `gfconv` | `gfdiv` | `gfrank` | `gfroots`

Introduced before R2006a

gfminpol

Find minimal polynomial of Galois field element

Syntax

```
pol = gfminpol(k,m)
pol = gfminpol(k,m,p)
pol = gfminpol(k,prim_poly,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `minpol` function with Galois arrays. For details, see "Minimal Polynomials".

`pol = gfminpol(k,m)` produces a minimal polynomial for each entry in `k`. `k` must be either a scalar or a column vector. Each entry in `k` represents an element of GF(2^m) in exponential format. That is, `k` represents alpha`^k`, where alpha is a primitive element in GF(2^m). The *i*th row of `pol` represents the minimal polynomial of `k`(*i*). The coefficients of the minimal polynomial are in the base field GF(2) and listed in order of ascending exponents.

`pol = gfminpol(k,m,p)` finds the minimal polynomial of A^k over GF(`p`), where `p` is a prime number, `m` is an integer greater than 1, and A 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, `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, `pol` is a matrix having *len* rows; the rth row of `pol` gives the coefficients of the minimal polynomial of $A^{k(r)}$ in order of ascending powers.

`pol = gfminpol(k,prim_poly,p)` is the same as the first syntax listed, except that A is a root of the primitive polynomial for GF(p^m) specified by `prim_poly`. `prim_poly` is a polynomial character vector or 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".

See Also

`gfcosets` | `gfprimdf` | `gfroots`

Introduced before R2006a

gfmul

Multiply elements of 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 prime. To work in GF(2^m), apply the `.*` operator to Galois arrays. For details, see “Example: Multiplication”.

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, 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” for an explanation of these formats. If `a` and `b` are matrices of the same size, the function treats each element independently.

Examples

“Arithmetic in Galois Fields” 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)
```

The output is

```
a =

     6
```

See Also

`gfadd` | `gfdeconv` | `gfdiv` | `gfsub` | `gftuple`

Introduced before R2006a

gfpretty

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 `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 `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, the spacing in the display might not look correct.

Examples

Display statements about the elements of GF(81).

```
p = 3; m = 4;
ii = randi([1,p^m-2],1,1); % Random exponent for prim element
primpolys = gfprimfd(m,'all',p);
[rows, cols] = size(primpolys);
jj = randi([1,rows],1,1); % 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

                                         3    4
                                2 + 2 X  + X
then the element

                                       22
```

```
                                      A
can also be expressed as

                                       2    3
                                  2 + A  + A
```

See Also

gfprimdf | gftuple

Introduced before R2006a

gfprimck

Check whether polynomial over Galois field is primitive

Syntax

```
ck = gfprimck(a)
ck = gfprimck(a,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. If you are working in GF(2^m), use the `isprimitive` function. For details, see *Finding Primitive Polynomials* in “Primitive Polynomials and Element Representations”.

`ck = gfprimck(a)` checks whether the degree-m GF(2) polynomial `a` is a primitive polynomial for GF(2^m), where m = length(`a`) - 1. The output `ck` is as follows:

- -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)

`ck = gfprimck(a,p)` checks whether the degree-m GF(P) polynomial `a` is a primitive polynomial for GF(p^m). p is a prime number.

`a` is either a polynomial character vector or a row vector representing the polynomial by listing its coefficients in ascending order. For example, in GF(5), `'4 + 3x + 2x^3'` and `[4 3 0 2]` are equivalent.

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” contains examples.

Algorithms

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.

References

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

[2] Krogsgaard, K., and T., Karp, *Fast Identification of Primitive Polynomials over Galois Fields: Results from a Course Project*, ICASSP 2005, Philadelphia, PA, 2004.

See Also

`gfadd` | `gfminpol` | `gfprimdf` | `gfprimfd` | `gftuple`

Introduced before R2006a

gfprimdf

Provide default primitive polynomials for Galois field

Syntax

```
pol = gfprimdf(m)
pol = gfprimdf(m,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `primpoly` function. For details, see *Finding Primitive Polynomials* in “Primitive Polynomials and Element Representations”.

`pol = gfprimdf(m)` outputs the default primitive polynomial `pol` in GF(2^m).

`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

Find Default Primitive Polynomial for GF(52)

Find the default primitive polynomial for GF(52) by using the `gfprimdf` function. Then use the `gfpretty` function to display it in polynomial format.

```
pol = gfprimdf(2,5)
```

```
pol = 1×3

     2     1     1
```

```
gfpretty(pol)
```

```
                                   2
                       2 + X + X
```

Find Default Primitive Polynomials for Range of Galois Fields

Find the default primitive polynomials for a range of Galois fields by using the `gfprimdf` function.

Use the `gfpretty` function to display the default primitive polynomial for each of the fields GF(3*m*), where the range for *m* is [3, 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

gfminpol | gfprimck | gfprimfd | gftuple

Introduced before R2006a

gfprimfd

Find primitive polynomials for Galois field

Syntax

```
pol = gfprimfd(m,opt,p)
```

Description

Note This function performs computations in GF(p^m), where p is prime. To work in GF(2^m), use the `primpoly` function. For details, see *Finding Primitive Polynomials* in “Primitive Polynomials and Element Representations”.

- If `m` = 1, `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, `pol` = [1 1]. If `m` > 1, 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 =

     []
```

Algorithms

`gfprimfd` tests for primitivity using `gfprimck`. If *`opt`* is `'min'`, `'max'`, or omitted, 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

`gfminpol` | `gfprimck` | `gfprimdf` | `gftuple`

Introduced before R2006a

gfrank

Compute rank of matrix over Galois field

Syntax

```
rk = gfrank(A,p)
```

Description

Note This function performs computations in GF(p) where p is prime. If you are working in GF(2^m), use the `rank` function with Galois arrays. For details, see “Computing Ranks”.

`rk = gfrank(A,p)` calculates the rank of the matrix `A` in GF(`p`), where `p` is a prime number.

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
```

Algorithms

`gfrank` uses an algorithm similar to Gaussian elimination.

Introduced before R2006a

gfrepcov

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

   $$\mathrm{A\_0} + \mathrm{A\_1}x + \mathrm{A\_2}x^2 + \cdots + \mathrm{A\_(m\text{-}1)}x^{m-1}$$

   Each entry `A_k` is either one or zero.
2. [A_0 A_1 A_2 ... A_(m-1)] represents the polynomial

   $$x^{\mathrm{A\_0}} + x^{\mathrm{A\_1}} + x^{\mathrm{A\_2}} + \cdots + x^{\mathrm{A\_(m\text{-}1)}}$$

   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.

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`

Introduced before R2006a

gfroots

Find roots of polynomial over 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 prime. To work in GF(2^m), use the `roots` function with Galois arrays. For details, see “Roots of Polynomials”.

For all syntaxes, `f` is a polynomial character vector or 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. `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.

`[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 preceding paragraph. `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”.

Examples

“Roots of Polynomials” contains a description and example of the use of `gfroots`.

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 `alph`. (The output is omitted here.) Because prim_poly is both the primitive polynomial and the polynomial whose roots are sought, alph 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

Introduced before R2006a

gfsub

Subtract polynomials over 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 prime. To work in GF(2^m), apply the - operator to Galois arrays of equal size. For details, see "Example: Addition and Subtraction".

`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, the function treats each row independently. Alternatively, `a` and `b` can be represented as polynomial character vectors.

`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), extra zeros are added at the end; if it has more than `len` entries, 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" for an explanation of these formats. If `a` and `b` are matrices of the same size, the function treats each element independently.

Examples

Subtract Two GF Arrays

Calculate $(2 + 3x + x^2) - (4 + 2x + 3x^2)$ over GF(5).

```
x = gfsub([2 3 1],[4 2 3],5)
```

```
x = 1×3

     3     1     3
```

Subtract the two polynomials and display the first two elements.

```
y = gfsub([2 3 1],[4 2 3],5,2)
```

```
y = 1×2

     3     1
```

For prime number p and exponent m, create a matrix listing all elements of GF(p^m) given primitive polynomial $2 + 2x + x^2$.

```
p = 3;
m = 2;
primpoly = [2 2 1];
field = gftuple((-1:p^m-2)',primpoly,p);
```

Subtract A^4 from A^2. The result is A^7.

```
g = gfsub(2,4,field)
```

```
g = 7
```

See Also

gfadd | gfconv | gfdeconv | gfdiv | gfmul | gftuple

Introduced before R2006a

gftable

Generate 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, which can be either a polynomial character vector or an integer. prim_poly represents a primitive polynomial for GF(2^m), where 1 < *m* < 16, using the format described in “Primitive Polynomials and Element Representations”. The function places the file, called userGftable.mat, in your current working folder. 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, 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 expect t3 to be similar to t1 and to be significantly smaller than t2, assuming that you do not already have a userGftable.mat file that includes the (m, prim_poly) pair (8, 501). Notice that before executing the gftable command, MATLAB displays a warning and that after executing gftable, there is no warning. By executing the gftable command you save the GF table for faster calculations.

```
% 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

Topics

“Speed and Nondefault Primitive Polynomials”

Introduced before R2006a

gftrunc

Minimize length of 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, the polynomial has degree d. The row vector `c` omits these high-order zeros and thus has length d + 1.

Examples

Truncate Row-Vector Representation of Galois Field Polynomial

Use `gftrunc` to truncate the row-vector representation of $x^2 + 2x^3 + 3x^4 + 4x^7 + 5x^8$, removing nonsignificant zero-valued elements.

```
vec = [0 0 1 2 3 0 0 4 5 0 0]
```

```
vec = 1×11

     0     0     1     2     3     0     0     4     5     0     0
```

```
gfpretty([vec])
```

```
                         2      3      4      7      8
                        X  + 2 X  + 3 X  + 4 X  + 5 X
```

Zeros are removed from the end of the row-vector representation, but not from the beginning or middle of the row vector.

```
c = gftrunc([0 0 1 2 3 0 0 4 5 0 0])
```

```
c = 1×9

     0     0     1     2     3     0     0     4     5
```

```
gfpretty(c)
```

```
                         2      3      4      7      8
                        X  + 2 X  + 3 X  + 4 X  + 5 X
```

See Also

`gfadd` | `gfconv` | `gfdeconv` | `gfsub` | `gftuple`

Introduced before R2006a

gftuple

Simplify or convert Galois field element formatting

Syntax

```
tp = gftuple(a,m)
tp = gftuple(a,prim_poly)
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 prime. To perform equivalent computations in GF(2^m), apply the `.^` operator and the `log` function to Galois arrays. For more information, see “Example: Exponentiation” and “Example: Elementwise Logarithm”.

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”.

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, 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, 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)` 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(`2^m`), where `m` is a positive integer.

tp = gftuple(a,prim_poly) is the same as the syntax above, except that prim_poly is a polynomial character vector or a row vector that lists the coefficients of a degree m primitive polynomial for GF(2^m) in order of ascending exponents.

tp = gftuple(a,m,p) is the same as tp = gftuple(a,m) except that 2 is replaced by a prime number p.

tp = gftuple(a,prim_poly,p) is the same as tp = gftuple(a,prim_poly) except that 2 is replaced by a prime number p.

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 character vector; 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

- “List of All Elements of a Galois Field” (end of section)
- “Converting to Simplest Polynomial Format”

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)
tp2 = gftuple([0, 0, 0, 1],3,3)
```

The output is below.

```
tp1 =

     1     0     0
     0     1     0


tp2 =


     2     1     0
```

The outputs reflect that, according to the default primitive polynomial for GF(3^3), the relations below are true.

$\alpha^0 = 1 + 0\alpha + 0\alpha^2$

$\alpha^1 = 0 + 1\alpha + 0\alpha^2$

$0 + 0\alpha + 0\alpha^2 + \alpha^3 = 2 + \alpha + 0\alpha^2$

Algorithms

`gftuple` uses recursive callbacks to determine the exponential format.

See Also

`gfadd` | `gfconv` | `gfdeconv` | `gfdiv` | `gfmul` | `gfprimdf`

Introduced before R2006a

gfweight

Calculate minimum distance of 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 polynomial character vector or a row vector that gives the coefficients of the generator polynomial in order of ascending powers.

Examples

Calculate Minimum Distance of Linear Block Code

Calculate the minimum distance of a cyclic code using several methods.

Create the generate polynomial for a (7,4) cyclic code.

```
n = 7;
genpoly = cyclpoly(n,4);
```

Calculate the minimum distance for the cyclic code using:

1. Generator polynomial `genmat`
2. Parity check matrix `parmat`
3. Generator polynomial `genpoly`
4. Generator polynomial specified as a character vector

```
[parmat, genmat] = cyclgen(n,genpoly);
wts = [gfweight(genmat,'gen') gfweight(parmat,'par'),...
    gfweight(genpoly,n) gfweight('1+x2+x3',n)]
```

```
wts = 1×4

     3     3     3     3
```

See Also

bchgenpoly | cyclpoly | hammgen

Topics

“Block Codes”

Introduced before R2006a

gray2bin

(To be removed) Convert Gray-encoded positive integers to corresponding Gray-decoded integers

Note will be removed in a future release. Use the appropriate modulation object or function to remap constellation points instead. For more information, see “Compatibility Considerations”.

Syntax

```
y = gray2bin(x,modulation,M)
[y,map] = gray2bin(x,modulation,M)
```

Description

`y = gray2bin(x,modulation,M)` generates a Gray-decoded output vector or matrix `y` with the same dimensions as its input parameter `x`. `x` can be a scalar, vector, matrix, or 3-D array. `modulation` is the modulation type and must be `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`. `M` is the modulation order and must be an integer power of 2.

`[y,map] = gray2bin(x,modulation,M)` generates a Gray-decoded output `y` with its respective Gray-encoded constellation map, `map`.

You can use map output to label a Gray-encoded constellation. The map output gives the Gray-encoded labels for the corresponding modulation. See “Binary to Gray Symbol Mapping” on page 2-446 example.

Note If you are converting binary coded data to Gray-coded data and modulating the result immediately afterwards, you should use the appropriate modulation object or function with the `'Gray'` symbol ordering option, instead of `gray2bin`.

Examples

Binary to Gray Symbol Mapping

This example shows how to use the `bin2gray` and `gray2bin` functions to map integer inputs from a natural binary order symbol mapping to a Gray-coded signal constellation and vice versa, assuming 16-QAM modulation. In addition, a visual representation of the difference between Gray-coded and binary-coded symbol mappings is shown.

Convert Binary to Gray

Create a complete vector of 16-QAM integers.

```
M = 16;
x = (0:M-1);
```

Convert the input vector from a natural binary order to a Gray-encoded vector using `bin2gray`.

```
[y,mapy] = bin2gray(x,'qam',M);
```

Convert Gray to Binary

Convert the Gray-encoded symbols, `y`, back to a binary ordering using `gray2bin`.

```
z = gray2bin(y,'qam',M);
```

Verify that the original data, `x`, and the final output vector, `z`, are identical.

```
isequal(x,z)
```

```
ans = logical
   1
```

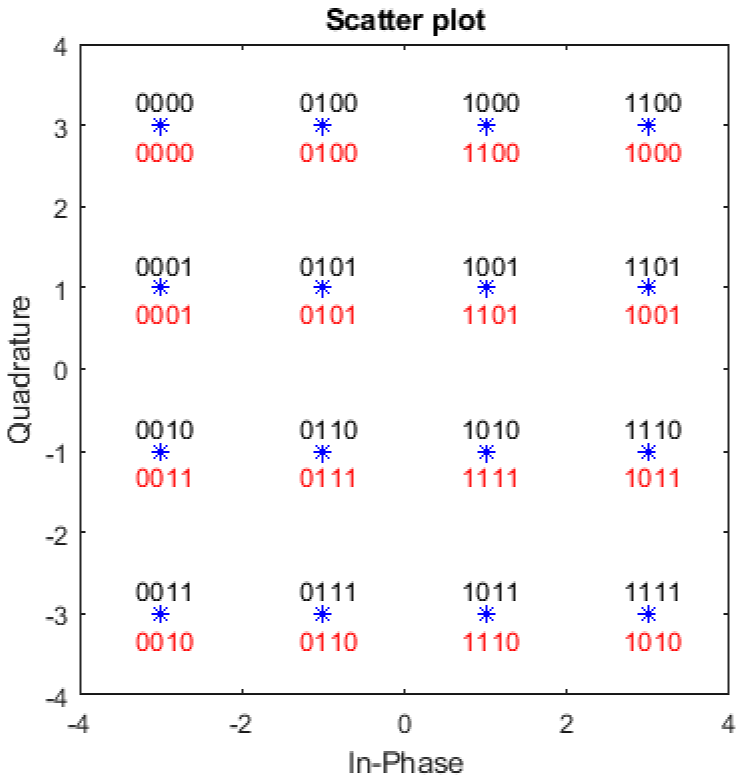
Show Symbol Mappings

To create a constellation plot showing the different symbol mappings, use the `qammod` function to find the complex symbol values.

```
sym = qammod(x,M);
```

Plot the constellation symbols and label them using the Gray (`y`) and binary (`z`) output vectors. The binary representation of the Gray-coded symbols is shown in black while the binary representation of the naturally ordered symbols is shown in red. Set the axes scaling so that all points are displayed.

```
scatterplot(sym,1,0,'b*');
for k = 1:16
    text(real(sym(k))-0.3,imag(sym(k))+0.3,...
        dec2base(mapy(k),2,4));

    text(real(sym(k))-0.3,imag(sym(k))-0.3,...
        dec2base(z(k),2,4),'Color',[1 0 0]);
end
axis([-4 4 -4 4])
```



Input Arguments

x — Gray-encoded data

vector | matrix

Gray-encoded input data, specified as a vector or matrix.

Data Types: `double`

`modulation` — Modulation type

`'qam'` | `'pam'` | `'fsk'` | `'dpsk'` | `'psk'`

Modulation type, specified as, `'qam'`, `'pam'`, `'fsk'`, `'dpsk'`, or `'psk'`

`M` — Modulation order

scalar

Modulation order, specified as an integer power of 2.

Data Types: `double`

Output Arguments

y — Gray-decoded data

vector | matrix

Gray-decoded data with the same size and dimensions input `x`.

`map` — Map of labels
vector

Map output to label a Gray-encoded constellation, specified as a vector with a length the size of the modulation order, `M`. The map gives the Gray-encoded labels for the corresponding modulation.

Compatibility Considerations

`gray2bin` will be removed
Not recommended starting in R2020a

`gray2bin` will be removed in a future release. Use the appropriate modulation object or function instead. If your workflow uses `bin2gray` or `gray2bin` with any of the modulations schemes in this table, follow the appropriate example.

Modulation	Old Functionality	Use This Instead

QAM (`qammod` and `qamdemod`)	`x = randi([0 63],1,100);`
	`y = bin2gray(x,'qam',64);`
	`z = qammod(y,64,'bin');`
	
	`x = 2*(randn(100,1)+1j*randn(100,1));`
	`y = qamdemod(x,64,'bin');`
	`z = gray2bin(y,'qam',64);`
		`x = randi([0 63],1,100);`
		`z = qammod(x,64,'gray');`
		
		`x = 2*(randn(100,1)+1j*randn(100,1));`
		`z = qamdemod(x,64,'gray')`

PAM (`pammod` and `pamdemod`)	`x = randi([0 63],1,100);`
	`y = gray2bin(x,'pam',64);`
	`z = pammod(y,64,pi/4,'bin');`
	
	`x = 2*(randn(100,1)+1j*randn(100,1));`
	`y = pamdemod(x,64,pi/4,'bin');`
	`z = bin2gray(y,'pam',64);`
		`x = randi([0 63],1,100);`
		`z = pammod(x,64,pi/4,'gray');`
		
		`x = 2*(randn(100,1)+1j*randn(100,1));`
		`z = pamdemod(x,64,pi/4,'gray')`

FSK (`fskmod` and `fskdemod`)	`x = randi([0 63],1,100);`
	`y = gray2bin(x,'fsk',64);`
	`z = fskmod(y,64,1,256,256,'cont','bin')`
	
	`x = 2*(randn(512,1)+1j*randn(512,1));`
	`y = fskdemod(x,64,1,256,256,'bin');`
	`z = bin2gray(y,'fsk',64)`
		`x = randi([0 63],1,100);`
		`z = fskmod(x,64,1,256,256,'cont','gray')`
		`x = 2*(randn(512,1)+1j*randn(512,1));`
		`z = fskdemod(x,64,1,256,256,'gray');`

DPSK (`dpskmod` and `dpskdemod`)	`x = randi([0 63],1,100);`
	`y = gray2bin(x,'dpsk',64);`
	`z = dpskmod(y,64,pi/4,'bin');`
	
	`x = 2*(randn(100,1)+1j*randn(100,1));`
	`y = dpskdemod(x,64,pi/4,'bin');`
	`z = bin2gray(y,'dpsk',64);`
		`x = randi([0 63],1,100);`
		`z = dpskmod(x,64,pi/4,'gray');`
		
		`x=2*(randn(100,1)+1j*randn(100,1));`
		`z = dpskdemod(x,64,pi/4,'gray');`

PSK (`pskmod` and `pskdemod`)	`x=randi([0 63],1,100);`
	`y=gray2bin(x,'psk',64);`
	`z=pskmod(y,64,0,'bin');`
	
	`x = 2*(randn(100,1)+1j*randn(100,1));`
	`y = pskdemod(x,64,0,'bin');`
	`z = bin2gray(y,'psk',64);`
		`x=randi([0 63],1,100);`
		`z=pskmod(x,64,0,'gray');`
		
		`x = 2*(randn(100,1)+1j*randn(100,1));`
		`z = pskdemod(x,64,0,'gray');`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`dpskdemod` | `dpskmod` | `fskdemod` | `fskmod` | `pamdemod` | `pammod` | `pskdemod` | `pskmod` | `qamdemod` | `qammod`

Topics

Gray Encoding a Modulated Signal

Introduced before R2006a

gsmCheckTimeMask

Inspect GSM burst against time mask

Syntax

```
gsmCheckTimeMask(gsmCfg)
gsmCheckTimeMask(gsmCfg,tn)

pf = gsmCheckTimeMask(gsmCfg)
pf = gsmCheckTimeMask(gsmCfg,tn)
```

Description

`gsmCheckTimeMask(gsmCfg)` plots the burst for the first time slot and the upper and lower time masks for the input GSM configuration object. The `RiseTime`, `RiseDelay`, `FallTime`, and `FallDelay` properties of the configuration object define the power level versus time characteristics. For more information, see “Time Mask” on page 2-465.

`gsmCheckTimeMask(gsmCfg,tn)` plots the burst for the specified time slot, `tn`.

`pf = gsmCheckTimeMask(gsmCfg)` returns a pass or fail result for the specified configuration object indicating compliance of the burst in the first time slot with the time mask defined in the GSM standard. For more information, see “Time Mask” on page 2-465.

`pf = gsmCheckTimeMask(gsmCfg,tn)` returns a pass or fail result indicating compliance of the burst in the specified time slot, `tn`.

Examples

Check GSM Burst Against Time Mask

Create a GSM uplink TDMA frame configuration object with default settings. The GSM TDMA frame has eight time slots. Check the burst in the first time slot against the time mask specified by the GSM standard.

Create a GSM uplink TDMA frame configuration object with default settings.

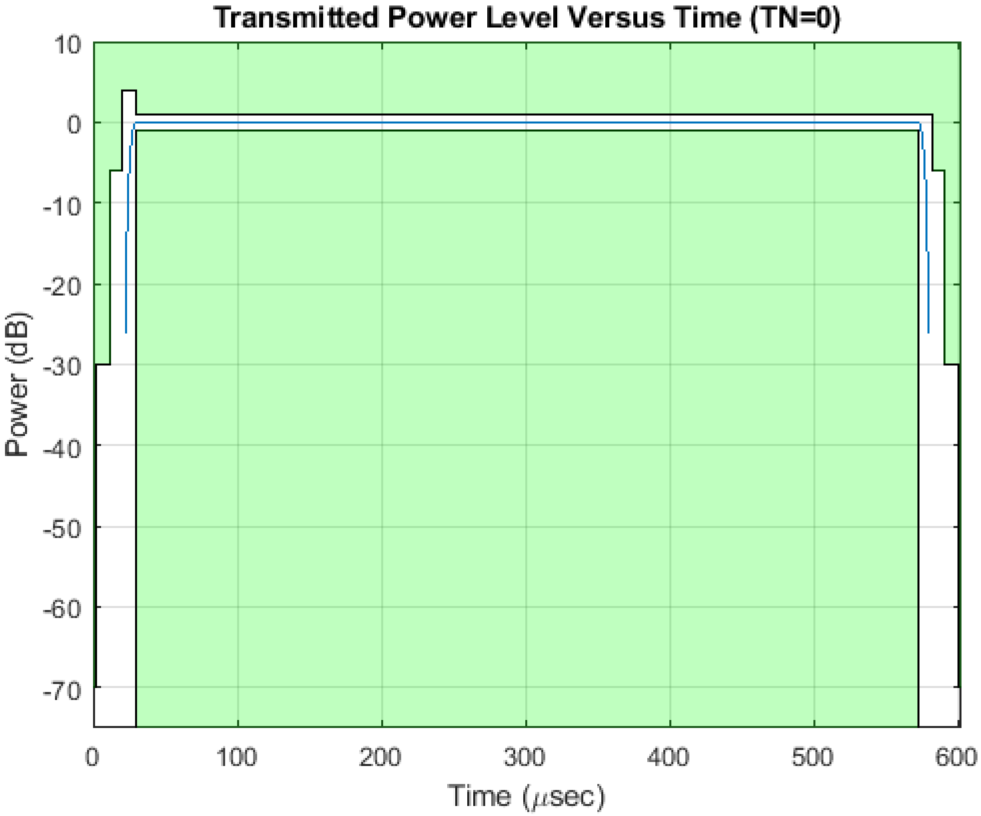
```
cfggsmul = gsmUplinkConfig;
```

Use the `gsmCheckTimeMask` function to view the time mask and verify that the configured rise and fall characteristics of the burst comply with the time mask specified in the GSM standard. Plot the GSM burst and time mask. When no time slot number is provided, the `gsmCheckTimeMask` function shows the first time slot, `TN=0`.

```
pf = gsmCheckTimeMask(cfggsmul);
if pf
    disp('Time mask test passed.')
else
    disp('Time mask test failed.')
end
```

```
Time mask test passed.
```

```
gsmCheckTimeMask(cfggsmul);
```



Adjust the rise time of the GSM uplink TDMA frame configuration object, specifying a value that causes a time mask failure.

```
cfggsmul.RiseTime = 5
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 5
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

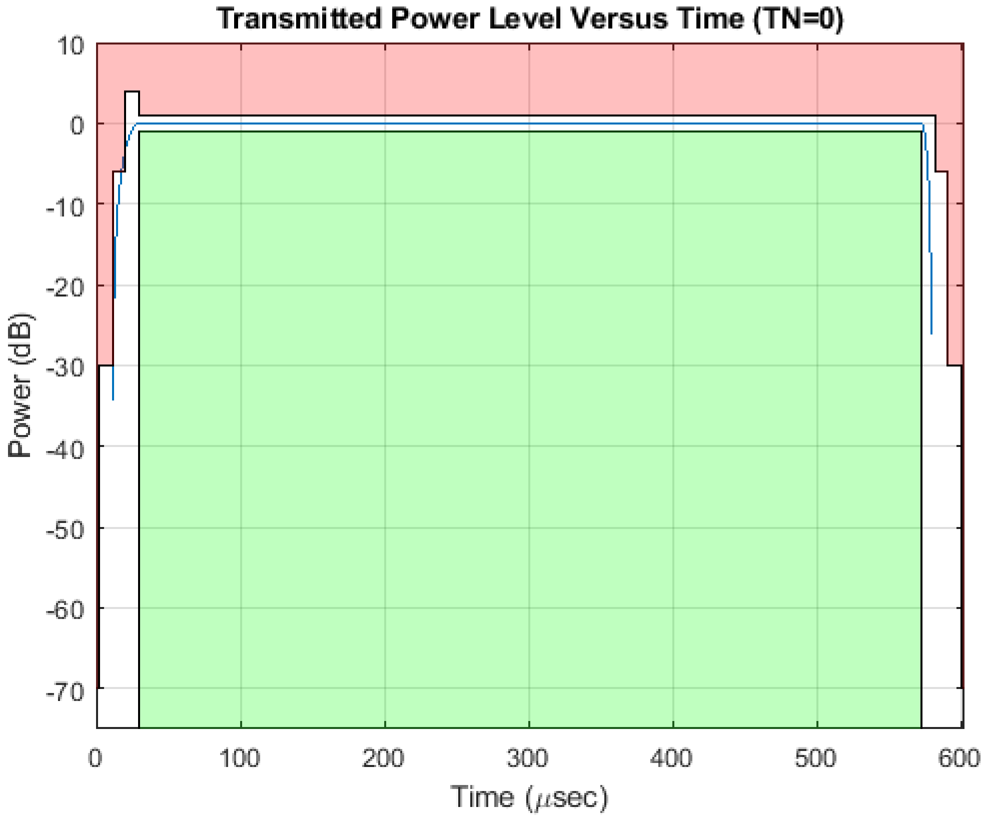
Use the gsmCheckTimeMask function to inspect the time mask of cfggsmul. The pass or fail result shows that the cfggsmul configuration now fails the time mask and the plot shows the upper time mask fails.

```
pf = gsmCheckTimeMask(cfggsmul);
if pf
```

```
    disp('Time mask test passed.')
else
    disp('Time mask test failed.')
end
```

```
Time mask test failed.
```

```
gsmCheckTimeMask(cfggsmul);
```



Check GSM Burst in Specified Time Slot Against Time Mask

Create a GSM uplink TDMA frame configuration object with default settings. The GSM TDMA frame has eight time slots. Check the burst in the specified time slot against the time mask specified by the GSM standard.

Create a GSM downlink TDMA frame configuration object with default settings.

```
cfggsmul = gsmDownlinkConfig;
```

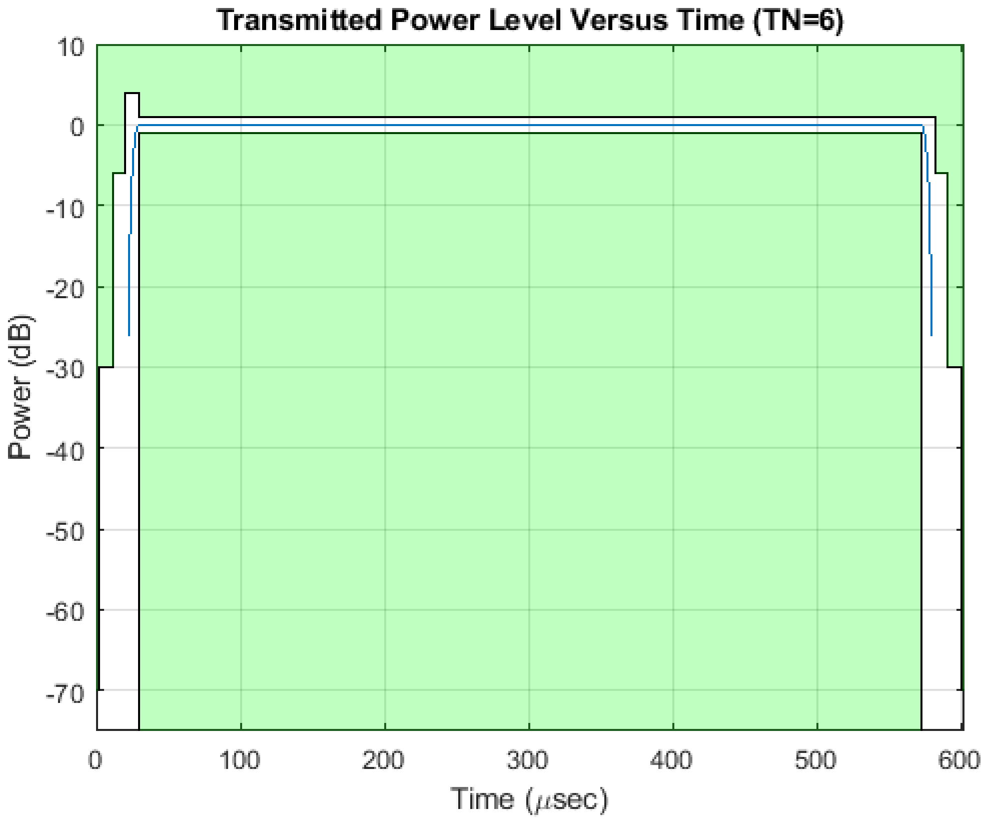
Use the `gsmCheckTimeMask` function to view the time mask and verify that the configured rise and fall characteristics of the burst in the specified time slot comply with the time mask specified by the GSM standard. Plot the GSM burst and time mask.

```
tn = 6; % Time slot number 6
pf = gsmCheckTimeMask(cfggsmul,tn);
```

```
if pf
    disp('Time mask test passed.')
else
    disp('Time mask test failed.')
end
```

```
Time mask test passed.
```

```
gsmCheckTimeMask(cfggsmul,tn);
```



Adjust the fall delay of the GSM downlink TDMA frame configuration object, specifying a value that causes a time mask failure.

```
cfggsmul.FallDelay = 4
```

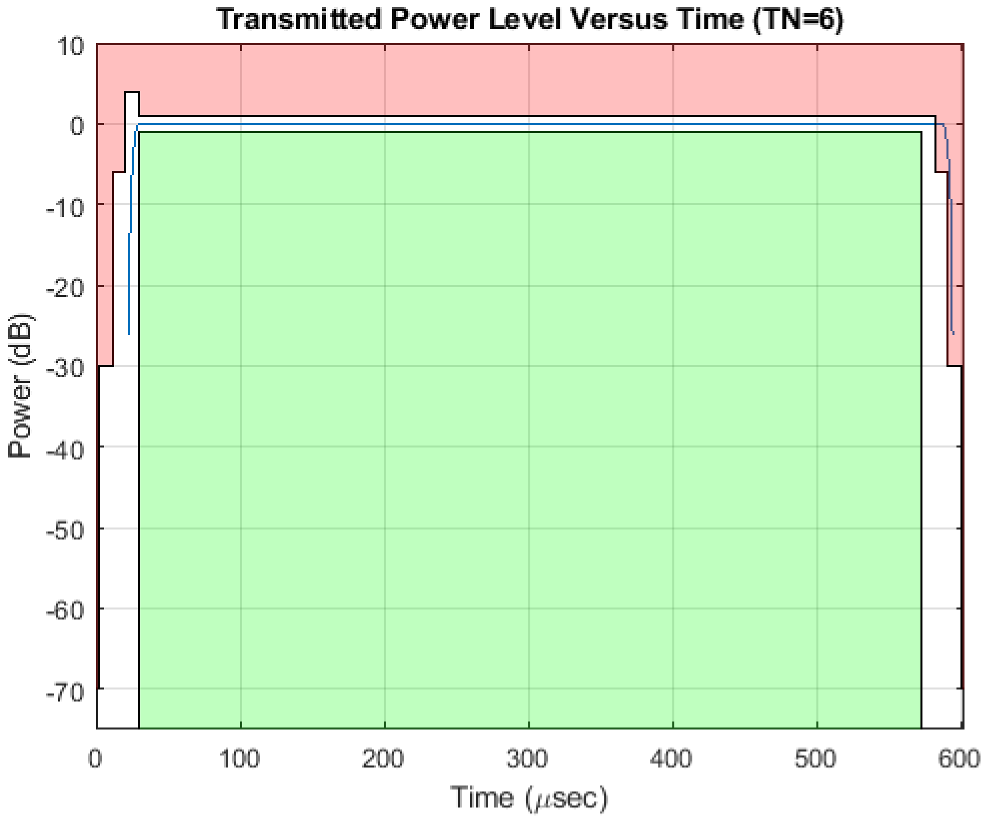
```
cfggsmul = 
  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 4
```

Use the `gsmCheckTimeMask` function to inspect the time mask of `cfggsmul`. The pass or fail result shows that the `cfggsmul` configuration now fails the time mask and the plot shows the upper time mask fails.

```
pf = gsmCheckTimeMask(cfggsmul,tn);
if pf
    disp('Time mask test passed.')
else
    disp('Time mask test failed.')
end
```

```
Time mask test failed.
```

```
gsmCheckTimeMask(cfggsmul,tn);
```



Check Time Mask for GSM Bursts

Create GSM downlink and uplink TDMA frame configuration objects that use the various burst types available.

- Normal bursts and bursts with no data are valid for downlink and uplink frames.
- Frequency correction, synchronization, and dummy bursts are valid in downlink frames only.
- Access bursts are valid in uplink frames only.

View time masks for the different burst types against the time mask specified by the GSM standard for the downlink and uplink frames.

Create a GSM downlink TDMA frame configuration object that configures the times slot bursts as [NB FB SB Dummy Off Off Off Off].

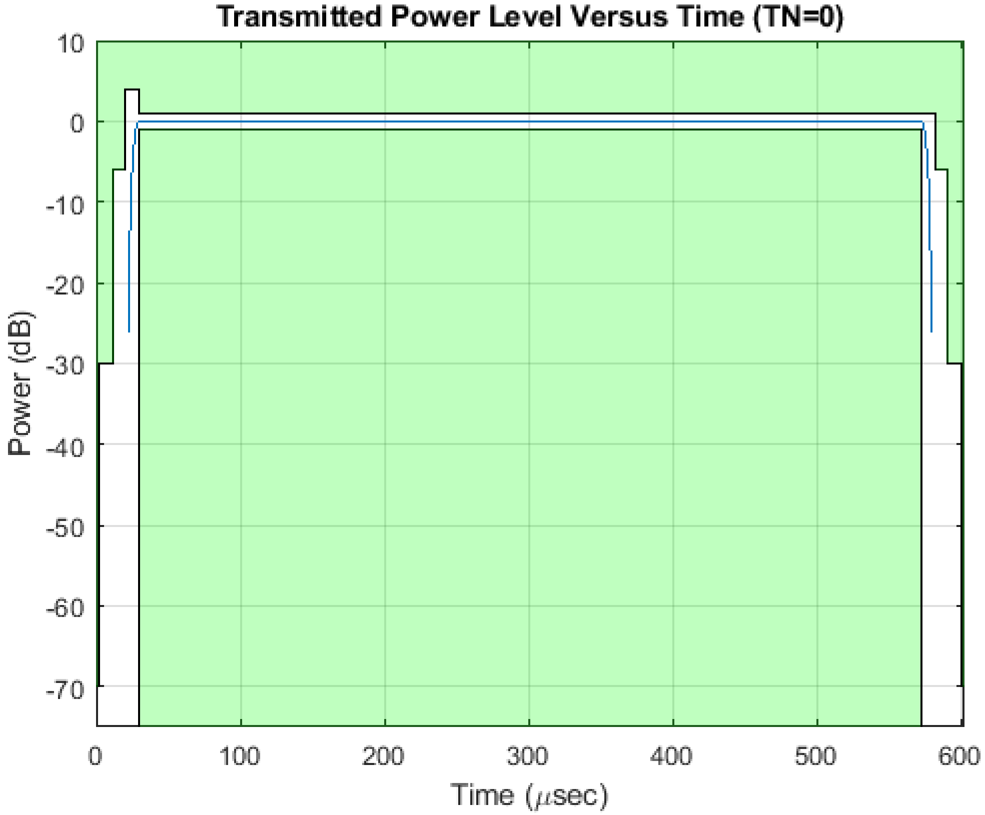
```
cfggsmdl = gsmDownlinkConfig('BurstType',["NB" "FB" "SB" "Dummy" "Off" "Off" "Off" "Off"])
```

```
cfggsmdl = 
  gsmDownlinkConfig with properties:

           BurstType: [NB    FB    SB    Dummy    Off    Off    Off    Off]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

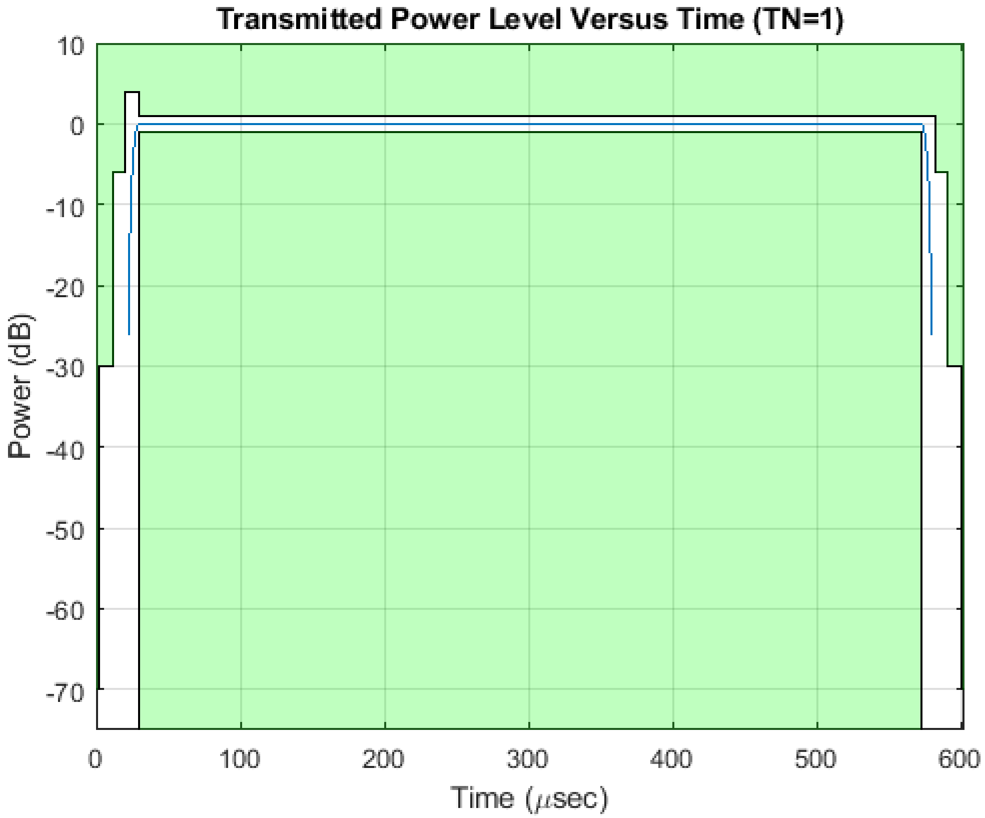
Use the gsmCheckTimeMask function to view the time mask for the different time slot burst types. For downlink GSM TDMA frames the same time mask limits applies for all burst types.

```
for tn = 0:4
    [dlbt,dlbtVal] = enumeration(cfggsmdl.BurstType);
    dlBurstInfo = ['Downlink (TN=',num2str(tn),'), BurstType: ',dlbtVal{tn+1}];
    disp(dlBurstInfo)
    gsmCheckTimeMask(cfggsmdl,tn);
end
```

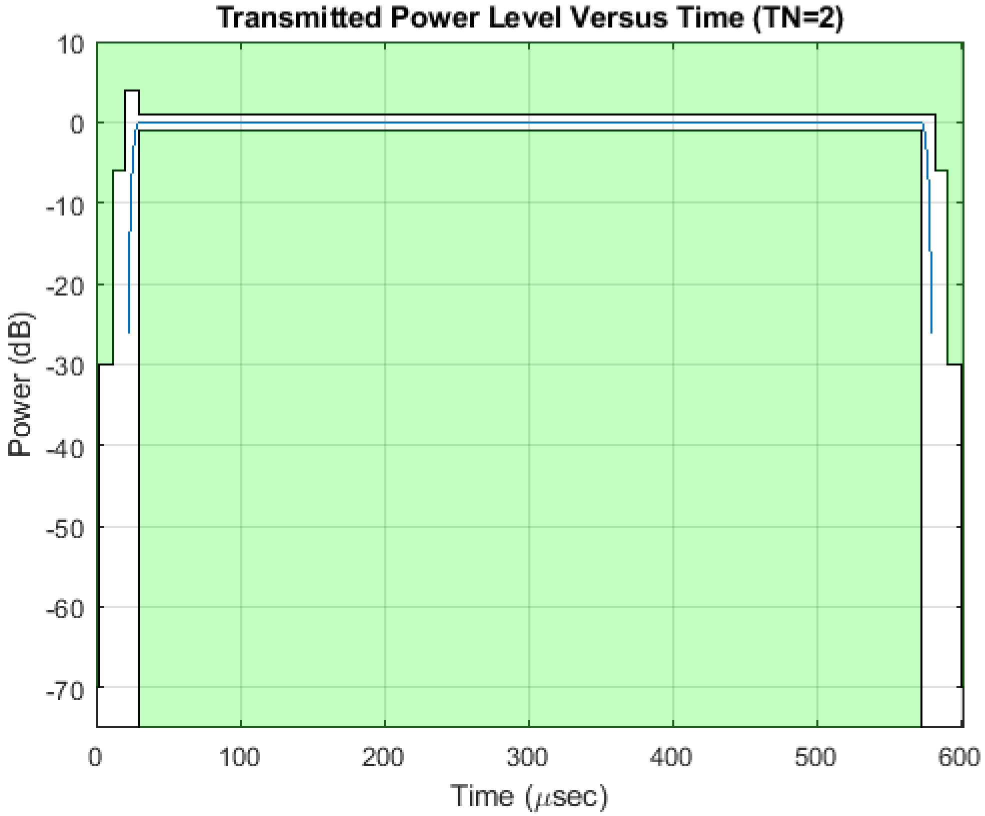
```
Downlink (TN=0), BurstType: NB
```



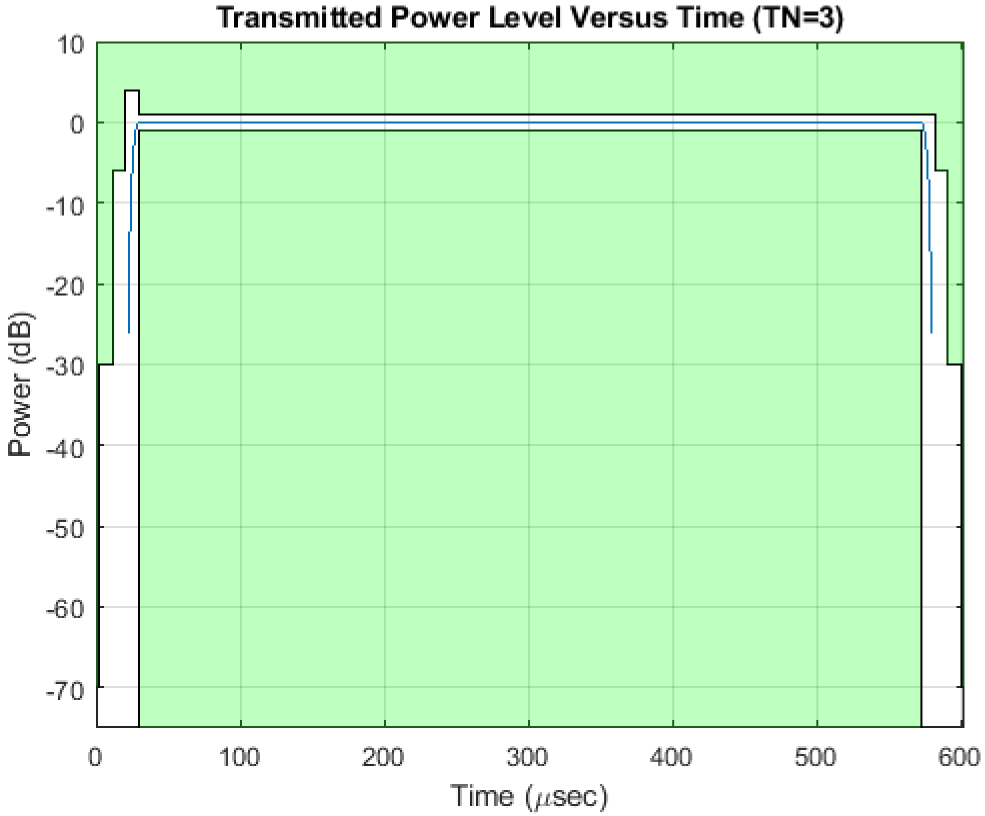
```
Downlink (TN=1), BurstType: FB
```



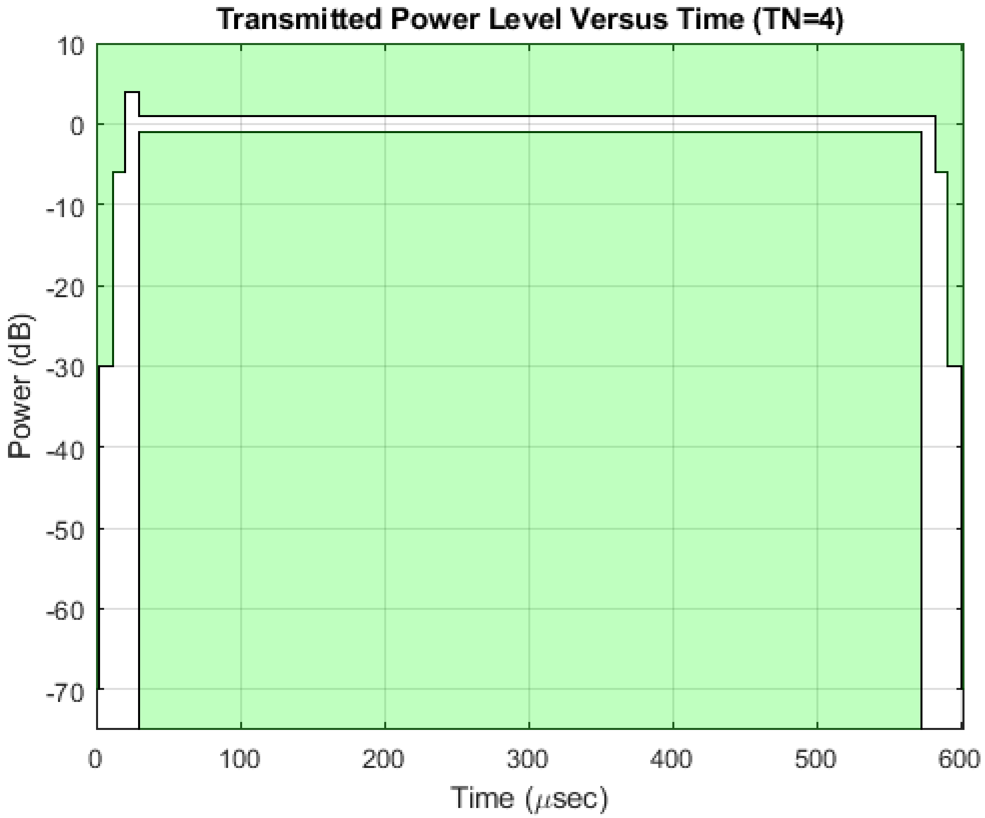
```
Downlink (TN=2), BurstType: SB
```



```
Downlink (TN=3), BurstType: Dummy
```



```
Downlink (TN=4), BurstType: Off
```



Create a GSM uplink TDMA frame configuration object that configures the times slot bursts as [NB AB Off Off Off Off Off Off].

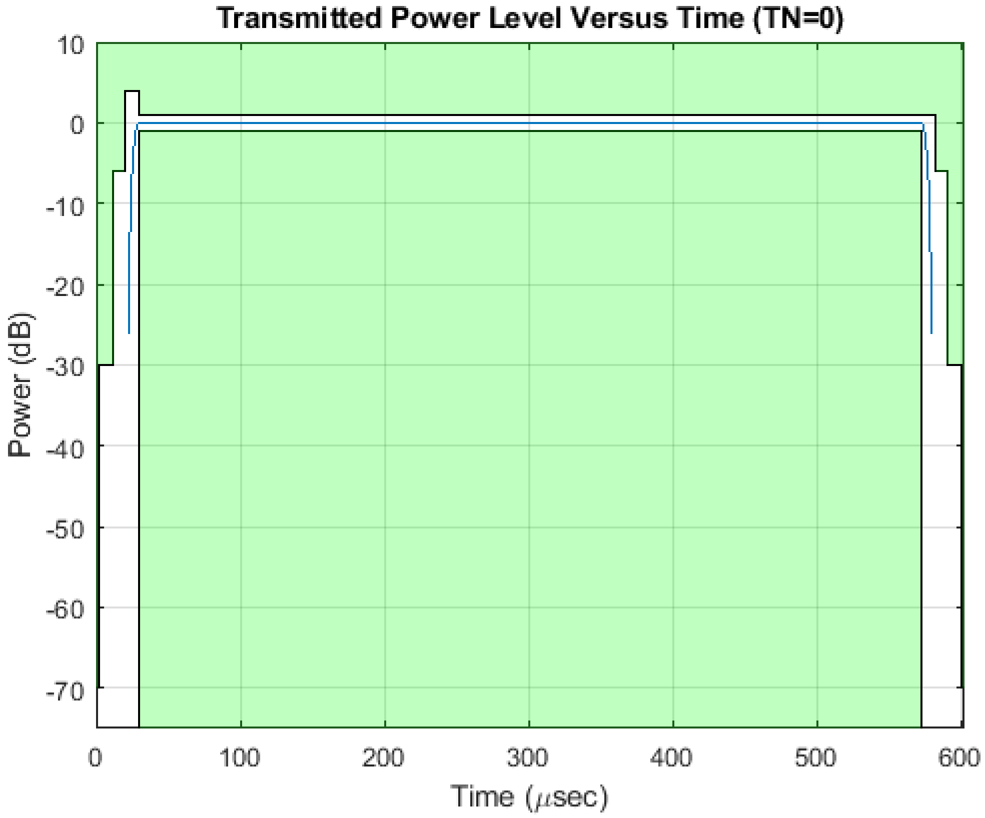
```
cfggsmul = gsmUplinkConfig('BurstType',["NB" "AB" "Off" "Off" "Off" "Off" "Off" "Off"])
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

          BurstType: [NB    AB    Off    Off    Off    Off    Off    Off]
    SamplesPerSymbol: 16
                TSC: [0 1 2 3 4 5 6 7]
        Attenuation: [0 0 0 0 0 0 0 0]
           RiseTime: 2
          RiseDelay: 0
           FallTime: 2
          FallDelay: 0
```

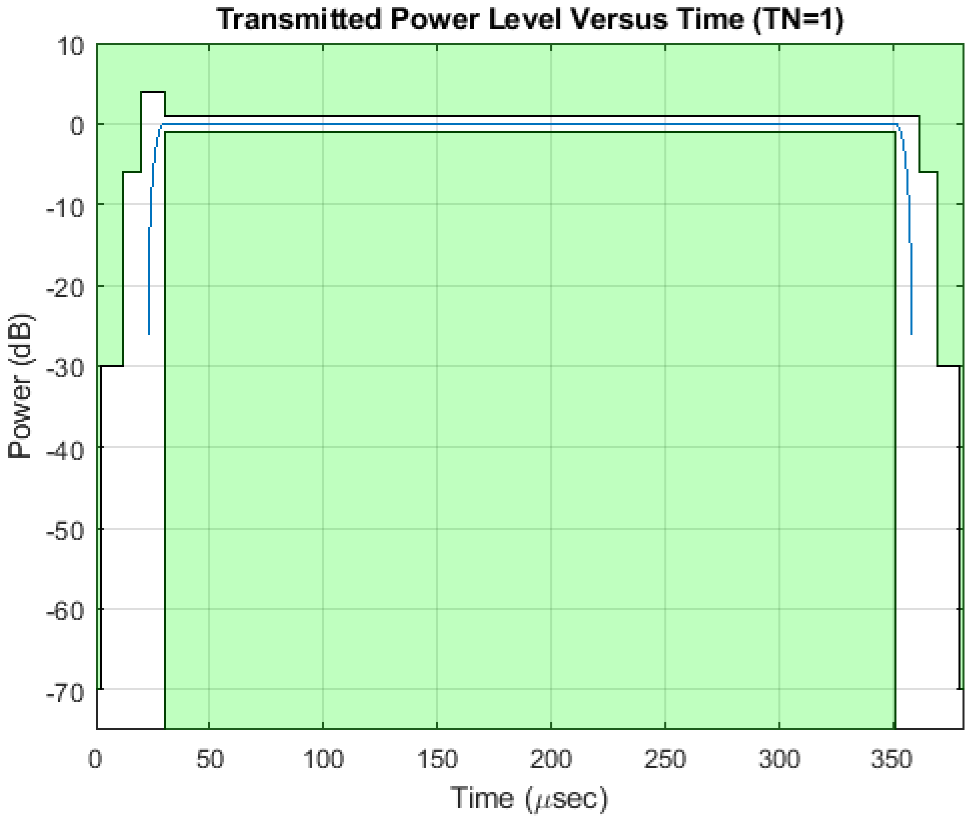
Use the `gsmCheckTimeMask` function to view the time masks for the different time slot burst types. For uplink GSM TDMA frames the access burst has a shorter time mask than the normal burst or no data burst.

```
for tn = 0:2
    [ulbt,ulbtVal] = enumeration(cfggsmul.BurstType);
    ulBurstInfo = ['Uplink (TN=',num2str(tn),'), BurstType: ',ulbtVal{tn+1}];
    disp(ulBurstInfo)
    gsmCheckTimeMask(cfggsmul,tn);
end
```

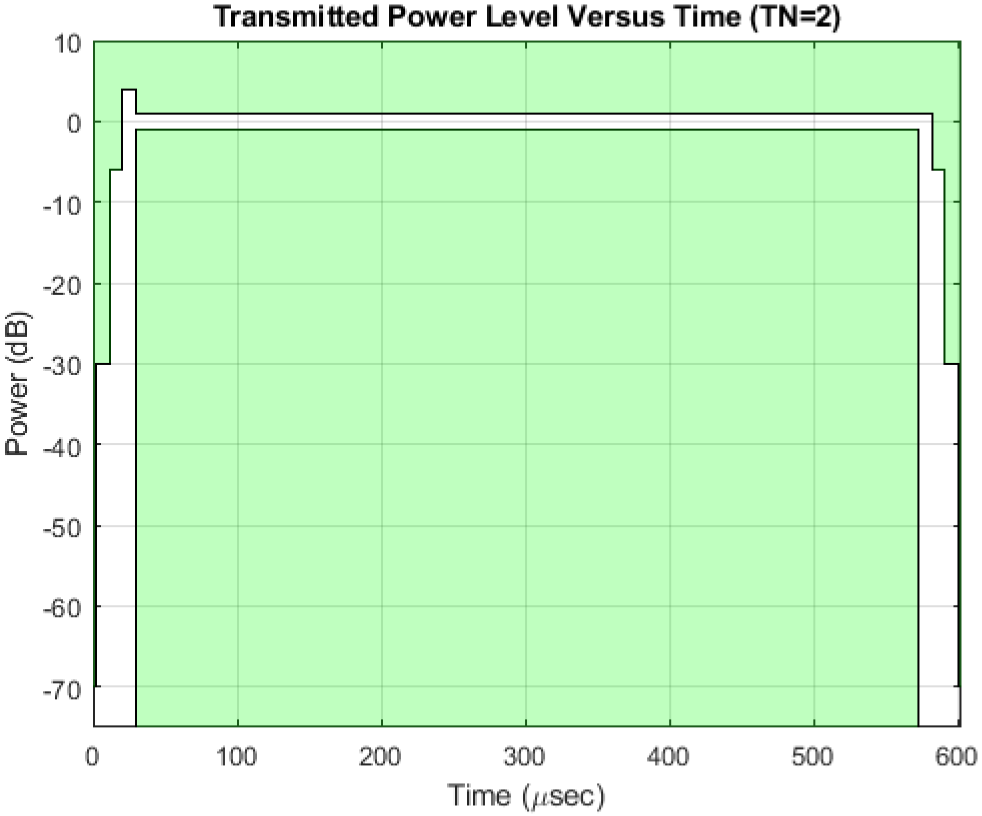
```
Uplink (TN=0), BurstType: NB
```



```
Uplink (TN=1), BurstType: AB
```



```
Uplink (TN=2), BurstType: Off
```



Input Arguments

gsmCfg — GSM configuration

gsmUplinkConfig object | gsmDownlinkConfig object

GSM configuration, specified as a gsmUplinkConfig or gsmDownlinkConfig object.

tn — Time slot number

0 (default) | integer in the range [0, 7]

Time slot number, specified as an integer in the range [0, 7].

Data Types: double

Output Arguments

pf — Pass or fail result

0 | 1

Pass or fail result, returned as:

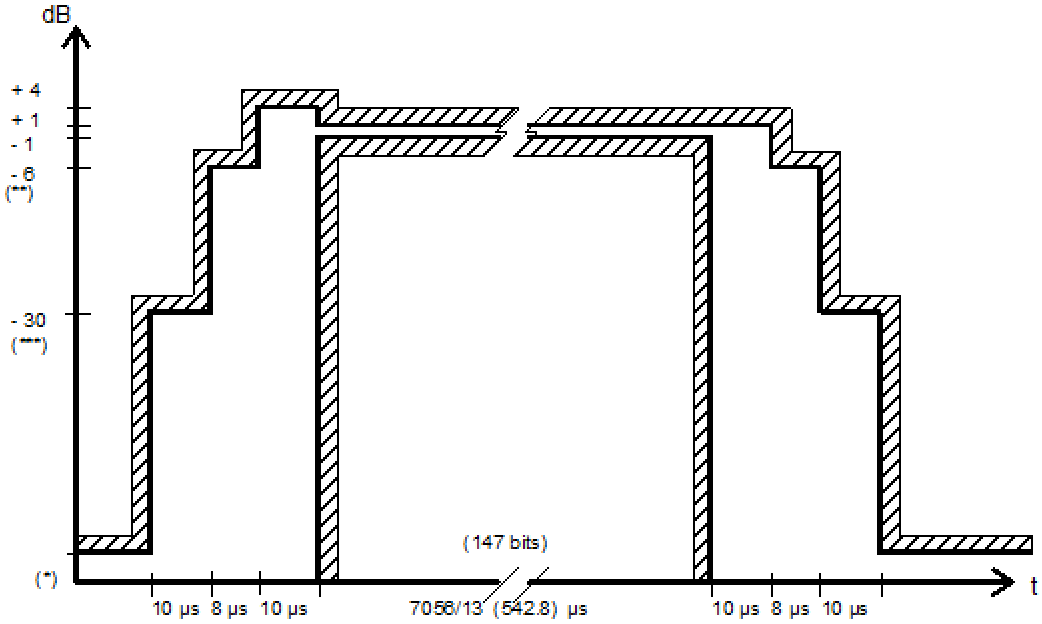
- 1 if the time mask passes
- 0 if the time mask fails

For more information, see “Time Mask” on page 2-465.

More About

Time Mask

The time mask defines the allowable transmitted power level versus time for time slot bursts in a GSM TDMA frame. This figure, from Annex B of TS 45.005, shows the upper and lower power limits for the time mask of a burst.



References

[1] 3GPP TS 45.005. "GSM/EDGE Radio transmission and reception." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Outputting a plot is not supported for code generation.

See Also

Objects

`gsmDownlinkConfig` | `gsmUplinkConfig`

Functions

`gsmFrame` | `gsmInfo`

Topics

"GSM TDMA Frame Parameterization for Waveform Generation"

Introduced in R2019b

gsmFrame

Create GSM waveform

Syntax

```
gsmWaveform = gsmFrame(gsmCfg)
gsmWaveform = gsmFrame(gsmCfg,numFrames)
```

Description

`gsmWaveform = gsmFrame(gsmCfg)` creates a GSM waveform with one TDMA frame based on the input GSM configuration object. The encrypted bit field of the transmission data bursts is filled with random data. For more information, see “GSM Frames, Time Slots, and Bursts” on page 2-473.

`gsmWaveform = gsmFrame(gsmCfg,numFrames)` creates a GSM waveform, with `numFrames` identically configured TDMA frames. In each frame, the encrypted bit field of the transmission data bursts is filled with random data. For more information, see “GSM Frames, Time Slots, and Bursts” on page 2-473.

Examples

Create GSM Uplink Waveform

Create a GSM uplink TDMA frame configuration object with default settings, and then create a GSM waveform containing one TDMA frame. GSM TDMA frames have eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM uplink TDMA frame configuration object with default settings.

```
cfggsmul = gsmUplinkConfig
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

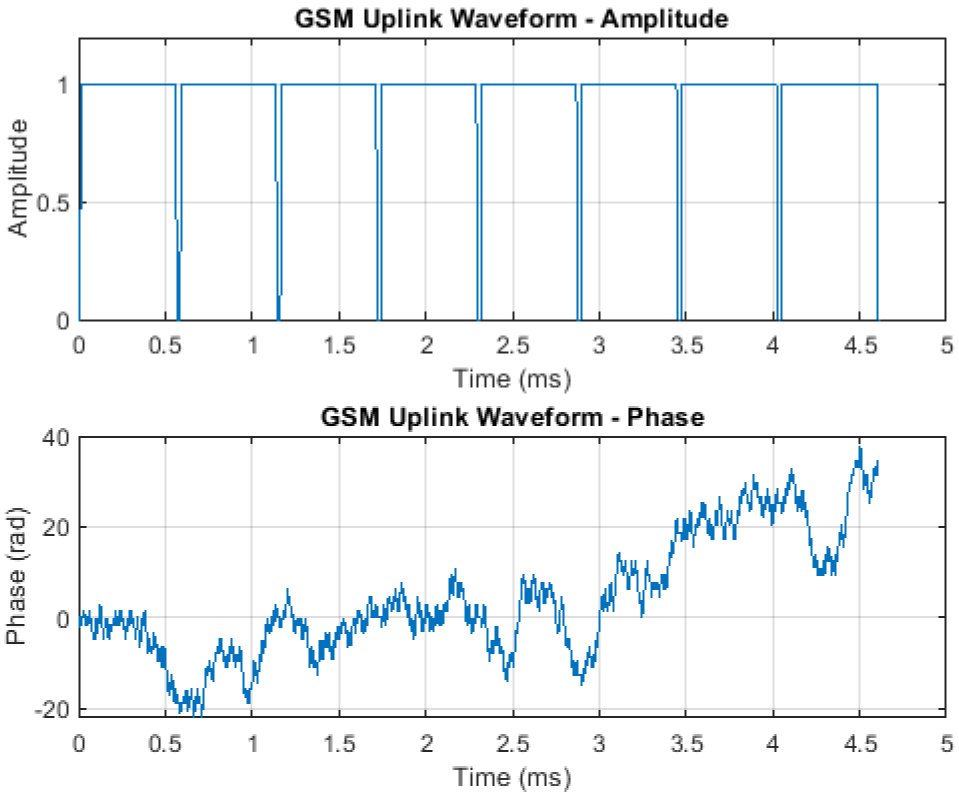
```
wfInfo = gsmInfo(cfggsmul)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmul);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Uplink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Create GSM Uplink Waveform Containing Five TDMA Frames

Create a GSM downlink TDMA frame configuration object with default settings, and then create a GSM waveform containing five TDMA frames. GSM TDMA frames have eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM uplink TDMA frame configuration object, specifying 3 dB of attenuation in the last time slot to help identify the end of each frame.

```
cfggsmul = gsmUplinkConfig('Attenuation',[0 0 0 0 0 0 0 3])
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 3]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the frame length in samples to a variable, `spf`, for use in computing the plot timescale.

```
wfInfo = gsmInfo(cfggsmul)
```

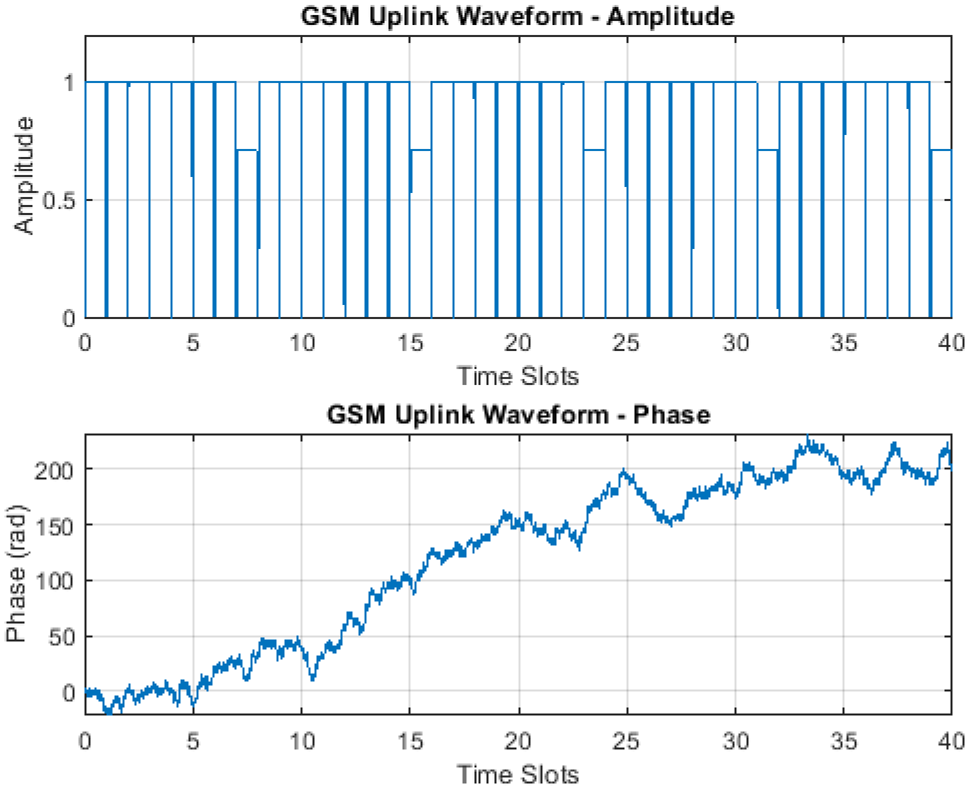
```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
spf = wfInfo.FrameLengthInSamples;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform. The last time slot of each frame is 3 dB less than the other time slots in that frame.

```
numFrames = 5;
waveform = gsmFrame(cfggsmul,numFrames);

t = 8*(0:length(waveform)-1)/spf;

numTS = 8*numFrames;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 numTS 0 1.2])
title('GSM Uplink Waveform - Amplitude')
```

```
xlabel('Time Slots')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time Slots')
ylabel('Phase (rad)')
```



Create GSM Downlink Waveform

Create a GSM downlink TDMA frame configuration object with default settings, and then create a GSM waveform containing one TDMA frame. The GSM TDMA frame has eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM downlink TDMA frame configuration object with default settings.

```
cfggsmdl = gsmDownlinkConfig
```

```
cfggsmdl = 
  gsmDownlinkConfig with properties:

          BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
```

```
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmDownlinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

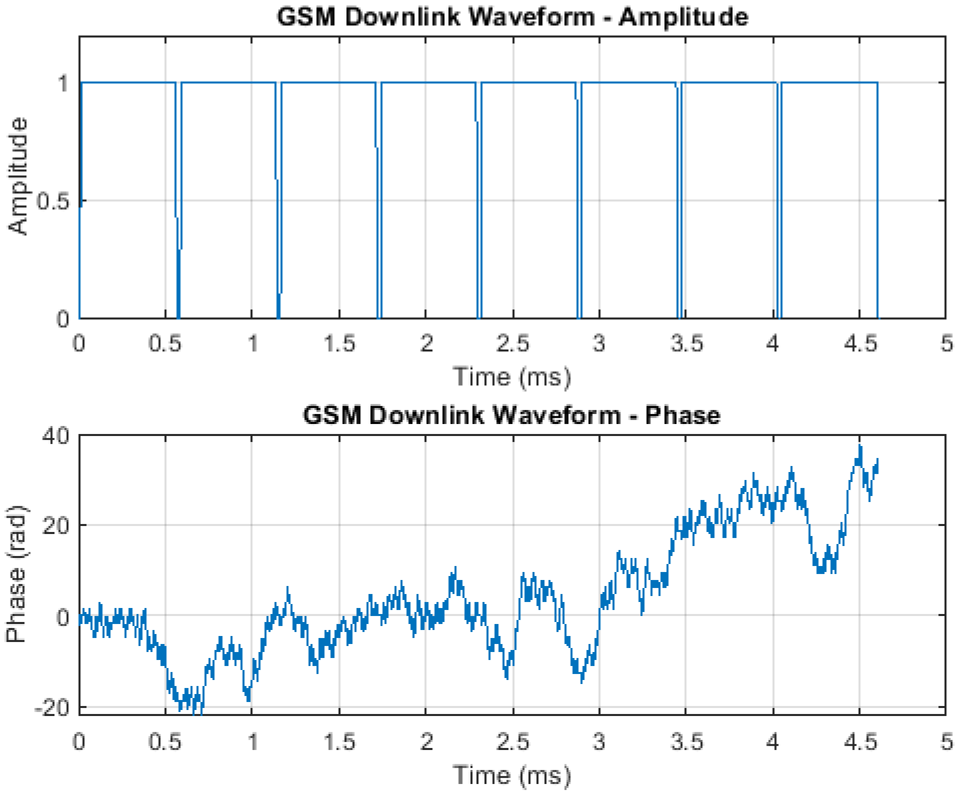
```
wfInfo = gsmInfo(cfggsmdl)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmdl);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Downlink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Downlink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Input Arguments

`gsmCfg` — GSM configuration

`gsmUplinkConfig` object | `gsmDownlinkConfig` object

GSM configuration, specified as a `gsmUplinkConfig` or `gsmDownlinkConfig` object.

`numFrames` — Number of TDMA frames

`16` (default) | positive integer

Number of TDMA frames in the waveform, specified as a positive integer.

Data Types: `double`

Output Arguments

`gsmWaveform` — Output time-domain waveform

complex-valued column vector

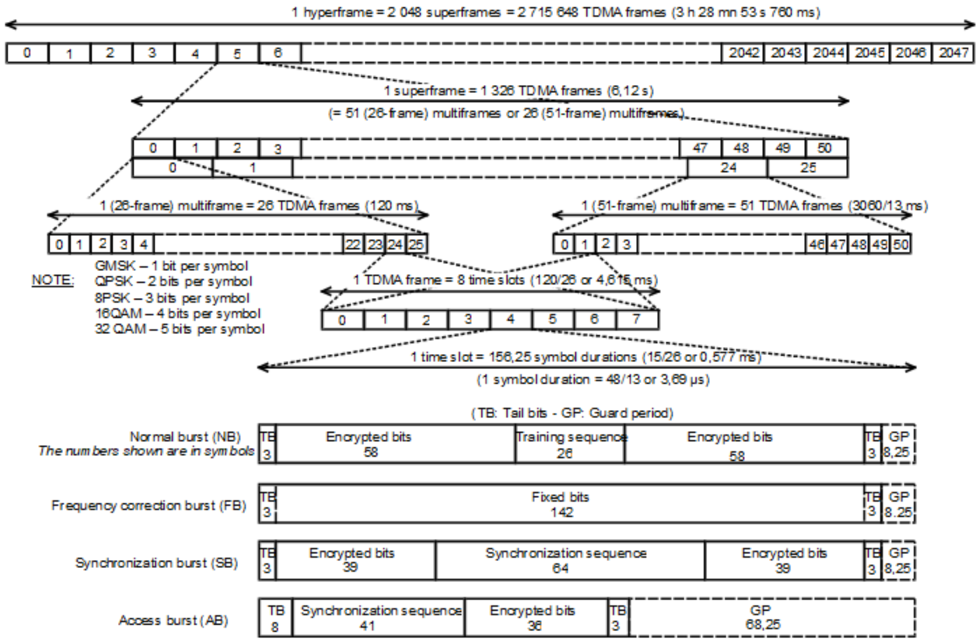
Output time-domain waveform, returned as a complex-valued column vector of length N_s, where N_s represents the number of time-domain samples. The function generates this waveform in the form of complex in-phase quadrature (IQ) samples.

More About

GSM Frames, Time Slots, and Bursts

In GSM, transmissions consist of TDMA frames. Each GSM TDMA frame consists of eight time slots. The transmission data content of a time slot is called a burst. As described in Section 5.2 of 3GPP TS 45.011, a GSM time slot has a 156.25-symbol duration when using the normal symbol period, which is a time interval of 15/26 ms or about 576.9 microseconds. A guard period of 8.25 symbols or about 30.46 microseconds separates each time slot. The GSM standards describes a symbol as one bit period. Since GSM uses GMSK modulation, there is one bit per bit period. The transmission timing of a burst within a time slot is defined in terms of the bit number (BN). The BN refers to a particular bit period within a time slot. The bit with the lowest BN is transmitted first. BN0 is the first bit period, and BN156 is the last quarter-bit period.

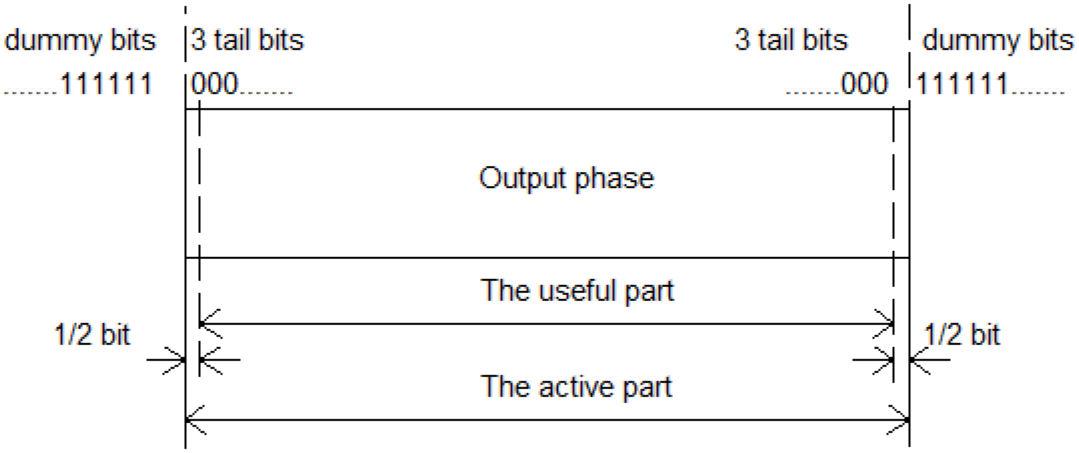
This image from 3GPP TS 45.011 shows the relationship between different frame types and the relationship between different burst types.


1 hyperframe = 2 048 superframes = 2 715 648 TDMA frames (3 h 28 mn 53 s 760 ms)
0
1
2
3
4
5
6
2042
2043
2044
2045
2046
2047
1 superframe = 1 326 TDMA frames (6,12 s)
(= 51 (26-frame) multiframes or 26 (51-frame) multiframes)
0
1
2
3
47
48
49
50
0
1
24
25
1 (26-frame) multiframe = 26 TDMA frames (120 ms)
0
1
2
3
4
22
23
24
25
1 (51-frame) multiframe = 51 TDMA frames (3060/13 ms)
0
1
2
3
46
47
48
49
50
NOTE:
GMSK – 1 bit per symbol
QPSK – 2 bits per symbol
8PSK – 3 bits per symbol
16QAM – 4 bits per symbol
32 QAM – 5 bits per symbol
1 TDMA frame = 8 time slots (120/26 or 4,615 ms)
0
1
2
3
4
5
6
7
1 time slot = 156,25 symbol durations (15/26 or 0,577 ms)
(1 symbol duration = 48/13 or 3,69 μs)
(TB: Tail bits - GP: Guard period)
Normal burst (NB)
The numbers shown are in symbols
TB 3
Encrypted bits 58
Training sequence 26
Encrypted bits 58
TB 3
GP 8,25
Frequency correction burst (FB)
TB 3
Fixed bits 142
TB 3
GP 8.25
Synchronization burst (SB)
TB 3
Encrypted bits 39
Synchronization sequence 64
Encrypted bits 39
TB 3
GP 8,25
Access burst (AB)
TB 8
Synchronization sequence 41
Encrypted bits 36
TB 3
GP 68,25

This table shows the supported burst types and their characteristics.

Burst Type	Description	Link Direction	Useful Duration
NB	Normal burst	Uplink/Downlink	147
FB	Frequency correction burst	Downlink	147
SB	Synchronization burst	Downlink	147
Dummy	Dummy burst	Downlink	147
AB	Access burst	Uplink	87
Off	No burst sent	Uplink/Downlink	0

Useful duration, described in Section 5.2.2 of 3GPP TS 45.002, is a characteristic of GSM bursts. The useful duration, or useful part, of a burst is defined as beginning halfway through BN0 and ending half a bit period before the start of the guard period. The guard period is the period between bursts in successive time slots. This figure, from Section 2.2 of 3GPP TS 45.004, shows the leading and trailing ½ bit difference between the useful and active parts of the burst.



For more information, see "GSM TDMA Frame Parameterization for Waveform Generation".

Training Sequence Code (TSC)

Normal bursts include a training sequence bits field assigned a bit pattern based on the specified TSC. For GSM, you can select one of these eight training sequences for normal burst type time slots.

Training Sequence Code (TSC)	Training Sequence Bits (BN61, BN62, …, BN86)
0	(0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,1,1)
1	(0,0,1,0,1,1,0,1,1,1,0,1,1,1,1,0,0,0,1,0,1,1,0,1,1,1)
2	(0,1,0,0,0,0,1,1,1,0,1,1,1,0,1,0,0,1,0,0,0,0,1,1,1,0)
3	(0,1,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,1,1,1,0)
4	(0,0,0,1,1,0,1,0,1,1,1,0,0,1,0,0,0,0,0,1,1,0,1,0,1,1)
5	(0,1,0,0,1,1,1,0,1,0,1,1,0,0,0,0,0,1,0,0,1,1,1,0,1,0)
6	(1,0,1,0,0,1,1,1,1,1,0,1,1,0,0,0,1,0,1,0,0,1,1,1,1,1)
7	(1,1,1,0,1,1,1,1,0,0,0,1,0,0,1,0,1,1,1,0,1,1,1,1,0,0)

For more information, see Section 5.2.3 in 3GPP TS 45.002.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`gsmDownlinkConfig` | `gsmUplinkConfig`

Functions

`gsmCheckTimeMask` | `gsmInfo`

Topics

"GSM TDMA Frame Parameterization for Waveform Generation"

Introduced in R2019b

gsmInfo

View GSM waveform information

Syntax

```
infostruct= gsmInfo(gsmCfg)
```

Description

`infostruct= gsmInfo(gsmCfg)` returns a structure containing characteristic waveform information for the input GSM configuration object.

Examples

View GSM Configuration Object Information

View information from downlink and uplink GSM configuration objects.

Create a GSM downlink configuration object with default settings and use `gsmInfo` to view the waveform information structure.

```
cfgDL = gsmDownlinkConfig;
infostructDL = gsmInfo(cfgDL)
```

```
infostructDL = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

Create a GSM uplink configuration object with default settings and use `gsmInfo` to view the waveform information structure.

```
cfgUL = gsmUplinkConfig;
infostructUL = gsmInfo(cfgUL)
```

```
infostructUL = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

Create GSM Uplink Waveform Containing Five TDMA Frames

Create a GSM downlink TDMA frame configuration object with default settings, and then create a GSM waveform containing five TDMA frames. GSM TDMA frames have eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM uplink TDMA frame configuration object, specifying 3 dB of attenuation in the last time slot to help identify the end of each frame.

```
cfggsmul = gsmUplinkConfig('Attenuation',[0 0 0 0 0 0 0 3])
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 3]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the frame length in samples to a variable, `spf`, for use in computing the plot timescale.

```
wfInfo = gsmInfo(cfggsmul)
```

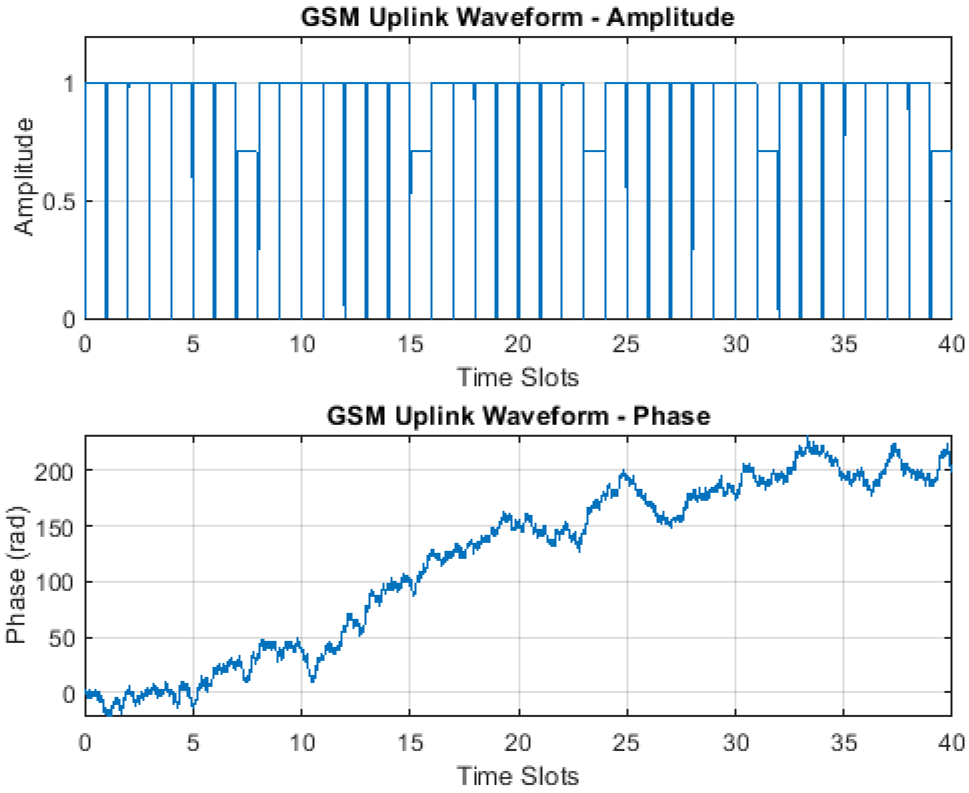
```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
spf = wfInfo.FrameLengthInSamples;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform. The last time slot of each frame is 3 dB less than the other time slots in that frame.

```
numFrames = 5;
waveform = gsmFrame(cfggsmul,numFrames);

t = 8*(0:length(waveform)-1)/spf;

numTS = 8*numFrames;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 numTS 0 1.2])
title('GSM Uplink Waveform - Amplitude')
xlabel('Time Slots')
ylabel('Amplitude')
subplot(2,1,2)
```

```
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time Slots')
ylabel('Phase (rad)')
```



Input Arguments

gsmCfg — GSM configuration

gsmUplinkConfig object | gsmDownlinkConfig object

GSM configuration, specified as a gsmUplinkConfig or gsmDownlinkConfig object.

Output Arguments

infostruct — Structure containing object information

struct

Structure containing these fields with information about the characteristic GSM waveform based on the input configuration object.

SymbolRate — GSM symbol rate

positive integer

GSM symbol rate in symbols per second, returned as a positive integer.

`SampleRate` — GSM sample rate
positive integer

GSM sample rate in samples per second, returned as a positive integer.

`BandwidthTimeProduct` — Product of bandwidth and symbol time of Gaussian pulse
positive integer

Product of bandwidth and symbol time of Gaussian pulse for the GMSK modulator, returned as a positive integer.

`BurstLengthInSymbols` — GSM burst length
positive scalar

GSM burst length in symbols, returned as a positive scalar.

`NumBurstsPerFrame` — Number of bursts in GSM TDMA frame
positive integer

Number of bursts in a GSM TDMA frame, returned as a positive integer.

`BurstLengthInSamples` — GSM burst length
positive integer

GSM burst length in samples, returned as a positive integer.

`FrameLengthInSamples` — GSM frame length
positive integer

GSM frame length in samples, returned as a positive integer.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`gsmDownlinkConfig` | `gsmUplinkConfig`

Functions

`gsmCheckTimeMask` | `gsmFrame`

Topics

“GSM TDMA Frame Parameterization for Waveform Generation”

Introduced in R2019b

hammgen

Parity-check and generator matrices for Hamming code

Syntax

```
h = hammgen(m)
h = hammgen(m,poly)
[h,g] = hammgen( ___ )
[h,g,n,k] = hammgen( ___ )
```

Description

`h = hammgen(m)` returns an `m`-by-*n* parity-check matrix, `h`, for a Hamming code of codeword length *n* = 2^m–1. The message length of the Hamming code is *n* – `m`. The binary primitive polynomial that the function uses to create the Hamming code is the default primitive polynomial in GF(`2^m`). For more details of this default polynomial, see the `gfprimdf` function.

`h = hammgen(m,poly)` specifies `poly`, a binary primitive polynomial for GF(2^m). The function uses `poly` to create the Hamming code.

`[h,g] = hammgen( ___ )` additionally returns a *k*-by-*n* generator matrix, `g`, that corresponds to the parity-check matrix `h`. Specify any of the input argument combinations from the previous syntaxes.

`[h,g,n,k] = hammgen( ___ )` also returns `n`, the codeword length and `k`, the message length, for the Hamming code.

Examples

Generate Hamming Code Parity-Check Matrix Using Default Primitive Polynomial

Generate a parity-check matrix, `h`, for a Hamming code of codeword length 7. The function uses the default primitive polynomial in GF(8) to create the Hamming code.

```
h = hammgen(3)
```

```
h = 3×7

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

Generate Hamming Code Parity-Check Matrix from Primitive Polynomials

Generate the parity-check matrices for the Hamming code of codeword length 15, specifying the primitive polynomials $1 + D + D^4$ and $1 + D^3 + D^4$ in GF(16).

```
h1 = hammgen(4,'1+D+D^4')
```

```
h1 = 4×15

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

```
h2 = hammgen(4,'1+D^3+D^4')
```

```
h2 = 4×15

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

Remove the embedded 4-by-4 identity matrices that is, the leftmost four columns in each parity-check matrix.

```
h1 = h1(:,5:end)
```

```
h1 = 4×11

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

```
h2 = h2(:,5:end)
```

```
h2 = 4×11

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

Verify that the two resulting matrices differ.

```
isequal(h1,h2)
```

```
ans = logical
   0
```

Generate Hamming Code Parity-Check and Generator Matrices

Generate the parity-check matrix, `h` and the generator matrix, `g` for the Hamming code of codeword length 7. Also return the codeword length, `n`, and the message length, `k` for the Hamming code. The function uses the default primitive polynomial in GF(8) to create the Hamming code.

```
[h,g,n,k] = hammgen(3)

h = 3×7

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


g = 4×7

     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
```

Input Arguments

`m` — Number of rows in parity-check matrix

integer greater than or equal to two

Number of rows in parity-check matrix, specified as an integer greater than or equal to two. The function uses this value to calculate the codeword length and the message length of the Hamming code.

Data Types: `double`

`poly` — Binary primitive polynomial in GF(2^m)

binary row vector | character vector | string scalar

Binary primitive polynomial in GF(2^m), specified as one of these values:

- Binary row vector of the polynomial coefficients in order of ascending powers
- Character vector
- String scalar

If `poly` is specified as a non-primitive polynomial, then the function `hammgen` displays an error.

Data Types: `double` | `char` | `string`

Output Arguments

`h` — Parity-check matrix for Hamming code

`m`-by-`n` matrix of binary values

Parity-check matrix for Hamming code, returned as an `m`-by-`n` matrix of binary values for the Hamming code.

Data Types: `single` | `double`

g — Generator matrix for Hamming code
k-by-n matrix of binary values

Generator matrix for Hamming code, returned as a k-by-n matrix of binary values corresponding to the parity-check matrix h.

Data Types: `single` | `double`

n — Codeword length of Hamming code
positive integer

Codeword length of Hamming code, returned as a positive integer. This value is calculated as 2^m–1.

Data Types: `single` | `double`

k — Message length of Hamming code
positive integer

Message length of Hamming code, returned as a positive integer. This value is calculated as n–m.

Data Types: `single` | `double`

Algorithms

`hammgen` uses the function `gftuple` to create the parity-check matrix by converting each element in the Galois field (GF) to its polynomial representation. Unlike `gftuple`, which performs computations in GF(2^m) and processes one m-tuple at a time, the `hammgen` function generates the entire sequence from 0 to 2^m–1. The computation algorithm uses all previously computed values to generate the computation result. If the value of m is less than 25 and the primitive polynomial is the default primitive polynomial for GF(2^m), the syntax `hammgen(m)` might be faster than the syntax `hammgen(m,poly)`.

See Also

Functions
`decode` | `encode` | `gen2par` | `gfprimdf` | `gftuple`

Topics
“Block Codes”

Introduced before R2006a

hank2sys

(To be removed) Convert Hankel matrix to linear system model

Compatibility

hank2sys will be removed in a future release.

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, `tol` is the order of the conversion. If `tol` < 1, `tol` is the tolerance in selecting the conversion order based on the singular values. If you omit `tol`, 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)
```

The output is

```
num =

         0    1.0000    0.0000    1.0000


den =

    1.0000    0.0000    0.0000    0.0000


sv =
    1.6180
    1.0000
    0.6180
```

Compatibility Considerations

hank2sys will be removed in a future release.
Not recommended starting in R2020b

hank2sys will be removed in a future release.

See Also

`hankel`

Introduced before R2006a

heldeintrlv

Restore ordering of symbols permuted using `helintrlv`

Syntax

```
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)
[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,init_state)
deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)
```

Description

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp)` restores the ordering of symbols in `data` by placing them in an array row by row and then selecting groups in a helical fashion to place in the output, `deintrlved`. `data` must have `col*ngrp` elements. If `data` is a matrix with multiple rows and columns, it must have `col*ngrp` rows, and the function processes the columns independently. `state` is a structure that holds the final state of the array. `state.value` stores input symbols that remain in the `col` columns of the array and do not appear in the output.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and `col` columns. The function initializes the top of the array with zeros. It then places `col*ngrp` symbols from the input into the next `ngrp` rows of the array. The function places symbols from the array in the output, `intrlved`, placing `ngrp` symbols at a time; the kth group of `ngrp` symbols comes from the kth column of the array, starting from row 1+(k-1)*`stp`. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

`[deintrlved,state] = heldeintrlv(data,col,ngrp,stp,init_state)` initializes the array with the symbols contained in `init_state.value` instead of zeros. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver. In this syntax, some output symbols are default values of 0, some are input symbols from `data`, and some are initialization values from `init_state.value`.

`deintrlved = heldeintrlv(data,col,ngrp,stp,init_state)` is the same as the syntax above, except that it does not record the deinterleaver's final state. This syntax is appropriate for the last in a series of calls to this function. However, if you plan to call this function again to continue the deinterleaving process, the syntax above is more appropriate.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `helintrlv` function, use the same `col`, `ngrp`, and `stp` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helintrlv` followed by `heldeintrlv` leaves data unchanged, after you take their combined delay of `col*ngrp*ceil(stp*(col-1)/ngrp)` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers”.

Note Because the delay is an integer multiple of the number of symbols in `data`, you must use `heldeintrlv` at least *twice* (possibly more times, depending on the actual delay value) before the function returns results that represent more than just the delay.

Examples

Recover interleaved data, taking into account the delay of the interleaver-deinterleaver pair.

```
col = 4; ngrp = 3; stp = 2; % Helical interleaver parameters
% Compute the delay of interleaver-deinterleaver pair.
delayval = col * ngrp * ceil(stp * (col-1)/ngrp);

len = col*ngrp; % Process this many symbols at one time.
data = randi([0 9],len,1); % Random symbols
data_padded = [data; zeros(delayval,1)]; % Pad with zeros.

% Interleave zero-padded data.
[i1,istate] = helintrlv(data_padded(1:len),col,ngrp,stp);
[i2,istate] = helintrlv(data_padded(len+1:2*len),col,ngrp, ...
              stp,istate);
i3 = helintrlv(data_padded(2*len+1:end),col,ngrp,stp,istate);

% Deinterleave.
[d1,dstate] = heldeintrlv(i1,col,ngrp,stp);
[d2,dstate] = heldeintrlv(i2,col,ngrp,stp,dstate);
d3 = heldeintrlv(i3,col,ngrp,stp,dstate);

% Check the results.
d0 = [d1; d2; d3]; % All the deinterleaved data
d0_trunc = d0(delayval+1:end); % Remove the delay.
ser = symerr(data,d0_trunc)
```

The output below shows that no symbol errors occurred.

```
ser =

     0
```

See Also

helintrlv

Topics

“Interleaving”

Introduced before R2006a

helintrlv

Permute symbols using helical array

Syntax

```
intrlved = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp)
[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state)
```

Description

intrlved = helintrlv(data,col,ngrp,stp) permutes the symbols in data by placing them in an unlimited-row array in helical fashion and then placing rows of the array in the output, intrlved. data must have col*ngrp elements. If data is a matrix with multiple rows and columns, it must have col*ngrp rows, and the function processes the columns independently.

The function uses the array internally for its computations. The array has unlimited rows indexed by 1, 2, 3,..., and col columns. The function partitions col*ngrp symbols from the input into consecutive groups of ngrp symbols. The function places the kth group in the array along column k, starting from row 1+(k-1)*stp. Positions in the array that do not contain input symbols have default values of 0. The function places col*ngrp symbols from the array in the output, intrlved, by reading the first ngrp rows sequentially. Some output symbols are default values of 0 rather than input symbols; similarly, some input symbols are left in the array and do not appear in the output.

[intrlved,state] = helintrlv(data,col,ngrp,stp) returns a structure that holds the final state of the array. state.value stores input symbols that remain in the col columns of the array and do not appear in the output.

[intrlved,state] = helintrlv(data,col,ngrp,stp,init_state) initializes the array with the symbols contained in init_state.value. The structure init_state is typically the state output from a previous call to this same function, and is unrelated to the corresponding deinterleaver. In this syntax, some output symbols are default values of 0, some are input symbols from data, and some are initialization values from init_state.value.

Examples

The example below rearranges the integers from 1 to 24.

```
% Interleave some symbols. Record final state of array.
[i1,state] = helintrlv([1:12]',3,4,1);
% Interleave more symbols, remembering the symbols that
% were left in the array from the earlier command.
i2 = helintrlv([13:24]',3,4,1,state);

disp('Interleaved data:')
disp([i1,i2]')
disp('Values left in array after first interleaving operation:')
state.value{:}
```

During the successive calls to helintrlv, it internally creates the three-column arrays

```
[1  0  0;
 2  5  0;
 3  6  9;
 4  7 10;
 0  8 11;
 0  0 12]
```

and

```
[13  8 11;
 14 17 12;
 15 18 21;
 16 19 22;
  0 20 23;
  0  0 24]
```

In the second array shown above, the 8, 11, and 12 are values left in the array from the previous call to the function. Specifying the `init_state` input in the second call to the function causes it to use those values rather than the default values of 0.

The output from this example is below. (The matrix has been transposed for display purposes.) The interleaved data comes from the top four rows of the three-column arrays shown above. Notice that some of the symbols in the first half of the interleaved data are default values of 0, some of the symbols in the second half of the interleaved data were left in the array from the first call to `helintrlv`, and some of the input symbols (20, 23, and 24) do not appear in the interleaved data at all.

```
Interleaved data:
  Columns 1 through 10

     1     0     0     2     5     0     3     6     9     4
    13     8    11    14    17    12    15    18    21    16

  Columns 11 through 12

     7    10
    19    22

Values left in array after first interleaving operation:

ans =

     []


ans =

     8


ans =

    11    12
```

The example on the reference page for `heldeintrlv` also uses this function.

See Also

`heldeintrlv`

Topics

"Interleaving"

Introduced before R2006a

helscandeintrlv

Restore ordering of symbols in helical pattern

Syntax

```
deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)
```

Description

`deintrlvd = helscandeintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements in a helical fashion and then sending the matrix contents to the output row by row. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function places input elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

To use this function as an inverse of the `helscanintrlv` function, use the same `Nrows`, `Ncols`, and `hstep` inputs in both functions. In that case, the two functions are inverses in the sense that applying `helscanintrlv` followed by `helscandeintrlv` leaves data unchanged.

Examples

Apply Helical Deinterleaving to Integer Row Vector

Apply helical scan deinterleaving to the vector [1:12], rearranging the vector using a 3-by-4 temporary matrix and diagonals of slope 1.

Internally, the `helscandeintrlv` function creates the 3-by-4 temporary matrix using length-four diagonals. As represented here.

```
[1 10  7  4;
 5  2 11  8;
 9  6  3 12]
```

```
ans = 3×4

     1    10     7     4
     5     2    11     8
     9     6     3    12
```

The function then sends the elements, row by row, to the output `d`.

```
d = helscandeintrlv(1:12,3,4,1)
```

```
d = 1×12

     1    10     7     4     5     2    11     8     9     6     3    12
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`helscanintrlv`

Topics

"Interleaving"

Introduced before R2006a

helscanintrlv

Reorder symbols in helical pattern

Syntax

```
intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)
```

Description

`intrlvd = helscanintrlv(data,Nrows,Ncols,hstep)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents to the output in a helical fashion. `Nrows` and `Ncols` are the dimensions of the temporary matrix. `hstep` is the slope of the diagonal, that is, the amount by which the row index increases as the column index increases by one. `hstep` must be a nonnegative integer less than `Nrows`.

Helical fashion means that the function selects elements along diagonals of the temporary matrix. The number of elements in each diagonal is exactly `Ncols`, after the function wraps past the edges of the matrix when necessary. The function traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples

The command below rearranges a vector using diagonals of two different slopes.

```
i1 = helscanintrlv(1:12,3,4,1) % Slope of diagonal is 1.
i2 = helscanintrlv(1:12,3,4,2) % Slope of diagonal is 2.
```

The output is below.

```
i1 =

  Columns 1 through 10

     1     6    11     4     5    10     3     8     9     2

  Columns 11 through 12

     7    12


i2 =

  Columns 1 through 10

     1    10     7     4     5     2    11     8     9     6

  Columns 11 through 12
```

```
     3     12
```

In each case, the function internally creates the temporary 3-by-4 matrix

```
[1  2  3  4;
 5  6  7  8;
 9 10 11 12]
```

To form i1, the function forms each slope-one diagonal by moving one row down and one column to the right. The first diagonal contains 1, 6, 11, and 4, while the second diagonal starts with 5 because that is beneath 1 in the temporary matrix.

To form i2, the function forms each slope-two diagonal by moving two rows down and one column to the right. The first diagonal contains 1, 10, 7, and 4, while the second diagonal starts with 5 because that is beneath 1 in the temporary matrix.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

helscandeintrlv

Topics

"Interleaving"

Introduced before R2006a

hex2poly

Convert hexadecimal character vector to binary coefficients

Syntax

```
b = hex2poly(hex)
b = hex2poly(hex,ord)
```

Description

`b = hex2poly(hex)` converts a hexadecimal character vector, `hex`, to a vector of binary coefficients, `b`.

`b = hex2poly(hex,ord)` specifies the power order, `ord`, of the coefficients that comprise the output. If omitted, `ord` is `'descending'`.

Examples

Convert Hexadecimal Polynomial to Binary Vector

Convert the hexadecimal polynomial `'1AF'` to a vector of binary coefficients. The coefficients represent the polynomial $x^8 + x^7 + x^5 + x^3 + x^2 + x + 1$.

```
b = hex2poly('1AF')
```

```
b = 1×9

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

Convert Hexadecimal into Ascending Order Binary Vector

Convert hexadecimal `'0x82608EDB'` to a vector of binary coefficients. Specify that the binary coefficients are in ascending order.

```
b = hex2poly('0x82608EDB','ascending')
```

```
b = 1×32

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

The binary representation corresponds to a polynomial of $x^{31} + x^{25} + x^{22} + x^{21} + x^{15} + x^{11} + x^{10} + x^9 + x^7 + x^6 + x^4 + x^3 + x + 1$.

Input Arguments

`hex` — Hexadecimal number

character vector

Hexadecimal number, specified as a character vector.

Example: `'FF'`

Example: `'0x3FA'`

Data Types: `char`

`ord` — Power order

`'descending'` (default) | `'ascending'`

Power order of the vector of binary coefficients, specified as a character vector having a value of `'ascending'` or `'descending'`.

Data Types: `char`

Output Arguments

`b` — Binary coefficients

vector

Binary coefficients representing a polynomial, returned as a row vector having length equal to $p + 1$, where p is the order of hexadecimal input.

See Also

`dec2hex` | `oct2poly`

Introduced in R2015b

hilbiir

(To be removed) Design Hilbert transform IIR filter

Compatibility

hilbiir will be removed in a future release. To design a Hilbert transform filter, use the `fdesign.hilbert` object.

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) = -jsgn(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

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

Choosing a Group Delay Parameter

The filter design is an approximation. If you provide the filter's group delay as an input argument, 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, 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 one-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 one-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” on page 2-497 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`), `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, the order of the filter is determined by

$$\frac{\text{truncated-singular-value}}{\text{maximum-singular-value}} < \text{tol}$$

If `tol` > 1, 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 “Syntaxes for Plots” on page 2-498.

`[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.

Examples

For an example using the function's default values, type one of the following commands at the MATLAB prompt.

```
hilbiir
[num,den] = hilbiir
```

Algorithms

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 [1].

Compatibility Considerations

`hilbiir` will be removed in a future release.

Not recommended starting in R2020b

hilbiir will be removed in a future release. To design Hilbert transform IIR filter, use the `fdesign.hilbert` object.

References

[1] Kailath, Thomas, *Linear Systems*, Englewood Cliffs, NJ, Prentice-Hall, 1980.

See Also

`grpdelay`

Introduced before R2006a

huffmandeco

Decode binary code by Huffman decoding

Syntax

```
sig = huffmandeco(code,dict)
```

Description

`sig = huffmandeco(code,dict)` decodes the numeric Huffman code vector, `code`, by using the Huffman codes described by input code dictionary `dict`. Input `dict` is an *N*-by-2 cell array, where *N* is the number of distinct possible symbols in the original signal that encodes `code`. The first column of `dict` represents the distinct symbols, and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` can be the prefix of any other codeword in `dict`. You can generate `dict` by using the `huffmandict` function and `code` by using the `huffmanenco` function. If all symbols in `dict` are numeric, output `sig` is a vector. If any symbol in `dict` is alphabetic, `sig` is a one-dimensional cell array.

Examples

Huffman Encoding and Decoding

Create unique symbols, and assign probabilities of occurrence to them.

```
symbols = 1:6;
p = [.5 .125 .125 .125 .0625 .0625];
```

Create a Huffman dictionary based on the symbols and their probabilities.

```
dict = huffmandict(symbols,p);
```

Generate a vector of random symbols.

```
inputSig = randsrc(100,1,[symbols;p]);
```

Encode the random symbols.

```
code = huffmanenco(inputSig,dict);
```

Decode the data. Verify that the decoded symbols match the original symbols.

```
sig = huffmandeco(code,dict);
isequal(inputSig,sig)
```

```
ans = logical
   1
```

Convert the original signal to a binary, and determine the length of the binary symbols.

```
binarySig = de2bi(inputSig);
seqLen = numel(binarySig)
seqLen = 300
```

Convert the Huffman-encoded symbols to binary, and determine the length of the encoded binary symbols.

```
binaryComp = de2bi(code);
encodedLen = numel(binaryComp)
```

```
encodedLen = 224
```

Huffman Encoding and Decoding with Alphanumeric Signal

Define the alphanumeric symbols in cell array form.

```
inputSig = {'a2',44,'a3',55,'a1'}
```

```
inputSig=1×5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

Define a Huffman dictionary. Codes for signal letters must be numeric.

```
dict = {'a1',0; 'a2',[1,0]; 'a3',[1,1,0]; 44,[1,1,1,0]; 55,[1,1,1,1]}
```

```
dict=5×2 cell array
    {'a1'}    {[       0]}
    {'a2'}    {1x2 double}
    {'a3'}    {1x3 double}
    {[44]}    {1x4 double}
    {[55]}    {1x4 double}
```

Encode the alphanumeric symbols.

```
enco = huffmanenco(inputSig,dict);
```

Decode the data. Verify that the decoded symbols match the original symbols.

```
sig = huffmandeco(enco,dict)
```

```
sig=1×5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

```
isequal(inputSig,sig)
```

```
ans = logical
   1
```

Input Arguments

code — Huffman code
numeric vector

Huffman code, specified as a numeric vector. This value must be a Huffman code encoded using a code dictionary produced by the `huffmandict` function.

Data Types: `double`

`dict` — Huffman code dictionary
N-by-2 cell array

Huffman code dictionary, specified as an *N*-by-2 cell array. *N* is the number of distinct possible symbols for the function to encode. The first column of `dict` represents the distinct symbols, and the second column represents the corresponding codewords. Each codeword is represented as a numeric row vector, and no codeword in `dict` can be the prefix of any other codeword in `dict`. You can generate `dict` by using the `huffmandict` function.

Data Types: `double` | `cell`

Output Arguments

`sig` — Decoded signal
numeric vector | numeric cell array | alphanumeric cell array

Decoded signal, returned as a numeric vector, numeric cell array, or alphanumeric cell array.

- If all symbols in input code dictionary `dict` are numeric, `sig` is a vector.
- If any symbol in input code dictionary `dict` is alphabetic, `sig` is a one-dimensional cell array.

References

[1] Sayood, Khalid. *Introduction to Data Compression*. 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

See Also

Functions
`huffmandict` | `huffmanenco`

Topics
“Huffman Coding”

Introduced before R2006a

huffmandict

Generate Huffman code dictionary for source with known probability model

Syntax

```
[dict,avglen] = huffmandict(symbols,prob)
[dict,avglen] = huffmandict(symbols,prob,N)
[dict,avglen] = huffmandict(symbols,prob,N,variance)
```

Description

`[dict,avglen] = huffmandict(symbols,prob)` generates a binary Huffman code dictionary, `dict`, for the source symbols, `symbols`, by using the maximum variance algorithm. The input `prob` specifies the probability of occurrence for each of the input symbols. The length of `prob` must equal the length of `symbols`. The function also returns average codeword length `avglen` of the dictionary, weighted according to the probabilities in the input `prob`.

`[dict,avglen] = huffmandict(symbols,prob,N)` generates an `N`-ary Huffman code dictionary using maximum variance algorithm. `N` must not exceed the number of source symbols.

`[dict,avglen] = huffmandict(symbols,prob,N,variance)` generates an `N`-ary Huffman code dictionary with the specified variance.

Examples

Generate Huffman Code and View Results

Generate a binary Huffman code dictionary, additionally returning the average code length.

Specify a symbol alphabet vector and a symbol probability vector.

```
symbols = (1:5); % Alphabet vector
prob = [.3 .3 .2 .1 .1]; % Symbol probability vector
```

Generate a binary Huffman code, displaying the average code length and the cell array containing the codeword dictionary.

```
[dict,avglen] = huffmandict(symbols,prob)
```

```
dict=5×2 cell array
    {[1]}    {1x2 double}
    {[2]}    {1x2 double}
    {[3]}    {1x2 double}
    {[4]}    {1x3 double}
    {[5]}    {1x3 double}


avglen = 2.2000
```

Display the fifth codeword from the dictionary.

```
samplecode = dict{5,2} % Codeword for fifth signal value
```

```
samplecode = 1×3

     1     1     0
```

Generate Ternary Huffman Codes

Use the code dictionary generator for Huffman coder function to generate binary and ternary Huffman codes.

Specify a symbol alphabet vector and a symbol probability vector.

```
symbols = (1:5); % Alphabet vector
prob = [.3 .3 .2 .1 .1]; % Symbol probability vector
```

Generate a binary Huffman code, displaying the cell array containing the codeword dictionary.

```
[dict,avglen] = huffmandict(symbols,prob);
dict(:,2) = cellfun(@num2str,dict(:,2),'UniformOutput',false)
```

```
dict=5×2 cell array
    {[1]}    {'0  1'   }
    {[2]}    {'0  0'   }
    {[3]}    {'1  0'   }
    {[4]}    {'1  1  1'}
    {[5]}    {'1  1  0'}
```

Generate a ternary Huffman code with minimum variance.

```
[dict,avglen] = huffmandict(symbols,prob,3,'min');
dict(:,2) = cellfun(@num2str,dict(:,2),'UniformOutput',false)
```

```
dict=5×2 cell array
    {[1]}    {'2'   }
    {[2]}    {'1'   }
    {[3]}    {'0  0'}
    {[4]}    {'0  2'}
    {[5]}    {'0  1'}
```

Input Arguments

`symbols` — Source symbols

vector | cell array | alphanumeric cell array

Source symbols, specified as a vector, cell array, or an alphanumeric cell array. `symbols` lists the distinct signal values that the source produces. If `symbols` is a cell array, it must be a 1-by-*S* or *S*-by-1 cell array, where *S* is the number of symbols.

Data Types: `double` | `cell`

`prob` — Probability of occurrence

vector in the range [0, 1]

Probability of occurrence for each symbol, specified as a vector in the range [0, 1]. The elements of this vector must sum to 1. The length this vector must equal the length of input `symbols`.

Data Types: `double`

`N` — N-ary Huffman code dictionary

scalar in the range [2, 10]

N-ary Huffman code dictionary, specified as a scalar in the range [2, 10]. This value must be less than or equal to the length of input `symbols`.

Data Types: `double`

`variance` — Variance for Huffman code

`'min'` | `'max'`

Variance for Huffman code, specified as one of these values.

- `'min'` — This function generates `N`-ary Huffman code dictionary with the minimum variance. If you do not specify the variance input argument, the function uses this case.
- `'max'` — This function generates `N`-ary Huffman code dictionary with the maximum variance.

Data Types: `char`

Output Arguments

`dict` — Huffman code dictionary

two-column cell array

Huffman code dictionary, returned as a two-column cell array. The first column lists the distinct signal values from input `symbols`. The second column corresponds to Huffman codewords, where each Huffman codeword is represented as a row vector. If you specify the input argument `N`, the function returns dict as an `N`-ary Huffman code dictionary.

Data Types: `double` | `cell`

`avglen` — Average codeword length

positive scalar

Average codeword length, weighted according to the probabilities in the input `prob`, returned as a positive scalar.

Data Types: `double`

References

[1] Sayood, Khalid. *Introduction to Data Compression*. 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

See Also

Functions
`huffmandeco` | `huffmanenco`

Topics
“Huffman Coding”

Introduced before R2006a

huffmanenco

Encode sequence of symbols by Huffman encoding

Syntax

```
code = huffmanenco(sig,dict)
```

Description

`code = huffmanenco(sig,dict)` encodes input signal `sig` using the Huffman codes described by input code dictionary `dict`. `sig` can have the form of a vector, cell array, or alphanumeric cell array. If `sig` is a cell array, it must be either a row or a column. `dict` is an N-by-2 cell array, where *N* is the number of distinct possible symbols to encode. The first column of `dict` represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a row vector, and no codeword in `dict` can be the prefix of any other codeword in `dict`. You can generate `dict` using the `huffmandict` function.

Examples

Huffman Encoding and Decoding

Create unique symbols, and assign probabilities of occurrence to them.

```
symbols = 1:6;
p = [.5 .125 .125 .125 .0625 .0625];
```

Create a Huffman dictionary based on the symbols and their probabilities.

```
dict = huffmandict(symbols,p);
```

Generate a vector of random symbols.

```
inputSig = randsrc(100,1,[symbols;p]);
```

Encode the random symbols.

```
code = huffmanenco(inputSig,dict);
```

Decode the data. Verify that the decoded symbols match the original symbols.

```
sig = huffmandeco(code,dict);
isequal(inputSig,sig)
```

```
ans = logical
   1
```

Convert the original signal to a binary, and determine the length of the binary symbols.

```
binarySig = de2bi(inputSig);
seqLen = numel(binarySig)
seqLen = 300
```

Convert the Huffman-encoded symbols to binary, and determine the length of the encoded binary symbols.

```
binaryComp = de2bi(code);
encodedLen = numel(binaryComp)
```

```
encodedLen = 224
```

Huffman Encoding and Decoding with Alphanumeric Signal

Define the alphanumeric symbols in cell array form.

```
inputSig = {'a2',44,'a3',55,'a1'}
```

```
inputSig=1×5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

Define a Huffman dictionary. Codes for signal letters must be numeric.

```
dict = {'a1',0; 'a2',[1,0]; 'a3',[1,1,0]; 44,[1,1,1,0]; 55,[1,1,1,1]}
```

```
dict=5×2 cell array
    {'a1'}    {[       0]}
    {'a2'}    {1x2 double}
    {'a3'}    {1x3 double}
    {[44]}    {1x4 double}
    {[55]}    {1x4 double}
```

Encode the alphanumeric symbols.

```
enco = huffmanenco(inputSig,dict);
```

Decode the data. Verify that the decoded symbols match the original symbols.

```
sig = huffmandeco(enco,dict)
```

```
sig=1×5 cell array
    {'a2'}    {[44]}    {'a3'}    {[55]}    {'a1'}
```

```
isequal(inputSig,sig)
```

```
ans = logical
   1
```

Input Arguments

sig — Input signal
vector | cell array | alphanumeric cell array

Input signal for the compression, specified as a vector, cell array, or an alphanumeric cell array. `sig` can have the form of a vector, cell array, or alphanumeric cell array. If `sig` is a cell array, it must be a 1-by-*S* or *S*-by-1 cell array, where *S* is the number of symbols.

Data Types: `double` | `cell`

`dict` — Huffman code dictionary
N-by-2 cell array

Huffman code dictionary, specified as an *N*-by-2 cell array. *N* is the number of distinct possible symbols for the function to encode. The first column of dict represents the distinct symbols and the second column represents the corresponding codewords. Each codeword is represented as a row vector, and no codeword in dict can be the prefix of any other codeword in dict. You can generate dict by using the `huffmandict` function.

Data Types: `double` | `cell`

Output Arguments

`code` — Encoded signal
vector

Encoded signal for the input Huffman code dictionary `dict`, returned as a vector.

References

[1] Sayood, Khalid. *Introduction to Data Compression*. 2nd ed. San Francisco: Morgan Kaufmann Publishers, 2000.

See Also

Functions
`huffmandeco` | `huffmandict`

Topics
“Huffman Coding”

Introduced before R2006a

ifft

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), 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.

Algorithms

If `x` is a column vector, `ifft` applies `dftmtx` to the multiplicative inverse of the primitive element of the Galois field and multiplies the resulting matrix by `x`.

See Also

`dftmtx` | `fft` | `gf`

Topics

"Signal Processing Operations in Galois Fields"

Introduced before R2006a

intdump

Integrate and dump

Syntax

```
y = intdump(x,nsamp)
```

Description

`y = intdump(x,nsamp)` integrates the signal `x` for one symbol period, then outputs the averaged one value into `Y`. `nsamp` is the number of samples per symbol. For two-dimensional signals, the function treats each column as one channel.

Examples

An example in “Combine Pulse Shaping and Filtering with Modulation” uses this function in conjunction with modulation.

Processes two independent channels, each of which contain three symbols of data made up of four samples.

```
s = rng;
rng(68521);
nsamp = 4; % Number of samples per symbol
ch1 = randi([0 1],3*nsamp,1); % Random binary channel
ch2 = rectpulse([1 2 3]',nsamp); % Rectangular pulses
x = [ch1 ch2]; % Two-channel signal
y = intdump(x,nsamp)
rng(s);
```

The output is below. Each column corresponds to one channel, and each row corresponds to one symbol.

```
y =

    0.5000    1.0000
    0.5000    2.0000
    1.0000    3.0000
```

See Also

`rectpulse`

Introduced before R2006a

intrlv

Reorder sequence of symbols

Syntax

```
intrlvd = intrlv(data,elements)
```

Description

`intrlvd = intrlv(data,elements)` rearranges the elements of `data` without repeating or omitting any elements. If `data` is a length-N vector or an N-row matrix, `elements` is a length-N vector that permutes the integers from 1 to N. The sequence in `elements` is the sequence in which elements from `data` or its columns appear in `intrlvd`. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

Examples

Apply Interleaving to Reorder Input Vector

Use the `intrlv` function to rearrange the elements of a vector to a random permutation determined by the `randperm` function.

```
p = randperm(10); % Permutation vector
a = intrlv(10:10:100,p)
```

```
a = 1×10

    60    30    70    80    50    10    20    40    90   100
```

The command below rearranges each of two columns of a matrix.

```
b = intrlv([.1 .2 .3 .4 .5; .2 .4 .6 .8 1]',[2 4 3 5 1])
```

```
b = 5×2

    0.2000    0.4000
    0.4000    0.8000
    0.3000    0.6000
    0.5000    1.0000
    0.1000    0.2000
```

Apply Interleaving to Reorder Input Matrix

Use the `intrlv` function to rearrange the elements of a matrix to a specified order.

Each column of the matrix applies the same reordering.

```
b = intrlv([.1 .2 .3 .4 .5; .2 .4 .6 .8 1]',[2 4 3 5 1])
```

```
b = 5×2

    0.2000    0.4000
    0.4000    0.8000
    0.3000    0.6000
    0.5000    1.0000
    0.1000    0.2000
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`deintrlv`

Topics

"Interleaving"

Introduced before R2006a

iqcoef2imbal

Convert compensator coefficient to amplitude and phase imbalance

Syntax

```
[A,P] = iqcoef2imbal(C)
```

Description

`[A,P] = iqcoef2imbal(C)` converts compensator coefficient `C` to its equivalent amplitude and phase imbalance.

Examples

Estimate I/Q Imbalance from Compensator Coefficient

Use `iqcoef2imbal` to estimate the amplitude and phase imbalance for a given complex coefficient. The coefficients are an output from the `step` function of the `IQImbalanceCompensator`.

Create a raised cosine transmit filter to generate a 64-QAM signal.

```
M = 64;
txFilt = comm.RaisedCosineTransmitFilter;
```

Modulate and filter random 64-ary symbols.

```
data = randi([0 M-1],100000,1);
dataMod = qammod(data,M);
txSig = step(txFilt,dataMod);
```

Specify amplitude and phase imbalance.

```
ampImb = 2; % dB
phImb = 15; % degrees
```

Apply the specified I/Q imbalance.

```
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Normalize the power of the received signal

```
rxSig = rxSig/std(rxSig);
```

Remove the I/Q imbalance using the `comm.IQImbalanceCompensator` System object. Set the compensator object such that the complex coefficients are made available as an output argument.

```
hIQComp = comm.IQImbalanceCompensator('CoefficientOutputPort',true);
[compSig,coef] = step(hIQComp,rxSig);
```

Estimate the imbalance from the last value of the compensator coefficient.

```
[ampImbEst,phImbEst] = iqcoef2imbal(coef(end));
```

Compare the estimated imbalance values with the specified ones. Notice that there is good agreement.

```
[ampImb phImb; ampImbEst phImbEst]
```

```
ans = 2×2

    2.0000   15.0000
    2.0178   14.5740
```

Input Arguments

C — Compensator coefficient

complex-valued scalar or vector

Coefficient used to compensate for an I/Q imbalance, specified as a complex-valued vector.

Example: `0.4+0.6i`

Example: `[0.1+0.2i; 0.3+0.5i]`

Data Types: `single` | `double`

Output Arguments

A — Amplitude imbalance

real-valued vector

Amplitude imbalance in dB, returned as a real-valued vector with the same dimensions as `C`.

P — Phase imbalance

real-valued vector

Phase imbalance in degrees, returned as a real-valued vector with the same dimensions as `C`.

More About

I/Q Imbalance Compensation

The function `iqcoef2imbal` is a supporting function for the `comm.IQImbalanceCompensator` System object.

Given a scaling and rotation factor, *G*, compensator coefficient, *C*, and received signal, *x*, the compensated signal, *y*, has the form

$$y = G[x + C\mathrm{conj}(x)]\ .$$

In matrix form, this can be rewritten as

$$\mathbf{Y} = \mathbf{R}\mathbf{X}\,,$$

where **X** is a 2-by-1 vector representing the imbalanced signal [X_I, X_Q] and **Y** is a 2-by-1 vector representing the compensator output [Y_I, Y_Q].

The matrix **R** is expressed as

$$\mathbf{R} = \begin{bmatrix} 1+\mathrm{Re}\{C\} & \mathrm{Im}\{C\} \\ \mathrm{Im}\{C\} & 1-\mathrm{Re}\{C\} \end{bmatrix}$$

For the compensator to perfectly remove the I/Q imbalance, **R** = **K**$^{-1}$ because **X = K S**, where **K** is a 2-by-2 matrix whose values are determined by the amplitude and phase imbalance and **S** is the ideal signal. Define a matrix **M** with the form

$$\mathbf{M} = \begin{bmatrix} 1 & -\alpha \\ \alpha & 1 \end{bmatrix}$$

Both **M** and **M**$^{-1}$ can be thought of as scaling and rotation matrices that correspond to the factor *G*. Because **K** = **R**$^{-1}$, the product **M**$^{-1}$ **R K M** is the identity matrix, where **M**$^{-1}$ **R** represents the compensator output and **K M** represents the I/Q imbalance. The coefficient α is chosen such that

$$\mathbf{KM} = L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\cos(\theta_Q) \\ I_{gain}\sin(\theta_I) & Q_{gain}\sin(\theta_Q) \end{bmatrix}$$

where *L* is a constant. From this form, we can obtain I_{gain}, Q_{gain}, θ_I, and θ_Q. For a given phase imbalance, Φ_{Imb}, the in-phase and quadrature angles can be expressed as

$$\theta_I = -(\pi/2)(\Phi_{Imb}/180)$$
$$\theta_Q = \pi/2 + (\pi/2)(\Phi_{Imb}/180)$$

Hence, $\cos(\theta_Q) = \sin(\theta_I)$ and $\sin(\theta_Q) = \cos(\theta_I)$ so that

$$L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\cos(\theta_Q) \\ I_{gain}\sin(\theta_I) & Q_{gain}\sin(\theta_Q) \end{bmatrix} = L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\sin(\theta_I) \\ I_{gain}\sin(\theta_I) & Q_{gain}\cos(\theta_I) \end{bmatrix}$$

The I/Q imbalance can be expressed as

$$\mathbf{KM} = \begin{bmatrix} K_{11}+\alpha K_{12} & -\alpha K_{11}+K_{12} \\ K_{21}+\alpha K_{22} & -\alpha K_{21}+K_{22} \end{bmatrix}$$
$$= L\begin{bmatrix} I_{gain}\cos(\theta_I) & Q_{gain}\sin(\theta_I) \\ I_{gain}\sin(\theta_I) & Q_{gain}\cos(\theta_I) \end{bmatrix}$$

Therefore,

$$(K_{21}+\alpha K_{22})/(K_{11}+\alpha K_{12}) = (-\alpha K_{11}+K_{12})/(-\alpha K_{21}+K_{22}) = \sin(\theta_I)/\cos(\theta_I)$$

The equation can be written as a quadratic equation to solve for the variable α, that is $D_1\alpha^2 + D_2\alpha + D_3 = 0$, where

$$D_1 = -K_{11}K_{12} + K_{22}K_{21}$$
$$D_2 = K_{12}^2 + K_{21}^2 - K_{11}^2 - K_{22}^2$$
$$D_3 = K_{11}K_{12} - K_{21}K_{22}$$

When $|C| \leq 1$, the quadratic equation has the following solution:

$$\alpha = \frac{-D_2 - \sqrt{D^2 - 4D_1D_3}}{2D_1}$$

Otherwise, when $|C| > 1$, the solution has the following form:

$$\alpha = \frac{-D_2 + \sqrt{D^2 - 4D_1D_3}}{2D_1}$$

Finally, the amplitude imbalance, A_{Imb}, and the phase imbalance, Φ_{Imb}, are obtained.

$$\mathbf{K}' = \mathbf{K}\begin{bmatrix} 1 & -\alpha \\ \alpha & 1 \end{bmatrix}$$

$$A_{Imb} = 20\log_{10}(K'_{11}/K'_{22})$$

$$\Phi_{Imb} = -2\tan^{-1}(K'_{21}/K'_{11})(180/\pi)$$

Note

- If C is real and $|C| \leq 1$, the phase imbalance is 0 and the amplitude imbalance is $20\log_{10}((1–C)/(1+C))$
- If C is real and $|C| > 1$, the phase imbalance is 180° and the amplitude imbalance is $20\log_{10}((C+1)/(C-1))$.
- If C is imaginary, $A_{Imb} = 0$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`iqimbal` | `iqimbal2coef`

Objects

`comm.IQImbalanceCompensator`

Introduced in R2014b

iqimbal2coef

Convert I/Q imbalance to compensator coefficient

Syntax

```
C = iqimbal2coef(A,P)
```

Description

`C = iqimbal2coef(A,P)` converts an I/Q amplitude and phase imbalance to its equivalent compensator coefficient.

Examples

Generate Coefficients for I/Q Imbalance Compensation

Generate coefficients for the I/Q imbalance compensator System object™ using `iqimbal2coef`. The compensator corrects for an I/Q imbalance using the generated coefficients.

Create a raised cosine transmit filter System object.

```
txRCosFilt = comm.RaisedCosineTransmitFilter;
```

Modulate and filter random 64-ary symbols.

```
M= 64;
data = randi([0 M-1],100000,1);
dataMod = qammod(data,M);
txSig = txRCosFilt(dataMod);
```

Specify amplitude and phase imbalance.

```
ampImb = 2; % dB
phImb = 15; % degrees
```

Apply the specified I/Q imbalance.

```
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Normalize the power of the received signal.

```
rxSig = rxSig/std(rxSig);
```

Remove the I/Q imbalance by creating and applying a `comm.IQImbalanceCompensator` object. Set the compensator such that the complex coefficients are made available as an output argument.

```
iqComp = comm.IQImbalanceCompensator('CoefficientOutputPort',true);
[compSig,coef] = iqComp(rxSig);
```

Compare the final compensator coefficient to the coefficient generated by the `iqimbal2coef` function. Observe that there is good agreement.

```
idealcoef = iqimbal2coef(ampImb,phImb);
[coef(end); idealcoef]
```

```
ans = 2×1 complex

  -0.1137 + 0.1296i
  -0.1126 + 0.1334i
```

Input Arguments

`A` — Amplitude imbalance

real-valued scalar or vector

Amplitude imbalance in dB, specified as a real-valued row or column vector.

Example: `3`

Example: `[0; 5]`

Data Types: `double`

`P` — Phase imbalance

real-valued scalar or vector

Phase imbalance in degrees, specified as a real-valued row or column vector.

Example: `10`

Example: `[15; 45]`

Data Types: `double`

Output Arguments

`C` — Compensator coefficient

complex-valued vector

Coefficient that perfectly compensates for the I/Q imbalance, returned as a complex-valued vector having the same dimensions as `A` and `P`.

More About

I/Q Imbalance Compensation

The function `iqimbal2coef` is a supporting function for the `comm.IQImbalanceCompensator` System object.

Define **S** and **X** as 2-by-1 vectors representing the I and Q components of the ideal and I/Q imbalanced signals, respectively.

$$\mathbf{X} = \mathbf{K} \cdot \mathbf{S}$$

where **K** is a 2-by-2 matrix whose values are determined by the amplitude imbalance, *A*, and phase imbalance, *P*. *A* is expressed in dB and *P* is expressed in degrees.

The imbalance can be expressed as:

$$I_{gain} = 10^{0.5A/20}$$

$$Q_{gain} = 10^{-0.5A/20}$$

$$\theta_i = -\left(\frac{P}{2}\right)\left(\frac{\pi}{180}\right)$$

$$\theta_q = \frac{\pi}{2} + \left(\frac{P}{2}\right)\left(\frac{\pi}{180}\right)$$

Then **K** has the form:

$$\mathbf{K} = \begin{bmatrix} I_{gain}\cos(\theta_i) & Q_{gain}\cos(\theta_q) \\ I_{gain}\sin(\theta_i) & Q_{gain}\sin(\theta_q) \end{bmatrix}$$

The vector **Y** is defined as the I/Q imbalance compensator output.

$$\mathbf{Y} = \mathbf{R} \cdot \mathbf{X}$$

For the compensator to perfectly remove the I/Q imbalance, **R** must be the matrix inversion of **K**, namely:

$$\mathbf{R} = \mathbf{K}^{-1}$$

Using complex notation, the vector **Y** can be rewritten as:

$$\begin{aligned} y &= w_1 x + w_2 \text{conj}(x) \\ &= w_1\left(x + \left({}^{w_2}\!/\!{}_{w_1}\right)\text{conj}(x)\right) \end{aligned}$$

where,

$$\text{Re}\{w_1\} = (R_{11} + R_{22})/2$$

$$\text{Im}\{w_1\} = (R_{21} - R_{12})/2$$

$$\text{Re}\{w_2\} = (R_{11} - R_{22})/2$$

$$\text{Im}\{w_2\} = (R_{21} + R_{12})/2$$

The output of the function is w_2/w_1. To exactly obtain the original signal, the compensator output needs to be scaled and rotated by the complex number w_1.

Note There are cases for which the output of `iqimbal2coef` is unreliable.

- If the phase imbalance is ±90°, the in-phase and quadrature components will become co-linear; consequently, the I/Q imbalance cannot be compensated.
- If the amplitude imbalance is 0 dB and the phase imbalance is 180°, $w_1 = 0$ and $w_2 = 1i$; therefore, the compensator takes the form of $y = 1i{*}\text{conj}(x)$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`iqcoef2imbal` | `iqimbal`

Objects

`comm.IQImbalanceCompensator`

Introduced in R2014b

iqimbal

Apply I/Q imbalance to input signal

Syntax

```
y = iqimbal(x,A)
y = iqimbal(x,A,P)
```

Description

`y = iqimbal(x,A)` applies I/Q amplitude imbalance `A` to input signal `x`.

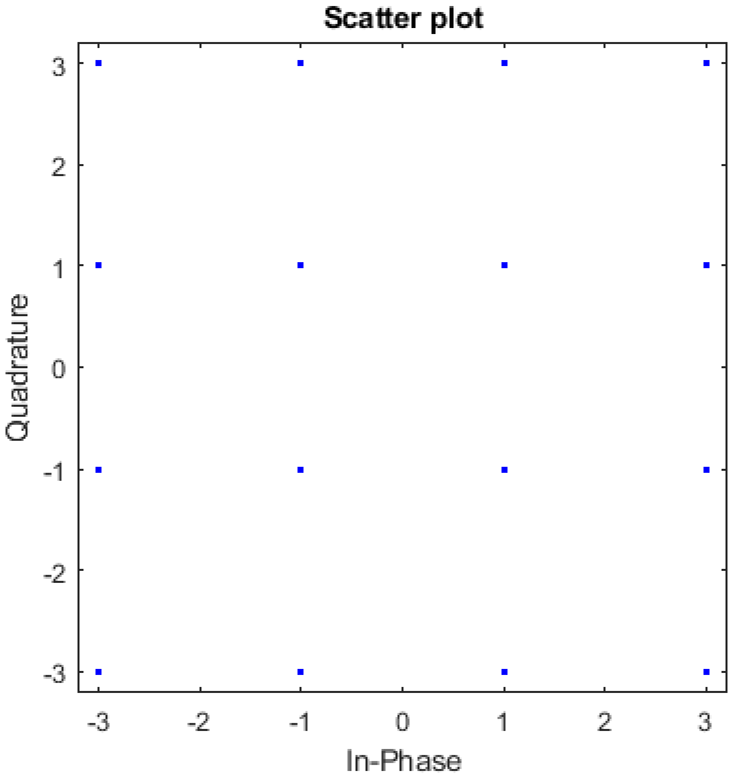
`y = iqimbal(x,A,P)` applies I/Q amplitude imbalance `A` and phase imbalance `P` to input signal `x`.

Examples

Apply Amplitude Imbalance to 16-QAM

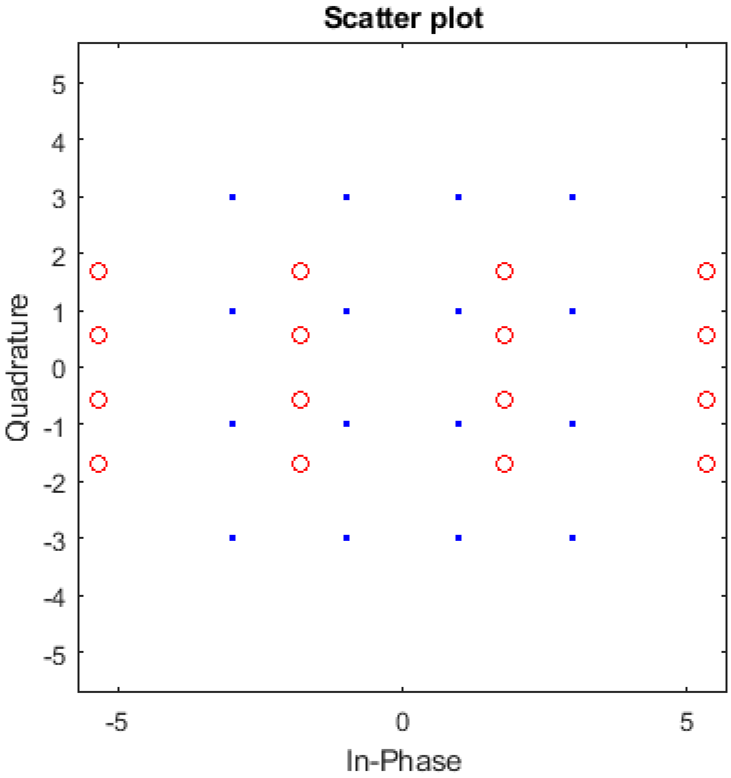
Generate a 16-QAM signal. Display the scatter plot.

```
x = qammod(randi([0 15],1000,1),16);
h = scatterplot(x);
hold on
```



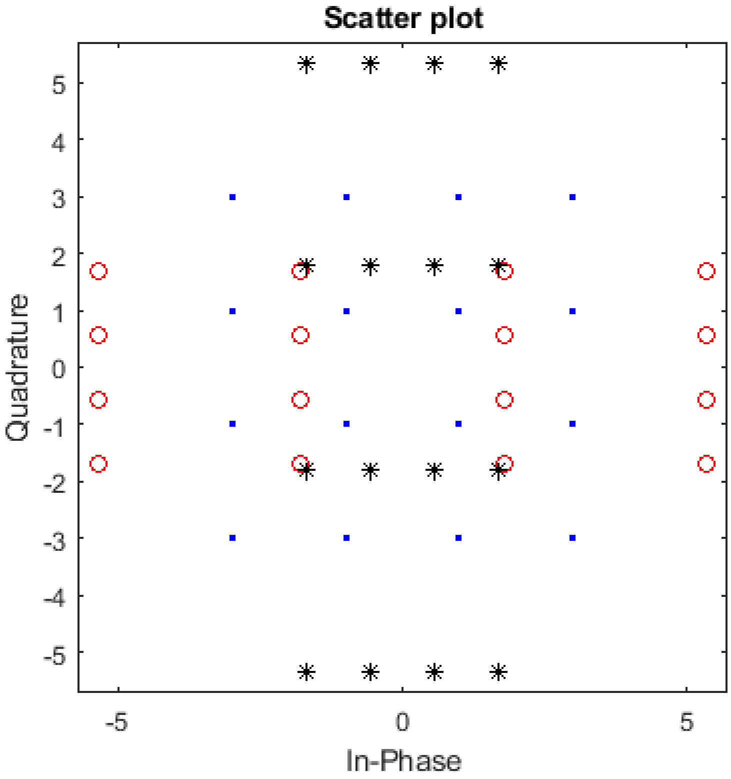
Apply a 10 dB amplitude imbalance. A positive amplitude imbalance causes horizontal stretching of the constellation.

```
y = iqimbal(x,10);
scatterplot(y,1,0,'ro',h)
```



Apply a -10 dB amplitude imbalance. A negative amplitude imbalance causes vertical stretching of the constellation.

```
z = iqimbal(x,-10);
scatterplot(z,1,0,'k*',h)
hold off
```



Apply Phase and Amplitude Imbalance to 16-QAM Signal

Generate a 16-QAM signal having two channels.

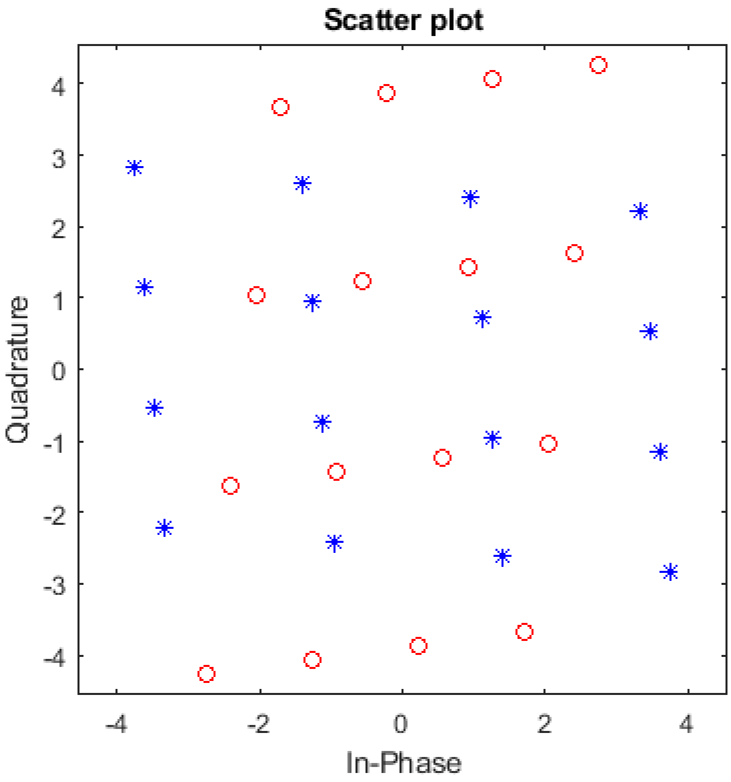
```
x = qammod(randi([0 15],1000,2),16);
```

Apply a 3 dB amplitude imbalance and a 10 degree phase imbalance to the first channel. Apply a –5 dB amplitude imbalance and a –15 degree phase imbalance to the second channel.

```
y = iqimbal(x,[3 -5],[10 -15]);
```

Plot the constellation diagram of both channels of the impaired signal.

```
h = scatterplot(y(:,1),1,0,'b*');
hold on
scatterplot(y(:,2),1,0,'ro',h)
hold off
```



The first channel is stretched horizontally, and the second channel is stretched vertically.

Apply I/Q Imbalance and DC Offset to QPSK

Apply a 1 dB, 5 degree I/Q imbalance to a QPSK signal. Then apply a DC offset. Visualize the offset using a spectrum analyzer.

Generate a QPSK sequence.

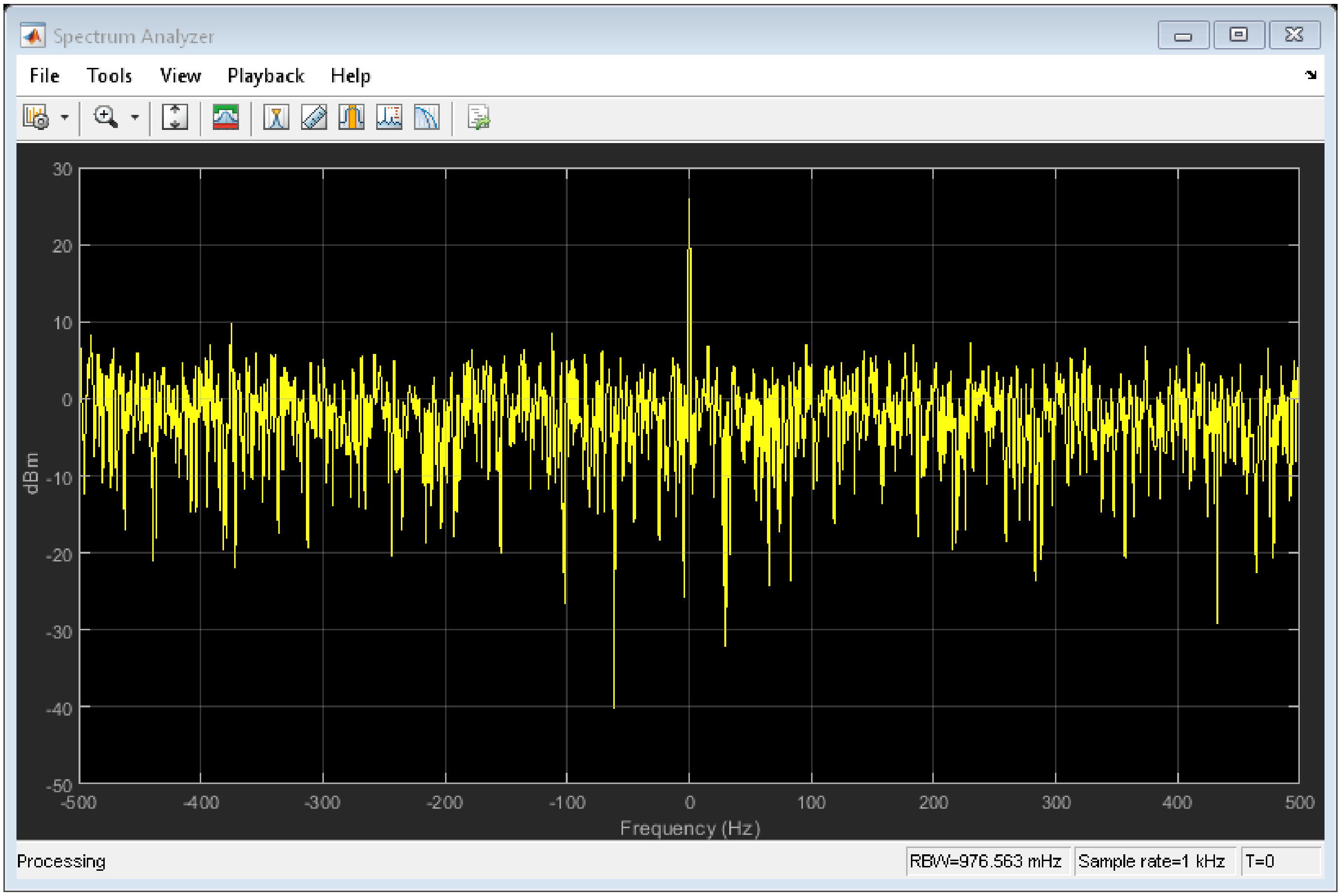
```
x = pskmod(randi([0 3],1e4,1),4,pi/4);
```

Apply a 1 dB amplitude imbalance and 5 degree phase imbalance to a QPSK signal. Apply a 0.5 + 0.3i DC offset.

```
y = iqimbal(x,1,5);
z = y + complex(0.5,0.3);
```

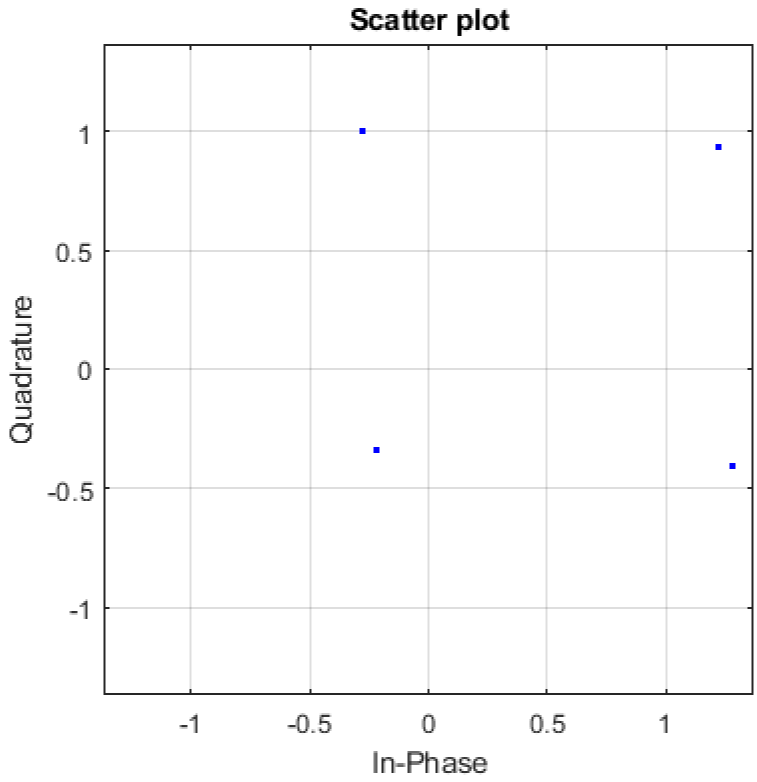
Plot the spectrum of the impaired signal.

```
sa = dsp.SpectrumAnalyzer('SampleRate',1000,'YLimits',[-50 30]);
sa(z)
```



Display the corresponding scatter plot.

```
scatterplot(z)
grid
```



The effect of the I/Q imbalance and the DC offset is observable.

Correct I/Q Imbalance on Noisy 8-PSK Signal

Generate random data and apply 8-PSK modulation.

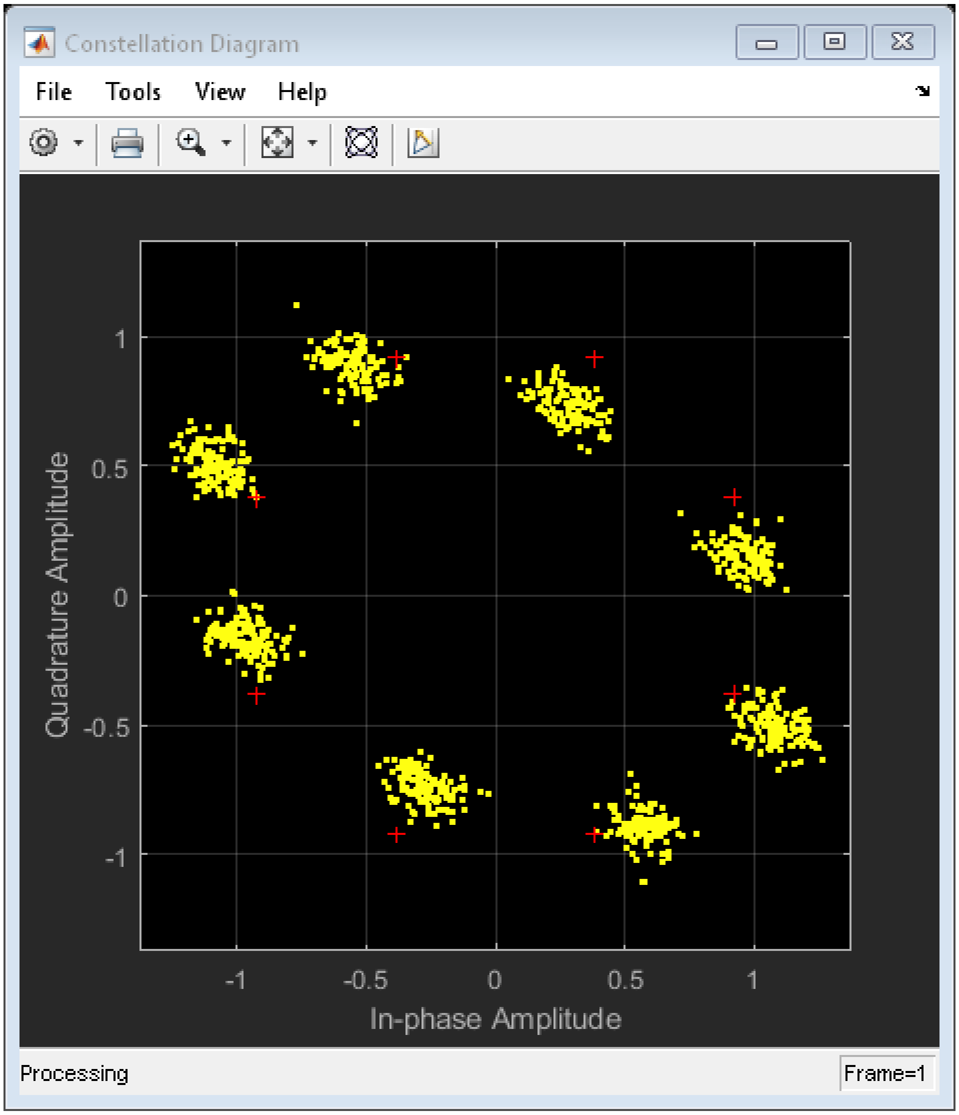
```
data = randi([0 7],2000,1);
txSig = pskmod(data,8,pi/8);
```

Pass the transmitted signal through an AWGN channel. Apply an I/Q imbalance.

```
noisySig = awgn(txSig,20);
rxSig = iqimbal(noisySig,2,20);
```

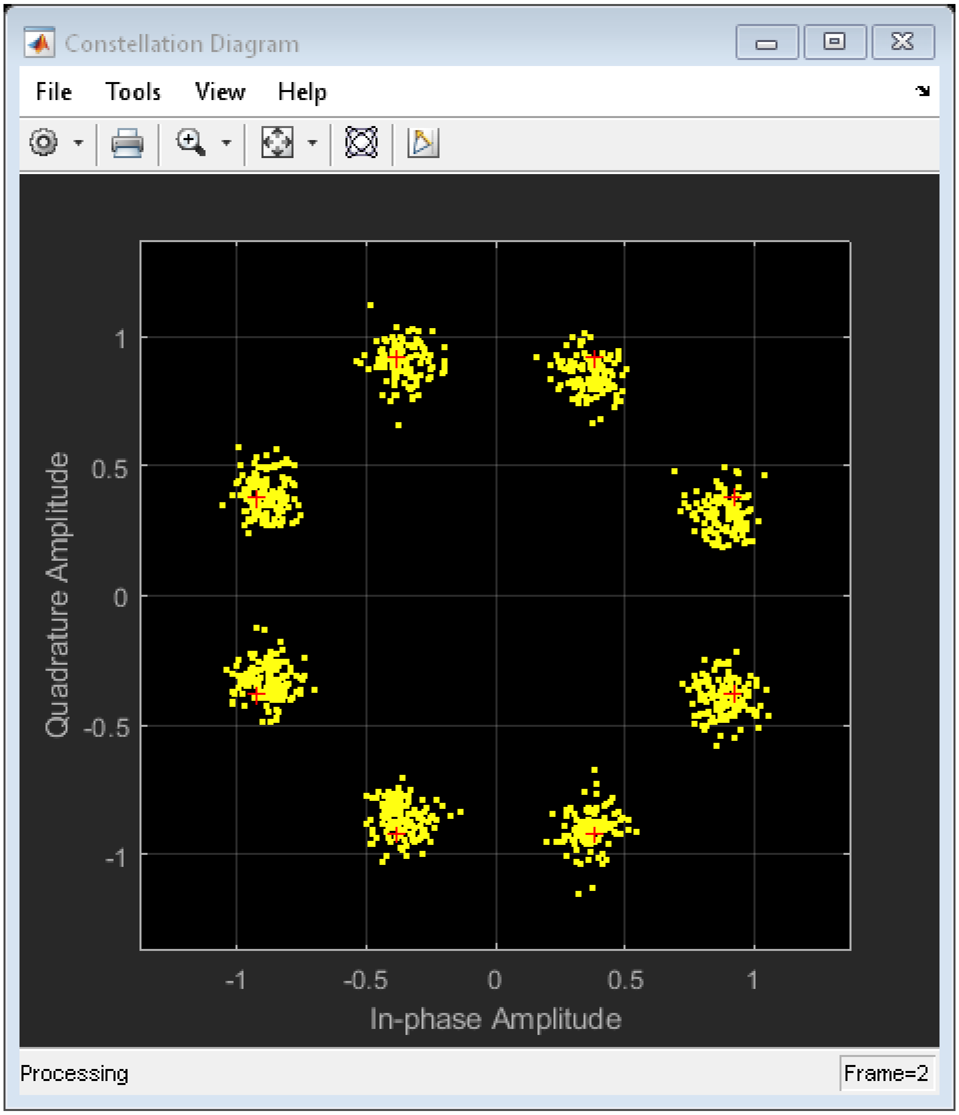
Create a constellation diagram object that displays only the last 1000 symbols. Plot the constellation diagram of the impaired signal.

```
cd = comm.ConstellationDiagram('ReferenceConstellation',pskmod(0:7,8,pi/8), ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',1000);
cd(rxSig)
```



Correct for the I/Q imbalance by using a `comm.IQImbalanceCompensator` object. Plot the constellation diagram of the signal after compensation.

```
iqComp = comm.IQImbalanceCompensator('StepSize',1e-3);
compSig = iqComp(rxSig);

cd(compSig)
```



The compensator removes the I/Q imbalance.

Input Arguments

x — Input signal

column vector | matrix

Input signal, specified as a column vector or matrix. The function supports multichannel operations, where the number of columns corresponds to the number of channels.

Example: `pskmod(randi([0 3],100,1),4,pi/4)`

Data Types: `single` | `double`

A — Amplitude imbalance

real scalar | row vector

Amplitude imbalance in dB, specified as a real scalar or row vector.

- If A is a scalar, the function applies the same amplitude imbalance to each channel.
- If A is a vector, then each element specifies the amplitude imbalance that is applied to the corresponding column (channel) of the input signal. The number of elements in A must equal the number of columns in x.

Example: 3

Example: [0 5]

Data Types: single | double

P — Phase imbalance
0 (default) | real scalar | row vector

Phase imbalance in degrees, specified as a real scalar or row vector.

- If P is omitted, a phase imbalance of zero degrees is used.
- If P is a scalar, the function applies the same phase imbalance to each channel.
- If P is a vector, then each element specifies the phase imbalance that is applied to the corresponding column (channel) of the input signal. The number of elements in P must equal the number of columns in x.

Example: 10

Example: [2.5 7]

Data Types: single | double

Output Arguments

y — Output signal
vector | matrix

Output signal, returned as a vector or matrix having the same dimensions as x. The number of columns in y corresponds to the number of channels.

Data Types: single | double

Algorithms

The iqimbal function applies an I/Q amplitude and phase imbalance to an input signal.

Given amplitude imbalance I_a in dB, the gain, g, resulting from the imbalance is defined as

$$g \triangleq g_r + ig_i = \left[10^{0.5\frac{I_a}{20}}\right] + i\left[10^{-0.5\frac{I_a}{20}}\right].$$

Applying the I/Q imbalance to input signal x results in output signal y such that

$$y = \mathrm{Re}(x) \cdot g_r e^{-i0.5 I_p(\pi/180)} + i\mathrm{Im}(x) \cdot g_i e^{i0.5 I_p(\pi/180)},$$

where g is the imbalance gain and I_p is the phase imbalance in degrees.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

I/Q Imbalance | `comm.IQImbalanceCompensator` | `iqcoef2imbal` | `iqimbal2coef`

Introduced in R2016b

iscatastrophic

True for trellis corresponding to catastrophic convolutional code

Syntax

```
iscatastrophic(s)
```

Description

`iscatastrophic(s)` returns `true` if the trellis `s` corresponds to a convolutional code that causes catastrophic error propagation. Otherwise, it returns `false`.

Examples

Determine if a Convolutional Code is Catastrophic

Determine if a convolutional code causes catastrophic error propagation.

Create the trellis for the standard, rate 1/2, constraint length 7 convolutional code.

```
t = poly2trellis(7,[171 133]);
```

Verify that the code is not catastrophic.

```
iscatastrophic(t)
```

```
ans = logical
   0
```

Create a trellis for a different convolutional code using the `poly2trellis` function.

```
u = poly2trellis(7,[161 143]);
```

Verify that the code is catastrophic.

```
iscatastrophic(u)
```

```
ans = logical
   1
```

References

[1] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall, 1995, pp. 274-275.

See Also

`convenc` | `istrellis` | `poly2trellis` | `struct`

Topics

"Convolutional Codes"

Introduced before R2006a

isprimitive

True for primitive polynomial for 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

The example below finds all primitive polynomials for GF(8) and then checks using `isprimitive` whether specific polynomials are primitive.

```
a = primpoly(3,'all','nodisplay'); % All primitive polys for GF(8)

isp1 = isprimitive(13) % 13 represents a primitive polynomial.

isp2 = isprimitive(14) % 14 represents a nonprimitive polynomial.
```

The output is below. If you examine the vector `a`, notice that `isp1` is true because `13` is an element in `a`, while `isp2` is false because `14` is not an element in `a`.

```
isp1 =

     1


isp2 =

     0
```

See Also

`gf` | `primpoly`

Topics

"Galois Field Computations"

Introduced before R2006a

istrellis

True for 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, `isok` is 1 and `status` is an empty character vector. Otherwise, `isok` is 0 and `status` 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, the shift register that receives the first input stream in the encoder provides the least significant bits in the state number, and 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)
```

The output is below.

```
isok =

     1


status =

     ''
```

Another example of a trellis is in “Trellis Description of a Convolutional Code”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`convenc` | `poly2trellis` | `struct` | `vitdec`

Topics

“Convolutional Codes”

Introduced before R2006a

legacychannelsim

(Removed) Toggles random number generation mode for channel objects

Note `legacychannelsim` has been removed. Use `comm.RayleighChannel` or `comm.RicianChannel` instead.

Syntax

```
b = legacychannelsim
legacychannelsim(true)
legacychannelsim(false)
oldmode = legacychannelsim(newmode)
```

Description

`b = legacychannelsim` returns `false` if the code you are running uses the R2009b (or later) version of the random number generator for `rayleighchan` or `ricianchan`. (By default, these use the 2009b random number generator.) It returns `true` if pre-R2009b versions are used. See Version 4.4. (R2009b) Communications Toolbox Release Notes for more information.

`legacychannelsim(true)` reverts the random number generation mode for channel objects to pre-2009b version.

Note `legacychannelsim(true)` will support the `reset(chan,randstate)` functionality.

`legacychannelsim(false)` sets the random number generation mode for channel objects to 2009b and later versions.

`oldmode = legacychannelsim(newmode)` sets the random number generation mode for channel objects to `newmode` and returns the previous mode, `oldmode`.

Compatibility Considerations

`legacychannelsim` has been removed

Errors starting in R2020b

legacychannelsim has been removed. Use `comm.RayleighChannel` or `comm.RicianChannel` instead.

Introduced in R2009b

lineareq

(To be removed) Construct linear equalizer object

Note will be removed in a future release. Use `comm.LinearEqualizer` instead.

Syntax

```
eqobj = lineareq(nweights,alg)
eqobj = lineareq(nweights,alg,sigconst)
eqobj = lineareq(nweights,alg,sigconst,nsamp)
```

Description

The `lineareq` function creates an equalizer object that you can use with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`eqobj = lineareq(nweights,alg)` constructs a symbol-spaced linear equalizer object. The equalizer has `nweights` complex weights, which are initially all zeros. `alg` describes the adaptive algorithm that the equalizer uses; you should create `alg` using any of these functions: `lms`, `signlms`, `normlms`, `varlms`, `rls`, or `cma`. The signal constellation of the desired output is `[-1 1]`, which corresponds to binary phase shift keying (BPSK).

`eqobj = lineareq(nweights,alg,sigconst)` specifies the signal constellation vector of the desired output.

`eqobj = lineareq(nweights,alg,sigconst,nsamp)` constructs a fractionally spaced linear equalizer object. The equalizer has `nweights` complex weights spaced at `T/nsamp`, where `T` is the symbol period and `nsamp` is a positive integer. `nsamp = 1` corresponds to a symbol-spaced equalizer.

Properties

The table below describes the properties of the linear equalizer object. To learn how to view or change the values of a linear equalizer object, see “Equalization”.

Tip To initialize or reset the equalizer object `eqobj`, enter `reset(eqobj)`.

Property	Description
`EqType`	Fixed value, `'Linear Equalizer'`
`AlgType`	Name of the adaptive algorithm represented by `alg`
`nWeights`	Number of weights

Property	Description
nSampPerSym	Number of input samples per symbol (equivalent to nsamp input argument). This value relates to both the equalizer structure (see the use of K in "Equalization") and an assumption about the signal to be equalized.
RefTap (except for CMA equalizers)	Reference tap index, between 1 and nWeights. Setting this to a value greater than 1 effectively delays the reference signal and the output signal by RefTap-1 with respect to the equalizer's input signal.
SigConst	Signal constellation, a vector whose length is typically a power of 2
Weights	Vector of complex coefficients. This is the set of w_i values in the schematic in "Equalization".
WeightInputs	Vector of tap weight inputs. This is the set of u_i values in the schematic in "Equalization".
ResetBeforeFiltering	If 1, each call to equalize resets the state of eqobj before equalizing. If 0, the equalization process maintains continuity from one call to the next.
NumSamplesProcessed	Number of samples the equalizer processed since the last reset. When you create or reset eqobj, this property value is 0.
Properties specific to the adaptive algorithm represented by alg	See reference page for the adaptive algorithm function that created alg: lms, signlms, normlms, varlms, rls, or cma.

Relationships Among Properties

If you change nWeights, MATLAB maintains consistency in the equalizer object by adjusting the values of the properties listed below.

Property	Adjusted Value
Weights	zeros(1,nWeights)
WeightInputs	zeros(1,nWeights)
StepSize (Variable-step-size LMS equalizers)	InitStep*ones(1,nWeights)
InvCorrMatrix (RLS equalizers)	InvCorrInit*eye(nWeights)

Examples

Configuring Linear Equalizers

This example configures the recommended comm.LinearEqualizer System object™ and the legacy lineareq feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r,x(1:100));
yNew = eqNew(r,x(1:100));
```

Use RLS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: 0.1000
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. For the `comm.LinearEqualizer` object, set the initial inverse correlation matrix to `eye(5)*0.2`.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ForgetFactor: 0.9500
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1, ...
    'InitialInverseCorrelationMatrix',eye(5)*0.2)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: [5×5 double]
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Use CMA Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from CMA algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,cma(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'Constant Modulus'
  nWeights: 5
  nSampPerSym: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [1 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','CMA','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'CMA'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputSamplesPerSymbol: 1
  AdaptWeightsSource: 'Property'
  AdaptWeights: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Compatibility Considerations

`lineareq` will be removed

Warns starting in R2020a

lineareq will be removed in a future release. Use `comm.LinearEqualizer` instead. For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-540.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

lloyds

Optimize quantization parameters using 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 “Represent Partitions”, “Represent Codebooks”, 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 condition 1 of the algorithm below, `reldistor` is the relative change in distortion between the last two iterations. In condition 2, `reldistor` is the same as `distor`.

Examples

The code below optimizes the quantization parameters for a sinusoidal transmission via a three-bit channel. Because the typical data is sinusoidal, `training_set` is a sampled sine wave. Because the channel can transmit three 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)
```

The output is below.

```
partition =

  Columns 1 through 6

   -0.8540   -0.5973   -0.3017    0.0031    0.3077    0.6023

  Column 7

    0.8572


codebook =

  Columns 1 through 6

   -0.9504   -0.7330   -0.4519   -0.1481    0.1558    0.4575

  Columns 7 through 8

    0.7372    0.9515
```

Algorithms

lloyds uses an iterative process to try to minimize the mean square distortion. The optimization processing ends when either

- The relative change in distortion between iterations is less than 10^{-7}.
- The distortion is less than eps*max(training_set), where eps is the MATLAB floating-point relative accuracy.

References

[1] Lloyd, S.P., “Least Squares Quantization in PCM,” *IEEE Transactions on Information Theory*, Vol. IT-28, March, 1982, pp. 129–137.

[2] Max, J., “Quantizing for Minimum Distortion,” *IRE Transactions on Information Theory*, Vol. IT-6, March, 1960, pp. 7–12.

See Also

dpcmopt | quantiz

Topics

“Source Coding”

Introduced before R2006a

lms

(To be removed) Construct least mean square (LMS) adaptive algorithm object

Note will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedbackEqualizer` instead.

Syntax

```
alg = lms(stepsize)
alg = lms(stepsize,leakagefactor)
```

Description

The `lms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`alg = lms(stepsize)` constructs an adaptive algorithm object based on the least mean square (LMS) algorithm with a step size of `stepsize`.

`alg = lms(stepsize,leakagefactor)` sets the leakage factor of the LMS algorithm. `leakagefactor` must be between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Properties

The table below describes the properties of the LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Fixed value, `'LMS'`
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Algorithms

Referring to the schematics presented in “Equalization”, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of weights, w, this adaptive algorithm creates the new set of weights given by

(`LeakageFactor`) w + (`StepSize`) u^*e

where the * operator denotes the complex conjugate.

Compatibility Considerations

`lms` will be removed

Warns starting in R2020a

- lms will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to LMS.
- The `comm.LinearEqualizer` or `comm.DecisionFeedback` System objects do not have a leakage factor property. This is equivalent to setting `LeakageFactor` to `1` in the lms function.
- For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-548.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

log

Logarithm in Galois field

Syntax

```
y = log(x)
```

Description

`y = log(x)` computes the logarithm of each element in the Galois array `x`. `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)
```

The output is

```
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])
```

The output is

```
yy =

     0     1
```

See Also

`gf`

Introduced before R2006a

marcumq

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

$$Q(a,b) = \int_b^\infty x\mathrm{exp}\left(-\frac{x^2+a^2}{2}\right)I_0(ax)dx$$

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

$$Q(a,b) = \frac{1}{a^{m-1}}\int_b^\infty x^m\mathrm{exp}\left(-\frac{x^2+a^2}{2}\right)I_{m-1}(ax)dx$$

where `a` and `b` are nonnegative real numbers, and `m` is a positive integer. In this expression, $I_{\mathrm{m-1}}$ is the modified Bessel function of the first kind of order *m*-1.

If any of the inputs is a scalar, it is expanded to the size of the other inputs.

Examples

Generate and Plot Marcum Q Function Data

This example shows how to use the `marcumq` function.

Create an input vector, `x`.

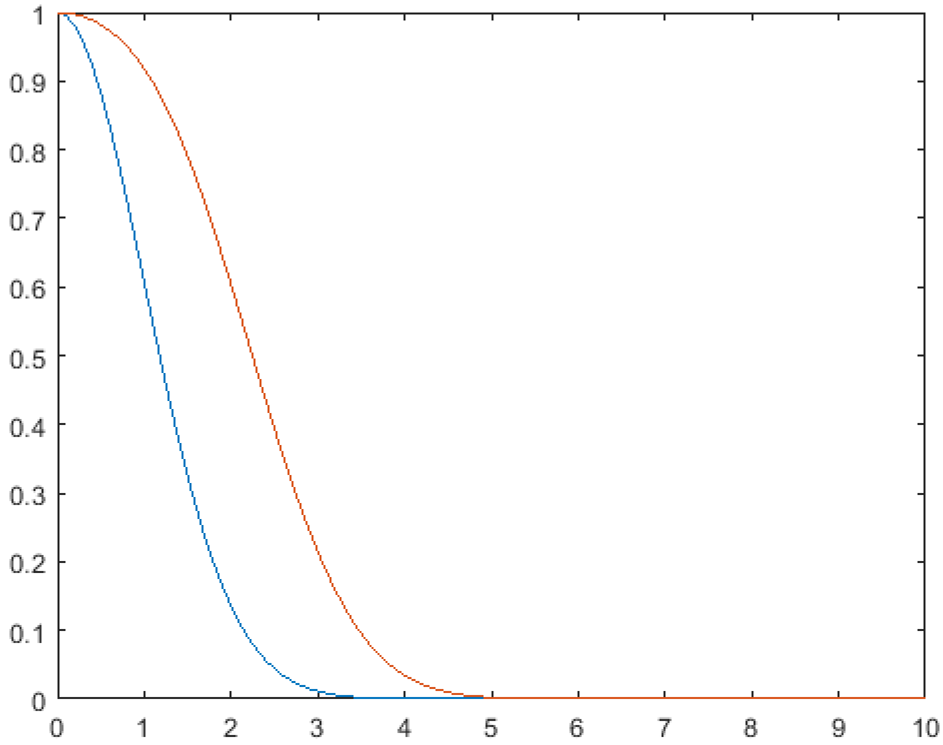
```
x = (0:0.1:10)';
```

Generate two output vectors for a=0 and a=2.

```
Q1 = marcumq(0,x);
Q2 = marcumq(2,x);
```

Plot the resultant Marcum Q functions.

```
plot(x,[Q1 Q2])
```



References

[1] 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.

[2] 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.

[3] Shnidman, D. A., “The Calculation of the Probability of Detection and the Generalized Marcum Q-Function,” *IEEE Transactions on Information Theory*, Vol. IT-35, March, 1989, pp. 389–400.

See Also

`besseli`

Introduced before R2006a

mask2shift

Convert mask vector to shift for 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 polynomial character vector
- 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, 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, 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

Convert Mask to Shift

Convert masks into shifts for a linear feedback shift register.

Convert a mask of $x^3 + 1$ into an equivalent shift for the linear feedback shift register whose connections are specified by the primitive polynomial $x^4 + x^3 + 1$.

```
s1 = mask2shift([1 1 0 0 1],[1 0 0 1])
```

```
s1 = 4
```

Convert a mask of `1` to a shift. The mask is equivalent to a shift of `0`.

```
s2 = mask2shift([1 1 0 0 1],[0 0 0 1])
```

```
s2 = 0
```

Convert a mask of x^2 into an equivalent shift for the primitive polynomial $x^3 + x + 1$.

```
s3 = mask2shift('x3+x+1','x2')
```

```
s3 = 2
```

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.

See Also

`isprimitive` | `log` | `primpoly` | `shift2mask`

Introduced before R2006a

matdeintrlv

Restore ordering of symbols by filling matrix by columns and emptying it by rows

Syntax

```
deintrlvd = matdeintrlv(data,Nrows,Ncols)
```

Description

deintrlvd = matdeintrlv(data,Nrows,Ncols) rearranges the elements in data by filling a temporary matrix with the elements column by column and then sending the matrix contents, row by row, to the output. Nrows and Ncols are the dimensions of the temporary matrix. If data is a vector, it must have Nrows*Ncols elements. If data is a matrix with multiple rows and columns, data must have Nrows*Ncols rows and the function processes the columns independently.

To use this function as an inverse of the matintrlv function, use the same Nrows and Ncols inputs in both functions. In that case, the two functions are inverses in the sense that applying matintrlv followed by matdeintrlv leaves data unchanged.

Examples

The code below illustrates the inverse relationship between matintrlv and matdeintrlv.

```
Nrows = 2; Ncols = 3;
data = [1 2 3 4 5 6; 2 4 6 8 10 12]';
a = matintrlv(data,Nrows,Ncols); % Interleave.
b = matdeintrlv(a,Nrows,Ncols) % Deinterleave.
```

The output below shows that b is the same as data.

```
b =

     1     2
     2     4
     3     6
     4     8
     5    10
     6    12
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

matintrlv

Topics

“Interleaving”

Introduced before R2006a

matintrlv

Reorder symbols by filling matrix by rows and emptying it by columns

Syntax

```
intrlvd = matintrlv(data,Nrows,Ncols)
```

Description

`intrlvd = matintrlv(data,Nrows,Ncols)` rearranges the elements in `data` by filling a temporary matrix with the elements row by row and then sending the matrix contents, column by column, to the output. `Nrows` and `Ncols` are the dimensions of the temporary matrix. If `data` is a vector, it must have `Nrows*Ncols` elements. If `data` is a matrix with multiple rows and columns, `data` must have `Nrows*Ncols` rows and the function processes the columns independently.

Examples

Apply Matrix Interleaving to Reorder Input Matrix

Use the `matintrlv` function to reorder the elements filling matrix by rows and emptying it by columns.

To form the first column of the output, the function creates the temporary 2-by-3 matrix [1 2 3; 4 5 6]. Then the function reads down each column of the temporary matrix to get [1 4 2 5 3 6].

```
b = matintrlv([1 2 3 4 5 6; 2 4 6 8 10 12]',2,3)
```

```
b = 6×2

     1     2
     4     8
     2     4
     5    10
     3     6
     6    12
```

To form the first column of the output, the function creates the temporary 3-by-2 matrix [1 2; 3 4; 5 6]. Then the function reads down each column of the temporary matrix to get [1 3 5 2 4 6].

```
b = matintrlv([1 2 3 4 5 6; 2 4 6 8 10 12]',3,2)
```

```
b = 6×2

     1     2
     3     6
     5    10
     2     4
     4     8
     6    12
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`matdeintrlv`

Topics

"Interleaving"

Introduced before R2006a

mil188qamdemod

MIL-STD-188-110 B/C standard-specific quadrature amplitude demodulation

Syntax

```
z = mil188qamdemod(y,M)
z = mil188qamdemod(y,M,Name,Value)
```

Description

`z = mil188qamdemod(y,M)` performs QAM demodulation on an input signal, `y`, that was modulated in accordance with MIL-STD-188-110 and the modulation order, `M`. For a description of MIL-STD-188-110 QAM demodulation, see “MIL-STD-188-110 QAM Hard Demodulation” on page 2-567 and “MIL-STD-188-110 QAM Soft Demodulation” on page 2-568.

`z = mil188qamdemod(y,M,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Demodulate MIL-STD-188-110B Specific 16-QAM Signal

Demodulate a 16-QAM signal that was modulated as specified in MIL-STD-188-110B. Plot the received constellation and verify that the output matches the input.

Set the modulation order and generate random data.

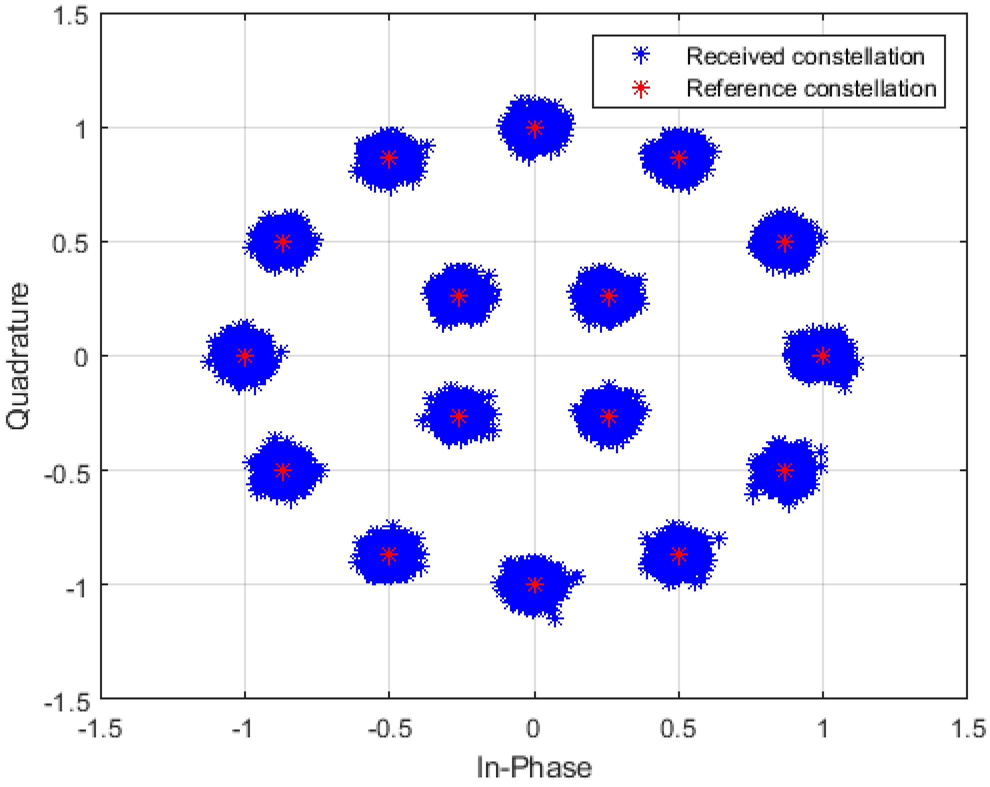
```
M = 16;
numSym = 20000;
x = randi([0 M-1],numSym,1);
```

Modulate the data and pass through a noisy channel.

```
txSig = mil188qammod(x,M);
rxSig = awgn(txSig,25,'measured');
```

Plot the transmitted and received signal.

```
plot(rxSig,'b*')
hold on; grid
plot(txSig,'r*')
xlim([-1.5 1.5]);
ylim([-1.5 1.5])
xlabel('In-Phase')
ylabel('Quadrature')
legend('Received constellation','Reference constellation')
```



Demodulate the received signal. Compare the demodulated data to the original data.

```
z = mil188qamdemod(rxSig,M);
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate MIL-STD-188-110C Specific 64-QAM Signal

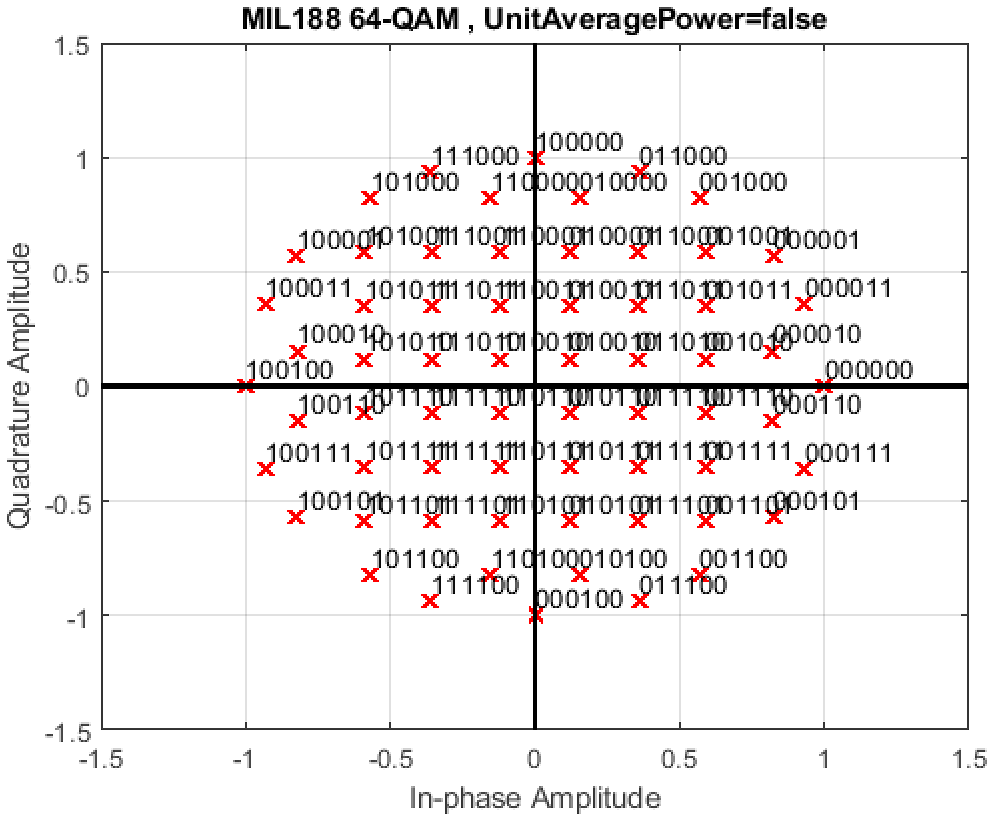
Demodulate a 64-QAM signal that was modulated as specified in MIL-STD-188-110C. Compute hard decision bit output and that verify the output matches the input.

Set the modulation order and generate random bit data.

```
M = 64;
numBitsPerSym = log2(M);
x = randi([0 1],1000*numBitsPerSym,1);
```

Modulate the data. Use name-value pairs to specify bit input data and to plot the constellation.

```
txSig = mil188qammod(x,M,'InputType','bit','PlotConstellation',true);
```



Demodulate the received signal. Compare the demodulated data to the original data.

```
z = mil188qamdemod(txSig,M,'OutputType','bit');
isequal(z,x)
```

```
ans = logical
   1
```

Soft Bit Demodulate MIL-STD-188-110 Specific 32-QAM Signal

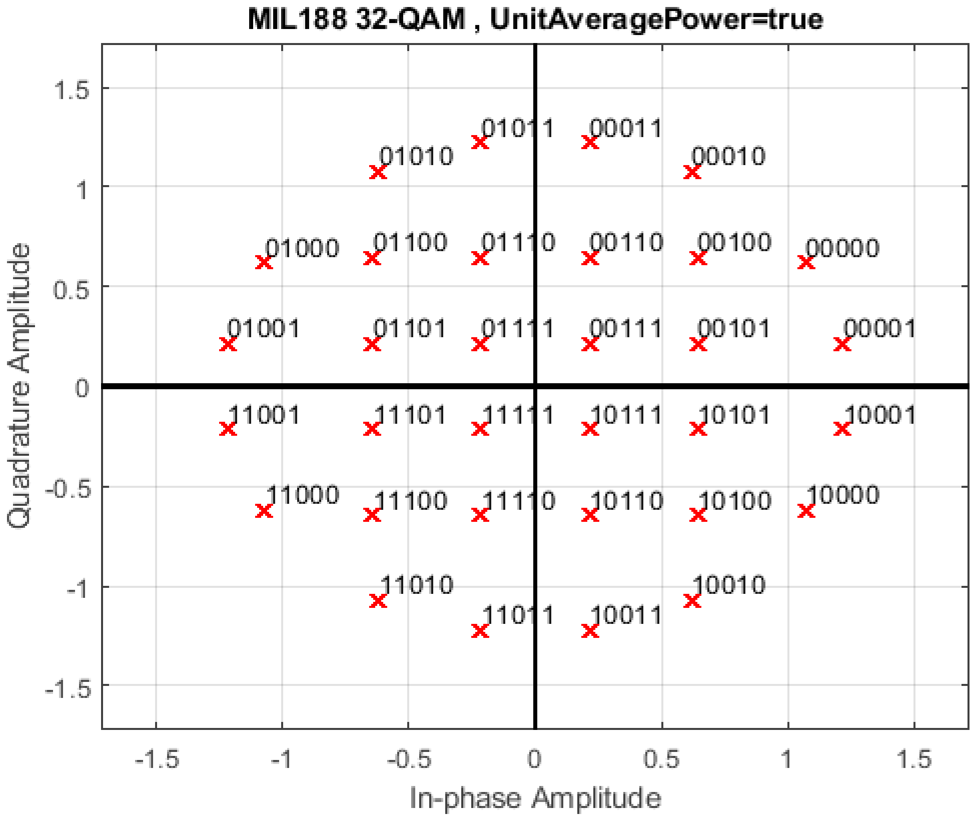
Demodulate a 32-QAM signal and calculate soft bits.

Set the modulation order and generate a random bit sequence.

```
M = 32;
numSym = 20000;
numBitsPerSym = log2(M);
x = randi([0 1], numSym*numBitsPerSym,1);
```

Modulate the data. Use name-value pairs to specify bit input data and unit average power, and to plot the constellation.

```
txSig = mil188qammod(x,M,'InputType','bit','UnitAveragePower',true, ...
    'PlotConstellation',true);
```

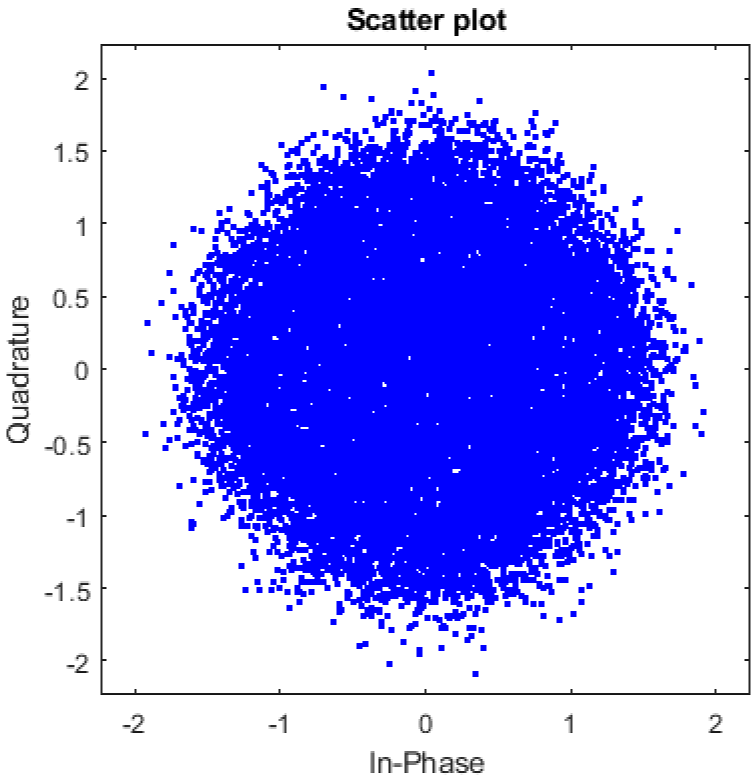


Pass the transmitted data through white Gaussian noise.

```
rxSig = awgn(txSig,10,'measured');
```

View the constellation using a scatter plot.

```
scatterplot(rxSig)
```



Demodulate the signal, computing soft bits using the approximate LLR algorithm.

```
z = mil188qamdemod(rxSig,M,'OutputType','approxllr', ...
    'NoiseVariance',10^(-1));
```

Input Arguments

y — Modulated signal

scalar | vector | matrix

Modulated signal, specified as a complex scalar, vector, or matrix. When y is a matrix, each column is treated as an independent channel.

y must be modulated in accordance with MIL-STD-188-110 [1].

Data Types: single | double
Complex Number Support: Yes

M — Modulation order

integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Example: 16

Data Types: double

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: y = mil188qamdemod(x,M,'OutputType','bit','OutputDataType','single');

OutputType — Output type
'integer' (default) | 'bit' | 'llr' | 'approxllr'

Output type, specified as the comma-separated pair consisting of OutputType and 'integer', 'bit', 'llr', or 'approxllr'.

Data Types: char | string

OutputDataType — Output data type
'double' (default) | ...

Output data type, specified as the comma-separated pair consisting of OutputDataType and one of the indicated data types. Acceptable values for OutputDataType depend on the OutputType value.

OutputType Value	Acceptable OutputDataType Values
'integer'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', or 'uint32'
'bit'	'double', 'single', 'int8', 'int16', 'int32', 'uint8', 'uint16', 'uint32', or 'logical'

Dependencies

This name-value pair argument applies only when OutputType is set to 'integer' or 'bit'.

Data Types: char | string

UnitAveragePower — Unit average power flag
false (default) | true

Unit average power flag, specified as the comma-separated pair consisting of UnitAveragePower and a logical scalar. When this flag is true, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is false, the function scales the constellation based on specifications in the relevant standard, as described in [1].

Data Types: logical

NoiseVariance — Noise variance
1 (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of NoiseVariance and a positive scalar or vector of positive values.

- When specified as a scalar, the same noise variance value is used on all input elements.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “MIL-STD-188-110 QAM Soft Demodulation” on page 2-568 for algorithm selection considerations.

Dependencies

This name-value pair argument applies only when `OutputType` is set to `'llr'` or `'approxllr'`.

Data Types: `double`

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

z — Demodulated signal

scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of `z` depend on the specified `OutputType` value.

`OutputType` Value	Return Value of `mil188qamdemod`	Dimensions of `z`
`'integer'`	Demodulated integer values from 0 to (`M` – 1)	`z` has the same dimensions as input `y`.
`'bit'`	Demodulated bits	The number of rows in `z` is $\log_2$(sum(`M`)) times the number of rows in `y`. Each demodulated symbol is mapped to a group of $\log_2$(sum(`M`)) elements in a column, where the first element represents the MSB and the last element represents the LSB.
`'llr'`	Log-likelihood ratio value for each bit	
`'approxllr'`	Approximate log-likelihood ratio value for each bit	

More About

MIL-STD-188-110 QAM Hard Demodulation

The hard demodulation algorithm uses optimum decision region-based demodulation. Since all the constellation points are equally probable, maximum a posteriori probability (MAP) detection reduces to a maximum likelihood (ML) detection. The ML detection rule is equivalent to choosing the closest constellation point to the received symbol. The decision region for each constellation point is designed by drawing perpendicular bisectors between adjacent points. A received symbol is mapped to the proper constellation point based on which decision region it lies in.

Since all MIL-STD constellations are quadrant-based symmetric, for each symbol the optimum decision region-based demodulation:

- Maps the received symbol into the first quadrant

- Chooses the decision region for the symbol
- Maps the constellation point back to its original quadrant using the sign of real and imaginary parts of the received symbol

MIL-STD-188-110 QAM Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems." Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`apskdemod` | `dvbsapskdemod` | `genqamdemod` | `mil188qammod` | `pskdemod` | `qamdemod`

Objects

`comm.GeneralQAMDemodulator` | `comm.PSKDemodulator`

Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018a

mil188qammod

MIL-STD-188-110 B/C standard-specific quadrature amplitude modulation (QAM)

Syntax

```
y = mil188qammod(x,M)
y = mil188qammod(x,M,Name,Value)
```

Description

`y = mil188qammod(x,M)` performs QAM modulation on the input signal, `x`, in accordance with MIL-STD-188-110 and the modulation order, `M`. For more information, see “MIL-STD-188-110” on page 2-574.

`y = mil188qammod(x,M,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'OutputDataType','double'` specifies the desired output data type as double. Specify name-value pair arguments after all other input arguments.

Examples

Apply 32-QAM to Data Per MIL-STD-188-110C

Modulate data using 32-QAM as specified in the MIL-188-110C standard. Display the result using a scatter plot.

Set M to 32 and create a data vector containing all possible symbols.

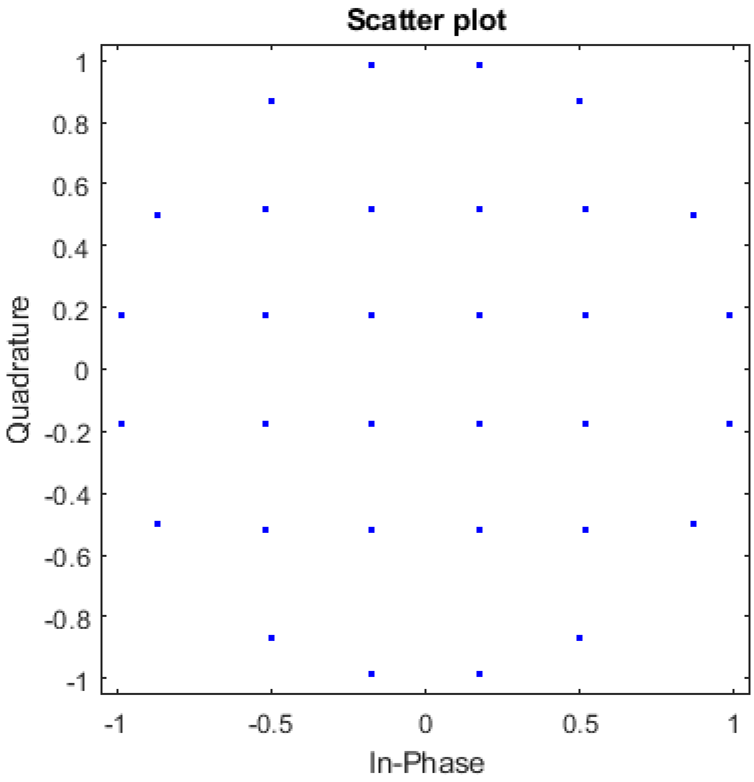
```
M = 32;
x = (0:M-1);
```

Modulate the data using QAM as specified in MIL-STD-188-110C.

```
y = mil188qammod(x,M);
```

Display the constellation as a scatter plot.

```
scatterplot(y)
```



Normalize 16-QAM Modulated MIL-STD-188-110B Signal by Average Power

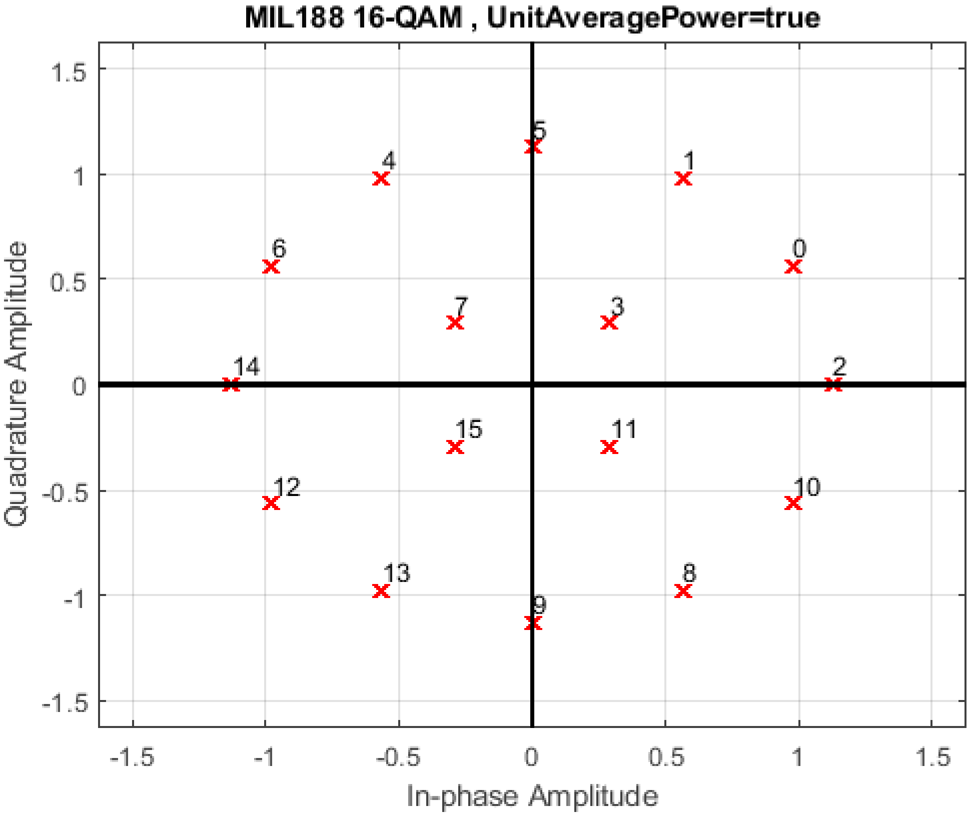
Modulate data using 16-QAM as specified in the MIL-STD-188-110B standard. Normalize the modulator output so that it has an average signal power of 1 W.

Set M and generate random data.

```
M = 16;
x = randi([0 M-1],1e5,1);
```

Modulate the data applying 16-QAM as specified in MIL-STD-188-110B. Using name-value pairs, set the unit average power to `true` and enable the constellation plot.

```
y = mil188qammod(x,M,'UnitAveragePower',true,'PlotConstellation',true);
```



Verify that the signal has approximately unit average power.

```
avgPow = mean(abs(y).^2)
```

```
avgPow = 1.0012
```

Apply 64-QAM MIL-STD-188-110B Modulation to Bit Data

Modulate a sequence of bits using 64-QAM as specified by MIL-STD188-110B. Display the constellation.

Set the modulation order and generate a sequence of random bits.

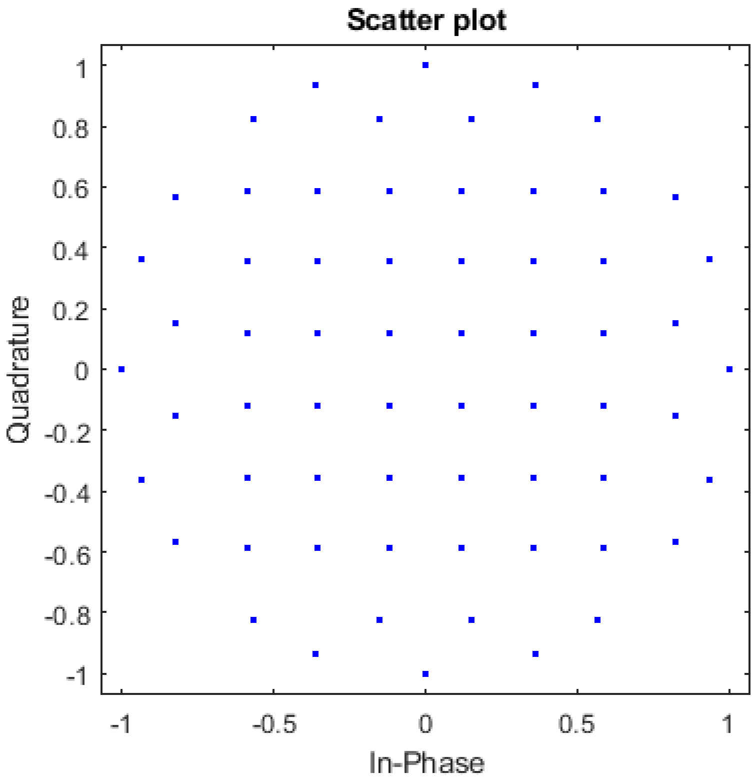
```
M = 64;
numBitsPerSym = log2(M);
data = randi([0 1],1000*numBitsPerSym,1);
```

Modulate the data applying 64-QAM as specified by MIL-STD-188-110B, and output constellation symbols of single data type.

```
y = mil188qammod(data,M,'InputType','bit','OutputDataType','single');
```

Plot the result constellation using a scatter plot.

```
scatterplot(y)
```



Input Arguments

x — Input signal

scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The elements of x must be binary values or integers that range from 0 to (`M` – 1), where `M` is the modulation order.

Note To process input signal as binary elements, set the `'InputType'` value to `'bit'`. For binary inputs, the number of rows must be an integer multiple of log$_2$(`M`). Groups of log$_2$(`M`) bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `logical`

`M` — Modulation order

integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the signal constellation.

Example: 16

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: y = mil188qammod(data,M,'InputType','bit','OutputDataType','single');

`InputType` — Input type

`'integer'` (default) | `'bit'`

Input type, specified as the comma-separated pair consisting of `'InputType'` and either `'integer'` or `'bit'`. If you specify `'integer'`, the input signal must consist of integers from 0 to `M` – 1. If you specify `'bit'`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2$(`M`).

Data Types: `char` | `string`

`OutputDataType` — Output data type

`'double'` (default) | `'single'`

Output data type, specified as the comma-separated pair consisting of `OutputDataType` and `'double'` or `'single'`.

Data Types: `char` | `string`

`UnitAveragePower` — Unit average power flag

`false` (default) | `true`

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a logical scalar. When this flag is `true`, the function scales the constellation to an average power of 1 watt referenced to 1 ohm. When this flag is `false`, the function scales the constellation based on specifications in the relevant standard, as described in [1].

Data Types: `logical`

`PlotConstellation` — Option to plot constellation

`false` (default) | `true`

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a logical scalar. To plot the constellation, set `PlotConstellation` to `true`.

Data Types: `logical`

Output Arguments

y — Modulated signal

scalar | vector | matrix

Modulated signal, returned as a complex scalar, vector, or matrix. The dimension of the output depends on the specified `InputType` value.

InputType	Dimensions of Output
`'integer'`	`y` has the same dimensions as input `x`.

InputType	Dimensions of Output
'bit'	The number of rows in y equals the number of rows in x divided by $\log_2$(M).

Data Types: double | single

More About

MIL-STD-188-110

MIL-STD-188-110 is a US Department of Defense standard for HF communications using serial PSK mode of both data and voice signals.

The standard specifies physical layer modulation schemes for tactical and long-haul communications. The modulation scheme specified by the standard is a mix of QAM and APSK. For a detailed description of the modulation scheme, see [1].

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems". Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

apskmod | dvbsapskmod | genqammod | mil188qamdemod | pskmod | qammod

Objects

comm.GeneralQAMModulator | comm.PSKModulator

Introduced in R2018a

minpol

Find minimal polynomial of Galois field element

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”.

See Also

`cosets` | `gf`

Topics

“Polynomials over Galois Fields”

Introduced before R2006a

mldivide

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)
```

The output is below.

```
ck1 =

     1


ck2 =

     1
```

Other examples are in “Solving Linear Equations”.

Limitations

The matrix `A` must be one of these types:

- A nonsingular square matrix
- A matrix, in which there are more rows than columns, such that `A'*A` is nonsingular
- A matrix, in which there are more columns than rows, such that `A*A'` is nonsingular

Algorithms

If `A` is an M-by-N matrix where M > N, `A \ B` is the same as `(A'*A) \ (A'*B)`.

If `A` is an M-by-N matrix where M < N, `A \ B` is the same as `A' * ((A*A') \ B)`. This solution is not unique.

See Also

`gf`

Topics

“Linear Algebra in Galois Fields”

Introduced before R2006a

mlseeq

Equalize linearly modulated signal using MLSE

Syntax

```
y = mlseeq(x,chcffs,const,tblen,opmode)
y = mlseeq( ___ ,nsamp)

y = mlseeq( ___ ,nsamp,preamble,postamble)

y = mlseeq( ___ ,nsamp,init_metric,init_states,init_inputs)
[y,final_metric,final_states,final_inputs] = mlseeq( ___ )
```

Description

`y = mlseeq(x,chcffs,const,tblen,opmode)` equalizes the baseband signal vector `x` using the maximum likelihood sequence estimation (MLSE). `chcffs` provides estimated channel coefficients. `const` provides the ideal signal constellation points. `tblen` specifies the traceback depth. `opmode` specifies the operation mode of the equalizer. MLSE is implemented using the “Viterbi Algorithm” on page 2-586.

`y = mlseeq( ___ ,nsamp)` specifies the number of samples per symbol in `x`, in addition to arguments in the previous syntax.

`y = mlseeq( ___ ,nsamp,preamble,postamble)` specifies the number of samples per symbol in `x`, preamble, and postamble, in addition to arguments in the first syntax. This syntax applies when `opmode` is `'rst'` only. For more information, see “Preamble and Postamble in Reset Operation Mode” on page 2-586.

`y = mlseeq( ___ ,nsamp,init_metric,init_states,init_inputs)`specifies the number of samples per symbol in `x`, initial likelihood state metrics, initial traceback states, and initial traceback inputs for the equalizer, in addition to arguments in the first syntax. These three inputs are typically the `final_metric`, `final_states`, and `final_inputs` outputs from a previous call to this function. This syntax applies when `opmode` is `'cont'` only. For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

`[y,final_metric,final_states,final_inputs] = mlseeq( ___ )` returns the normalized final likelihood state metrics, final traceback states, and final traceback inputs at the end of the traceback decoding process, using any of the previous input argument syntaxes. This syntax applies when `opmode` is `'cont'` only. For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

Examples

Using MLSE Equalizer Reset Operating Mode

Use the reset operating mode of the `mlseeq` equalizer. Demodulate the signal and check the bit error rate.

Specify the modulation order, equalizer traceback depth, number of samples per symbol, and message length.

```
M = 2;
tblen =  10;
nsamp = 2;
msgLen = 1000;
```

Generate the reference constellation.

```
const = pammod([0:M-1],M);
```

Generate a message with random data. Modulate and upsample the signal.

```
msgData = randi([0 M-1],msgLen,1);
msgSym = pammod(msgData,M);
msgSymUp = upsample(msgSym,nsamp);
```

Filter the data through a distortion channel and add Gaussian noise to the signal.

```
chanest = [0.986; 0.845; 0.237; 0.12345+0.31i];
msgFilt = filter(chanest,1,msgSymUp);
msgRx = awgn(msgFilt,5,'measured');
```

Equalize and then demodulate the signal to recover the message. To initialize the equalizer, provide the channel estimate, reference constellation, equalizer traceback depth, number of samples per symbol, and set the operating mode to reset. Check the message bit error rate. Your results might vary because this example uses random numbers.

```
eqSym = mlseeq(msgRx,chanest,const,tblen,'rst',nsamp);
eqMsg = pamdemod(eqSym,M);
```

```
[nerrs ber] = biterr(msgData, eqMsg)
```

```
nerrs = 1
```

```
ber = 1.0000e-03
```

Recover Message Containing Preamble

Recover a message that includes a preamble, equalize the signal, and check the symbol error rate.

Specify the modulation order, equalizer traceback depth, number of samples per symbol, preamble, and message length.

```
M = 4;
tblen = 16;
nsamp = 1;
preamble = [3;1];
msgLen = 500;
```

Generate the reference constellation.

```
const = pskmod(0:3,4);
```

Generate a message by using random data and prepend the preamble to the message. Modulate the random data.

```
msgData = randi([0 M-1],msgLen,1);
msgData = [preamble; msgData];
msgSym = pskmod(msgData,M);
```

Filter the data through a distortion channel and add Gaussian noise to the signal.

```
chcoeffs = [0.623; 0.489+0.234i; 0.398i; 0.21];
chanest = chcoeffs;
msgFilt = filter(chcoeffs,1,msgSym);
msgRx = awgn(msgFilt,9,'measured');
```

Equalize the received signal. To configure the equalizer, provide the channel estimate, reference constellation, equalizer traceback depth, operating mode, number of samples per symbol, and preamble. The same preamble symbols appear at the beginning of the message vector and in the syntax for `mlseeq`. Because the system does not use a postamble, an empty vector is specified as the last input argument in this `mlseeq` syntax.

Check the symbol error rate of the equalized signal. Run-to-run results vary due to use of random numbers.

```
eqSym = mlseeq(msgRx,chanest,const,tblen,'rst',nsamp,preamble,[]);
[nsymerrs,ser] = symerr(msgSym,eqSym)
```

```
nsymerrs = 8
```

```
ser = 0.0159
```

Using MLSE Equalizer Continuous Operating Mode

Use the continuous operating mode of the `mlseeq` equalizer. Demodulate received signal packets and check the symbol error statistics.

Specify the modulation order, equalizer traceback depth, number of samples per symbol, message length, and number of packets to process.

```
M = 4;
tblen =  10;
nsamp = 1;
msgLen = 1000;
numPkts = 25;
```

Generate the reference constellation.

```
const = pskmod(0:M-1,M);
```

Set the initial input parameters for the metric, states, and inputs of the equalizer to empty vectors. These initial assignments represent the parameters for the first packet transmitted.

```
eq_metric = [];
eq_states = [];
eq_inputs = [];
```

Assign variables for symbol error statistics.

```
ttlSymbErrs = 0;
aggrPktSER = 0;
```

Send and receive multiple message packets in a simulation loop. Between the packet transmission and reception filter each packet through a distortion channel and add Gaussian noise.

```
for jj = 1:numPkts
```

Generate a message with random data. Modulate the signal.

```
    msgData = randi([0 M-1],msgLen,1);
    msgMod = pskmod(msgData,M);
```

Filter the data through a distortion channel and add Gaussian noise to the signal.

```
    chanest = [.986; .845; .237; .12345+.31i];
    msgFilt = filter(chanest,1,msgMod);
    msgRx = awgn(msgFilt,10,'measured');
```

Equalize the received symbols. To configure the equalizer, provide the channel estimate, reference constellation, equalizer traceback depth, operating mode, number of samples per symbol, and the equalizer initialization information. Continuous operating mode is specified for the equalizer. In continuous operating mode, the equalizer initialization information (metric, states, and inputs) are returned and used as inputs in the next iteration of the `for` loop.

```
    [eqSym,eq_metric,eq_states,eq_inputs] = ...
        mlseeq(msgRx,chanest,const,tblen,'cont',nsamp, ...
        eq_metric,eq_states,eq_inputs);
```

Save the symbol error statistics. Update the symbol error statistics with the aggregate results. Display the total number of errors. Your results might vary because this example uses random numbers.

```
    [nsymerrs,ser] = symerr(msgMod(1:end-tblen),eqSym(tblen+1:end));
    ttlSymbErrs = ttlSymbErrs + nsymerrs;
    aggrPktSER = aggrPktSER + ser;
end
printTtlErr = 'A total of %d symbol errors over the %d packets received.\n';
fprintf(printTtlErr,ttlSymbErrs,numPkts);
```

```
A total of 155 symbol errors over the 25 packets received.
```

Display the aggregate symbol error rate.

```
printAggrSER = 'The aggregate symbol error rate was %6.5d.\n';
fprintf(printAggrSER,aggrPktSER/numPkts);
```

```
The aggregate symbol error rate was 6.26263e-03.
```

Input Arguments

x — Input signal

vector

Input signal, specified as a vector of modulated symbols. The vector length of `x` must be an integer multiple of `nsamp`.

Data Types: `double`
Complex Number Support: Yes

`chcffs` — Channel coefficients

vector

Channel coefficients, specified as a vector. The channel coefficients provide an estimate of the channel response. When `nsamp` > 1, the `chcffs` input specifies the oversampled channel coefficients.

Data Types: `double`
Complex Number Support: Yes

`const` — Reference constellation

vector

Reference constellation, specified as a vector with *M* elements. *M* is the modulation order. `const` lists the ideal signal constellation points in the sequence used by the modulator.

Data Types: `double`
Complex Number Support: Yes

`tblen` — Traceback depth

positive integer

Traceback depth, specified as a positive integer. The equalizer traces back from the likelihood state with the maximum metric.

Data Types: `double`

`opmode` — Operation mode

`'rst'` | `'cont'`

Operation mode, specified as `'rst'` or `'cont'`.

Value	Usage
`'rst'`	Run equalizer using reset operating mode. Enables you to specify a `preamble` and `postamble` that accompany the input signal. The function processes the input signal, `x`, independently of the input signal from any other invocations of this function. This operating mode does not incur an output delay. For more information, see “Preamble and Postamble in Reset Operation Mode” on page 2-586.
`'cont'`	Run equalizer using continuous operating mode. Enables you to save the internal state information of the equalizer for use in a subsequent invocation of this function. Continuous operating mode is useful if the input signal is partitioned into a stream of packets processed within a loop. This operating mode incurs an output delay of `tblen` symbols. For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

Data Types: `char`

`nsamp` — Number of samples per symbol

`1` (default) | positive integer

Number of samples per symbol, specified as a positive integer. `nsamp` is the oversampling factor.

Dependencies

The input signal, `x`, must be an integer multiple of `nsamp`.

Data Types: `double`

`preamble` — Input signal preamble

vector of integers

Input signal preamble, specified as a vector of integers between 0 and *M*–1, where *M* is the modulation order. To omit a preamble, specify `[]`.

For more information, see "Preamble and Postamble in Reset Operation Mode" on page 2-586.

Dependencies

This input argument applies only when `opmode` is set to `'rst'`.

Data Types: `double`

`postamble` — Input signal postamble

vector of integers

Input signal postamble, specified as a vector of integers between 0 and *M*–1, where *M* is the modulation order. To omit a postamble, specify `[]`.

For more information, see "Preamble and Postamble in Reset Operation Mode" on page 2-586.

Dependencies

This input argument applies only when `opmode` is set to `'rst'`.

Data Types: `double`

`init_metric` — Initial state metrics

[ ] (default) | column vector

Initial state metrics, specified as a column vector with N_{states} elements. For the description of N_{states}, see "Number of Likelihood States" on page 2-587.

For more information, see "Initialization in Continuous Operation Mode" on page 2-586.

Dependencies

This input argument applies only when `opmode` is set to `'cont'`. If specifying `[]` for `init_metric`, you must also specify `[]` for `init_states` and `init_inputs`.

Data Types: `double`

`init_states` — Initial traceback states

[ ] (default) | matrix of integers

Initial traceback states, specified as an N_{states}-by-`tblen` matrix of integers with values between 0 and N_{states}–1. For the description of N_{states}, see "Number of Likelihood States" on page 2-587.

For more information, see "Initialization in Continuous Operation Mode" on page 2-586.

Dependencies

This input argument applies only when `opmode` is set to `'cont'`. If specifying `[]` for `init_states`, you must also specify `[]` for `init_metric` and `init_inputs`.

Data Types: `double`

`init_inputs` — Initial traceback inputs

`[]` (default) | matrix of integers

Initial traceback inputs, specified as an N_{states}-by-`tblen` matrix of integers with values between 0 and M–1. For the description of N_{states}, see “Number of Likelihood States” on page 2-587.

For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

Dependencies

This input argument applies only when `opmode` is set to `'cont'`. If specifying `[]` for `init_inputs`, you must also specify `[]` for `init_metric` and `init_states`.

Data Types: `double`

Output Arguments

y — Output signal

vector

Output signal, returned as a vector of modulated symbols.

`final_metric` — Final normalized state metrics

vector

Final normalized state metrics, returned as a vector with N_{states} elements. `final_metric` corresponds to the final state metrics at the end of the traceback decoding process. For the description of N_{states}, see “Number of Likelihood States” on page 2-587.

For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

`final_states` — Final traceback states

vector

Final traceback states, returned as a N_{states}-by-`tblen` matrix of integers with values between 0 and N_{states}–1. `final_states` corresponds to the final traceback states at the end of the traceback decoding process. For the description of N_{states}, see “Number of Likelihood States” on page 2-587.

For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

`final_inputs` — Final traceback inputs

vector

Final traceback inputs, returned as an N_{states}-by-`tblen` matrix of integers with values between 0 and M–1. `final_inputs` corresponds to the final traceback inputs at the end of the traceback decoding process. M is the order of the modulation. For the description of N_{states}, see “Number of Likelihood States” on page 2-587.

For more information, see “Initialization in Continuous Operation Mode” on page 2-586.

More About

Viterbi Algorithm

The Viterbi algorithm is a sequential trellis search algorithm used to perform maximum likelihood sequence detection.

The MLSE equalizer uses the Viterbi algorithm to recursively search for the sequences that maximize the likelihood function. Using the Viterbi algorithm reduces the number of sequences in the trellis search by eliminating sequences as new data is received. The metric used to determine the maximum likelihood sequence is the correlation between the received signal and an estimated signal for each received symbol over the “Number of Likelihood States” on page 2-587.

For more information, see [1] and [2].

Preamble and Postamble in Reset Operation Mode

When operating the MLSE equalizer in reset mode, you can specify a preamble and postamble as input arguments. Specify `preamble` and `postamble` as vectors equal to the preamble and postamble that are prepended and appended, respectively, to the input signal. The `preamble` and `postamble` vectors consist of integers between 0 and *M*-1, where *M* is the number of elements in `const`. To omit the `preamble` or `postamble` input argument, specify `[]`.

When the function applies the Viterbi algorithm, it initializes state metrics in a way that depends on whether you specify a preamble, a postamble, or both:

- If `preamble` is nonempty, the function decodes the preamble and assigns a metric of 0 to the decoded state. If the preamble does not decode to a unique state (that is, if the length of the preamble is less than the channel memory), the decoder assigns a metric of 0 to all states that are represented by the preamble. The traceback path ends at one of the states represented by the preamble.
- If `preamble` is `[]`, the decoder initializes the metrics of all states to 0.
- If `postamble` is nonempty, the traceback path begins at the smallest of all possible decoded states that are represented by the postamble.
- If `postamble` is `[]`, the traceback path starts at the state with the smallest metric.

Initialization in Continuous Operation Mode

When operating the MLSE equalizer in continuous mode, you can initialize the equalization based on values returned in the previous call of the function.

At the end of the traceback decoding process, the function returns `final_metric`, `final_states`, and `final_inputs`. When `opmode` is `'cont'`, assign these outputs to `init_metric`, `init_states`, and `init_inputs`, respectively for the next call of the function. These assignments initialize the equalizer to start with the final state metrics, final traceback states, and final traceback inputs from the previous call of the function.

Each real number in `init_metric` represents the starting state metric of the corresponding state. `init_states` and `init_inputs` jointly specify the initial traceback memory of the equalizer.

Output Argument	Input Argument	Meaning	Matrix Size	Range of Values
`final_metric`	`init_metric`	State metrics	1-by-N_{states}	Real numbers
`final_states`	`init_states`	Traceback states	N_{states}-by-`tblen`	Integers between 0 and N_{states}–1
`fianl_inputs`	`init_inputs`	Traceback inputs	N_{states}-by-`tblen`	Integers between 0 and M–1

To use default values for `init_metric`, `init_states`, and `init_inputs`, specify each as `[]`. For the description of N_{states}, see “Number of Likelihood States” on page 2-587.

Number of Likelihood States

The number of likelihood states, N_{states}, is the number of correlative phase states in the trellis. N_{states} is equal to $M^{L\text{-}1}$, where M is the number of elements in `const` and L is the number of symbols in the nonoversampled impulse response of the channel.

References

[1] Proakis, John G. *Digital Communications*, Fourth Edition. New York: McGraw-Hill, 2001.

[2] Steele, Raymond, Ed. *Mobile Radio Communications*. Chichester, England: John Wiley & Sons, 1996.

See Also

Functions
`dfe` | `lineareq` | `lms` | `rls`

Objects
`comm.MLSEEqualizer`

Topics
“MLSE Equalizers”
“Recover Message Containing Preamble”
“Use mlseeq to Equalize a Vector in Continuous Operation Mode”

Introduced before R2006a

modnorm

Scaling factor for normalizing modulation output

Syntax

```
normfactor = modnorm(refconst,type,power)
```

Description

`normfactor = modnorm(refconst,type,power)` returns a scale factor for normalizing a PAM or QAM modulator output using the specified reference constellation, normalization type, and output power.

Examples

Normalize Power of QAM Signal

Generate a 16-QAM reference constellation.

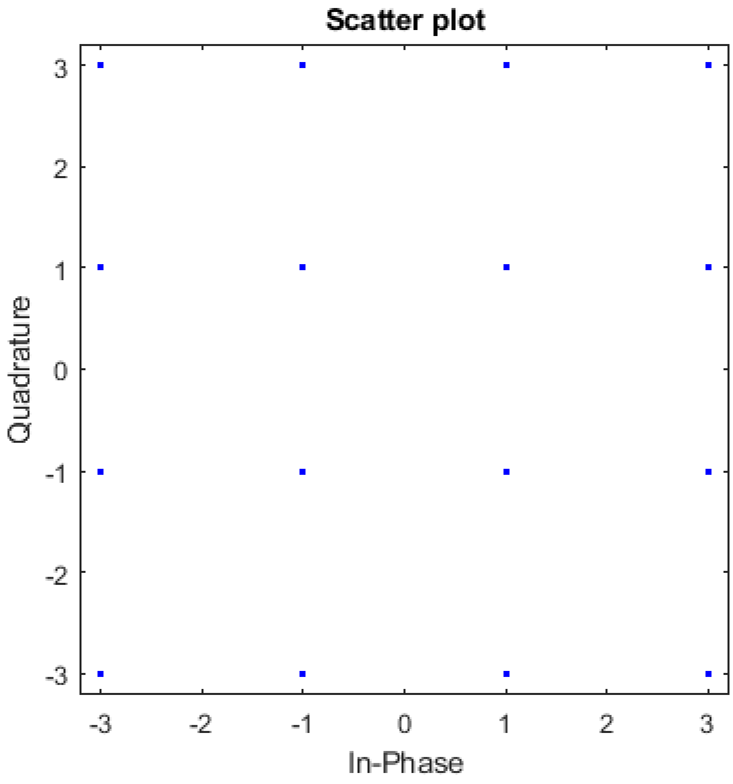
```
refconst = qammod(0:15,16);
```

Generate random symbols and apply 16-QAM modulation.

```
x = randi([0 15],1000,1);
y = qammod(x,16);
```

Plot the constellation.

```
h = scatterplot(y);
```



Compute the normalization factor so that the output signal has a peak power of 1 W.

```
nf = modnorm(refconst,'peakpow',1);
z = nf*y;
```

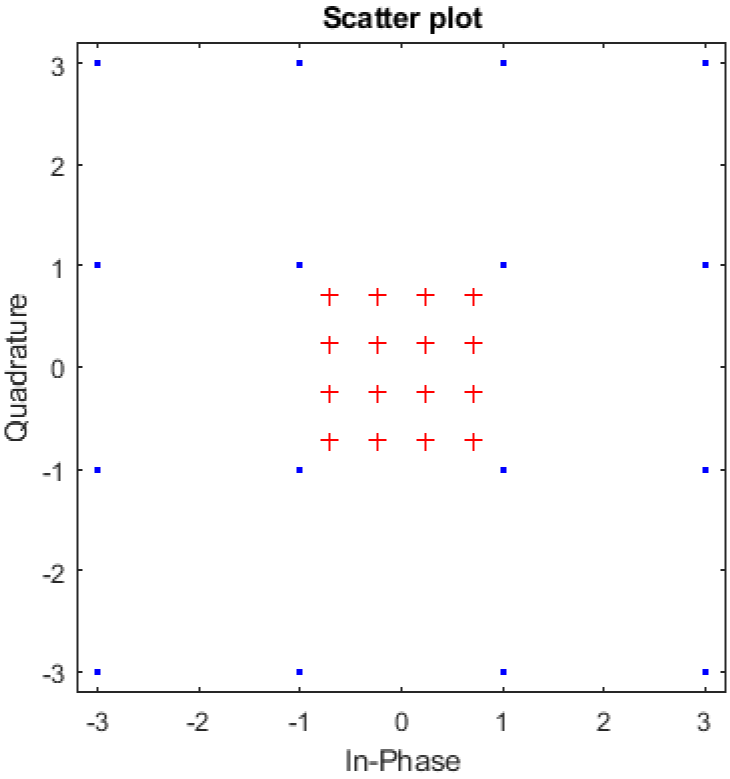
Confirm that no element of the normalized signal has a power greater than 1 W.

```
max(z.*conj(z))
```

```
ans = 1.0000
```

Plot the scatter plot of the normalized constellation.

```
hold on
scatterplot(z,1,0,'r+',h)
hold off
```



Input Arguments

`refconst` — Reference constellation

vector

Reference constellation, specified as a vector of complex elements that comprise the reference constellation points.

Example: `qammod(0:15,16)`

Data Types: `double`
Complex Number Support: Yes

`type` — Normalization type

`'avpow'` | `'peakpow'`

Normalization type, specified as either `'avpow'` or `'peakpow'`.

- If `type` is `'avpow'`, the normalization factor is calculated based on average power.
- If `type` is `'peakpow'`, the normalization factor is calculated based on peak power.

Data Types: `char`

`power` — Target power

scalar

Target power, specified as a real scalar. The target power is the intended power of the modulated signal multiplied by `normfactor`.

Data Types: `double`

Output Arguments

`normfactor` — Normalization factor

scalar

Normalization factor, returned as a real scalar. When a modulated signal is multiplied by the normalization factor, its average or peak power matches the target power. The function assumes that the signal you want to normalize has a minimum distance of 2.

See Also

`pamdemod` | `pammod` | `qamdemod` | `qammod`

Introduced before R2006a

mskdemod

Minimum shift keying demodulation

Syntax

```
z = mskdemod(y,nsamp)
z = mskdemod(y,nsamp,dataenc)
z = mskdemod(y,nsamp,dataenc,ini_phase)
z = mskdemod(y,nsamp,dataenc,ini_phase,ini_state)
[z,phaseout] = mskdemod(...)
[z,phaseout,stateout] = mskdemod(...)
```

Description

`z = mskdemod(y,nsamp)` demodulates the complex envelope `y` of a signal using the differentially encoded minimum shift keying (MSK) method. `nsamp` denotes the number of samples per symbol and must be a positive integer. The initial phase of the demodulator is 0. If `y` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`z = mskdemod(y,nsamp,`*`dataenc`*`)` specifies the method of encoding data for MSK. *`dataenc`* can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

`z = mskdemod(y,nsamp,`*`dataenc`*`,ini_phase)` specifies the initial phase of the demodulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of *`dataenc`*, set *`dataenc`* to `[]`.

`z = mskdemod(y,nsamp,`*`dataenc`*`,ini_phase,ini_state)` specifies the initial state of the demodulator. `ini_state` contains the last half symbol of the previously received signal. `ini_state` is an `nsamp`-by-C matrix, where C is the number of channels in `y`.

`[z,phaseout] = mskdemod(...)` returns the final phase of `y`, which is important for demodulating a future signal. The output `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values `0`, `pi/2`, `pi`, and `3*pi/2`.

`[z,phaseout,stateout] = mskdemod(...)` returns the final `nsamp` values of `y`, which is useful for demodulating the first symbol of a future signal. `stateout` has the same dimensions as the `ini_state` input.

Examples

MSK Demodulation

Modulate and demodulate a noisy MSK signal. Display the number of received errors.

Define the number of samples per symbol for the MSK signal.

```
nsamp = 16;
```

Initialize the simulation parameters.

```
numerrs = 0;
modPhase = zeros(1,2);
demodPhase = zeros(1,2);
demodState = complex(zeros(nsamp,2));
```

The main processing loop includes these steps:

- Generate binary data.
- MSK modulate the data.
- Pass the signal through an AWGN channel.
- Demodulate the MSK signal.
- Determine the number of bit errors.

```
for iRuns = 1:20
    txData = randi([0 1],100,2);
    [modSig,modPhase] = mskmod(txData,nsamp,[],modPhase);
    rxSig = awgn(modSig,20,'measured');
    [rxData,demodPhase,demodState] = mskdemod(rxSig,nsamp,[],demodPhase,demodState);
    numerrs = numerrs + biterr(txData,rxData);
end
```

Display the number of bit errors.

```
numerrs
```

```
numerrs = 0
```

References

[1] Pasupathy, Subbarayan, “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`comm.MSKDemodulator` | `fskdemod` | `fskmod` | `mskmod`

Topics

“Digital Modulation”

Introduced before R2006a

mskmod

Minimum shift keying modulation

Syntax

```
y = mskmod(x,nsamp)
y = mskmod(x,nsamp,dataenc)
y = mskmod(x,nsamp,dataenc,ini_phase)
[y,phaseout] = mskmod(...)
```

Description

`y = mskmod(x,nsamp)` outputs the complex envelope `y` of the modulation of the message signal `x` using differentially encoded minimum shift keying (MSK) modulation. The elements of `x` must be 0 or 1. `nsamp` denotes the number of samples per symbol in `y` and must be a positive integer. The initial phase of the MSK modulator is 0. If `x` is a matrix with multiple rows and columns, the function treats the columns as independent channels and processes them independently.

`y = mskmod(x,nsamp,`*`dataenc`*`)` specifies the method of encoding data for MSK. *`dataenc`* can be either `'diff'` for differentially encoded MSK or `'nondiff'` for nondifferentially encoded MSK.

`y = mskmod(x,nsamp,`*`dataenc`*`,ini_phase)` specifies the initial phase of the MSK modulator. `ini_phase` is a row vector whose length is the number of channels in `y` and whose values are integer multiples of `pi/2`. To avoid overriding the default value of *`dataenc`*, set *`dataenc`* to `[]`.

`[y,phaseout] = mskmod(...)` returns the final phase of `y`. This is useful for maintaining phase continuity when you are modulating a future bit stream with differentially encoded MSK. `phaseout` has the same dimensions as the `ini_phase` input, and assumes the values `0`, `pi/2`, `pi`, and `3*pi/2`.

Examples

Eye Diagram of MSK Signal

Generate a random binary signal.

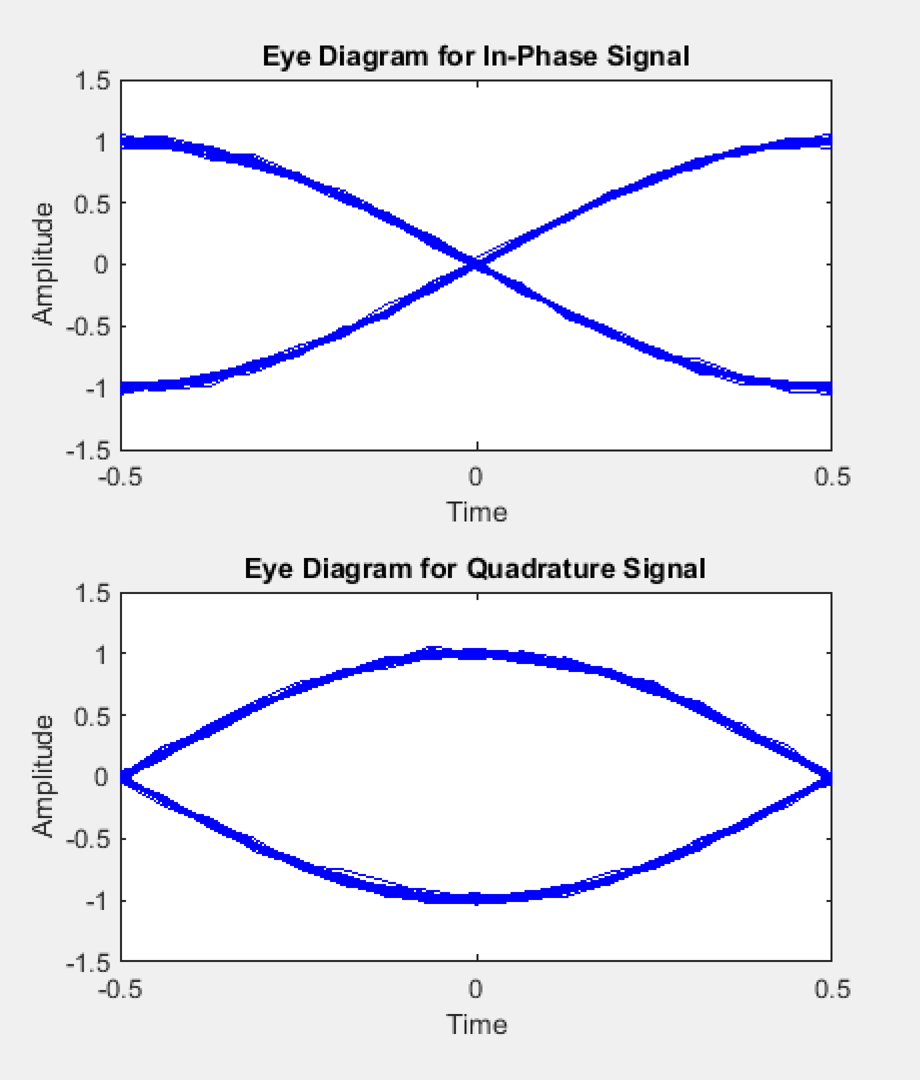
```
x = randi([0 1],100,1);
```

MSK modulate the data.

```
y = mskmod(x,8,[],pi/2);
```

Pass the signal through an AWGN channel. Display the eye diagram.

```
z = awgn(y,30,'measured');
eyediagram(z,16);
```



References

[1] Pasupathy, Subbarayan, “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`comm.MSKModulator` | `fskdemod` | `fskmod` | `mskdemod`

Introduced before R2006a

muxdeintrlv

Restore ordering of symbols using specified shift registers

Syntax

```
deintrlved = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay)
[deintrlved,state] = muxdeintrlv(data,delay,init_state)
```

Description

`deintrlved = muxdeintrlv(data,delay)` restores the ordering of elements in `data` by using a set of internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[deintrlved,state] = muxdeintrlv(data,delay)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[deintrlved,state] = muxdeintrlv(data,delay,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding interleaver.

Using an Interleaver-Deinterleaver Pair

To use this function as an inverse of the `muxintrlv` function, use the same `delay` input in both functions. In that case, the two functions are inverses in the sense that applying `muxintrlv` followed by `muxdeintrlv` leaves data unchanged, after you take their combined delay of `length(delay)*max(delay)` into account. To learn more about delays of convolutional interleavers, see “Delays of Convolutional Interleavers”.

Examples

The example below illustrates how to use the state input and output when invoking `muxdeintrlv` repeatedly. Notice that `[deintrlved1; deintrlved2]` is the same as `deintrlved`.

```
delay = [0 4 8 12]; % Delays in shift registers
symbols = 100; % Number of symbols to process
% Interleave random data.
intrlved = muxintrlv(randi([0 1],symbols,1),delay);

% Deinterleave some of the data, recording state for later use.
[deintrlved1,state] = muxdeintrlv(intrlved(1:symbols/2),delay);
% Deinterleave the rest of the data, using state as an input argument.
deintrlved2 = muxdeintrlv(intrlved(symbols/2+1:symbols),delay,state);

% Deinterleave all data in one step.
deintrlved = muxdeintrlv(intrlved,delay);

isequal(deintrlved,[deintrlved1; deintrlved2])
```

The output is below.

```
ans =

     1
```

Another example using this function is in “Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB”.

References

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

See Also

`muxintrlv`

Topics

“Interleaving”

Introduced before R2006a

muxintrlv

Permute symbols using shift registers with specified delays

Syntax

```
intrlved = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay)
[intrlved,state] = muxintrlv(data,delay,init_state)
```

Description

`intrlved = muxintrlv(data,delay)` permutes the elements in `data` by using internal shift registers, each with its own delay value. `delay` is a vector whose entries indicate how many symbols each shift register can hold. The length of `delay` is the number of shift registers. Before the function begins to process data, it initializes all shift registers with zeros. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

`[intrlved,state] = muxintrlv(data,delay)` returns a structure that holds the final state of the shift registers. `state.value` stores any unshifted symbols. `state.index` is the index of the next register to be shifted.

`[intrlved,state] = muxintrlv(data,delay,init_state)` initializes the shift registers with the symbols contained in `init_state.value` and directs the first input symbol to the shift register referenced by `init_state.index`. The structure `init_state` is typically the `state` output from a previous call to this same function, and is unrelated to the corresponding deinterleaver.

Examples

The examples in “Convolutional Interleaving and Deinterleaving Using a Sequence of Consecutive Integers in MATLAB” and on the reference page for the `convintrlv` function use `muxintrlv`.

The example on the reference page for `muxdeintrlv` illustrates how to use the `state` output and `init_state` input with that function; the process is analogous for this function.

References

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

See Also

`convintrlv` | `helintrlv` | `muxdeintrlv`

Topics

“Interleaving”

Introduced before R2006a

noisebw

Equivalent noise bandwidth of filter

Syntax

```
bw = noisebw(num, den, numsamp, Fs)
```

Description

`bw = noisebw(num, den, numsamp, Fs)` returns the two-sided equivalent noise bandwidth, in Hz, of a digital lowpass filter given in descending powers of z by numerator vector `num` and denominator vector `den`. The bandwidth is calculated over `numsamp` samples of the impulse response. `Fs` is the sampling rate of the signal that the filter would process; this is used as a scaling factor to convert a normalized unitless quantity into a bandwidth in Hz.

Examples

Noise Equivalent Bandwidth of Butterworth Filter

Computes the equivalent noise bandwidth of a Butterworth filter over 100 samples of the impulse response.

Set the sampling rate, Nyquist frequency, and carrier frequency.

```
fs = 16;
fNyq = fs/2;
fc = 0.5;
```

Generate the Butterworth filter.

```
[num,den] = butter(2,fc/fNyq);
```

Determine the noise bandwidth.

```
bw = noisebw(num,den,100,fs)
```

```
bw = 1.1049
```

Algorithms

The two-sided equivalent noise bandwidth is

$$\frac{\mathrm{Fs}\sum_{i=1}^{N}|h(i)|^2}{\left|\sum_{i=1}^{N}h(i)\right|^2}$$

where h is the impulse response of the filter described by `num` and `den`, and N is `numsamp`.

References

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

Introduced before R2006a

normlms

(To be removed) Construct normalized least mean square (LMS) adaptive algorithm object

Note will be removed in a future release. Consider using `comm.LinearEqualizer` or `comm.DecisionFeedback` instead.

Syntax

```
alg = normlms(stepsize)
alg = normlms(stepsize,bias)
```

Description

The `normlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`alg = normlms(stepsize)` constructs an adaptive algorithm object based on the normalized least mean square (LMS) algorithm with a step size of `stepsize` and a bias parameter of zero.

`alg = normlms(stepsize,bias)` sets the bias parameter of the normalized LMS algorithm. `bias` must be between 0 and 1. The algorithm uses the bias parameter to overcome difficulties when the algorithm's input signal is small.

Properties

The table below describes the properties of the normalized LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Fixed value, `'Normalized LMS'`
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.
`Bias`	Normalized LMS bias parameter, a nonnegative real number

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Algorithms

Referring to the schematics presented in “Equalization”, define *w* as the vector of all weights w_i and define *u* as the vector of all inputs u_i. Based on the current set of weights, *w*, this adaptive algorithm creates the new set of weights given by

$$(\text{LeakageFactor})w + \frac{(\text{StepSize})u^*e}{u^Hu + \text{Bias}}$$

where the * operator denotes the complex conjugate and *H* denotes the Hermitian transpose.

Compatibility Considerations

`normlms` will be removed

Warns starting in R2020a

- normlms will be removed in a future release. Consider using `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to LMS.
- The `comm.LinearEqualizer` or `comm.DecisionFeedback` System objects do not have a leakage factor property. This is equivalent to setting `LeakageFactor` to `1` in the normlms function.
- For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-601.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

oct2dec

Convert octal 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

Convert Octal Matrix to Decimal Equivalent

Convert a 2-by-2 octal matrix its decimal equivalent.

```
d = oct2dec([12 144;0 25])
```

```
d = 2×2

    10   100
     0    21
```

The octal number 144 is equivalent to 100 because 144 = $1(8^2) + 4(8^1) + 4(8^0)$ = 100.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`bi2de`

Introduced before R2006a

oct2poly

Convert octal number to binary coefficients

Syntax

```
b = oct2poly(oct)
b = oct2poly(oct,ord)
```

Description

`b = oct2poly(oct)` converts an octal number, `oct`, to a vector of binary coefficients, `b`.

`b = oct2poly(oct,ord)` specifies the power order, `ord`, of the coefficients that comprise the output. If omitted, `ord` is `'descending'`.

Examples

Convert Octal Number to Binary Vector

Convert the octal number `11` to a binary vector.

```
b = oct2poly(11)
```

```
b = 1×4

     1     0     0     1
```

The binary vector corresponds to the polynomial $x^3 + 1$.

Convert Octal Number to Ascending Order Binary Vector

Convert the octal number `65` to an ascending order binary vector.

```
b = oct2poly(65,'ascending')
```

```
b = 1×6

     1     0     1     0     1     1
```

Sixty-five octal is the generator polynomial of a (15,10) Hamming code in the Bluetooth v4.0 standard. The binary representation of 65 octal is 110101 and the GF(2) polynomial is $1 + x^2 + x^4 + x^5$ or [1 0 1 0 1 1] in ascending powers.

Input Arguments

`oct` — Octal number

scalar

Octal number, specified as a positive integer scalar.

Example: `15`

Example: `3177`

Data Types: `double`

`ord` — Power order

`'descending'` (default) | `'ascending'`

Power order of the binary coefficients vector, specified as a character vector having a value of `'ascending'` or `'descending'`.

Data Types: `char`

Output Arguments

`b` — Binary coefficients

vector

Binary coefficients representing a polynomial, returned as a row vector having length equal to $p + 1$, where p is the order of octal input.

See Also

`bi2de` | `de2bi` | `hex2poly` | `oct2dec`

Introduced in R2015b

ofdmdemod

Demodulate time-domain signal using orthogonal frequency division multiplexing (OFDM)

Syntax

```
outSym = ofdmdemod(ofdmSig,nfft,cplen)
outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset)
outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx)
[outSym,pilots] = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx,pilotidx)
```

Description

`outSym = ofdmdemod(ofdmSig,nfft,cplen)` performs OFDM demodulation on the input time domain signal specified in `ofdmSig`, using an FFT size specified by `nfft` and cyclic prefix length specified by `cplen`. For information, see “OFDM Demodulation” on page 2-613.

`outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset)` applies the symbol sampling offset, `symOffset`, for each OFDM symbol before demodulation of the input.

`outSym = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx)` removes null subcarriers from the locations specified in `nullidx`. For this syntax, the symbol sampling offset is applied to each OFDM symbol and the number of rows in the output is `nfft – length(nullidx)`, which accounts for the removal of null subcarriers. Use null subcarriers to account for guard bands and DC subcarriers. For information, see “Subcarrier Allocation and Guard Bands” on page 2-614.

`[outSym,pilots] = ofdmdemod(ofdmSig,nfft,cplen,symOffset,nullidx,pilotidx)` returns pilot subcarriers for the pilot indices specified in `pilotidx`. For this syntax, the symbol sampling offset is applied to each OFDM symbol and number of rows in the output is `nfft – length(nullidx) – length(pilotidx)`, which accounts for the removal of null and pilot subcarriers. The function assumes that pilot subcarrier locations are the same across each OFDM symbol and transmit antenna.

Examples

OFDM Demodulation with Different CP Lengths

OFDM-demodulate a signal with different CP lengths for different symbols.

Initialize input parameters defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft = 64;
cplen = [16 32];
nSym = 2;
dataIn = complex(randn(nfft,nSym),randn(nfft,nSym));
y1 = ofdmmod(dataIn,nfft,cplen);
```

Demodulate the OFDM symbols. Compare the results to the original input data. The difference between the signals is negligible.

```
x1 = ofdmdemod(y1,nfft,cplen);
max(x1-dataIn)
```

```
ans = 1×2 complex
10^-15 ×

   0.2220 - 0.7772i   0.2498 - 0.8882i
```

OFDM Mod-Demod SISO link

Apply OFDM multiplexing to a 16-QAM signal SISO link with Rayleigh fading.

```
s1 = RandStream('mt19937ar','Seed',12345);
nFFT = 64;
cpLen = 16;
nullIdx = [1:6 33 64-4:64].';
numTones = nFFT-length(nullIdx);

k = 4; % bits per symbol
M = 2^k;
constSym = qammod((0:M-1),M,'UnitAveragePower',true);

maxDopp = 1;
pathDelays = [0 4e-3 8e-3];
pathGains = [0 -2 -3];
sRate = 1000;
sampIdx = round(pathDelays/(1/sRate)) + 1;

chan = comm.RayleighChannel('PathGainsOutputPort',true, ...
    'MaximumDopplerShift',maxDopp,'PathDelays',pathDelays, ...
    'AveragePathGains',pathGains,'SampleRate',sRate, ...
    'RandomStream','mt19937ar with seed');

data = randi(s1,[0 M-1],numTones,1);

modOut = qammod(data,M,'UnitAveragePower',true);
```

Apply OFDM modulation and pass the signal through the channel.

```
y = ofdmmod(modOut,nFFT,cpLen,nullIdx);
[fadSig,pg] = chan(y);
```

Determine symbol sampling offset.

```
symOffset = min(max(sampIdx),cpLen)
```

```
symOffset = 9
```

OFDM demodulate the received signal.

```
x = ofdmdemod(fadSig,nFFT,cpLen,symOffset,nullIdx); % with a time shift
```

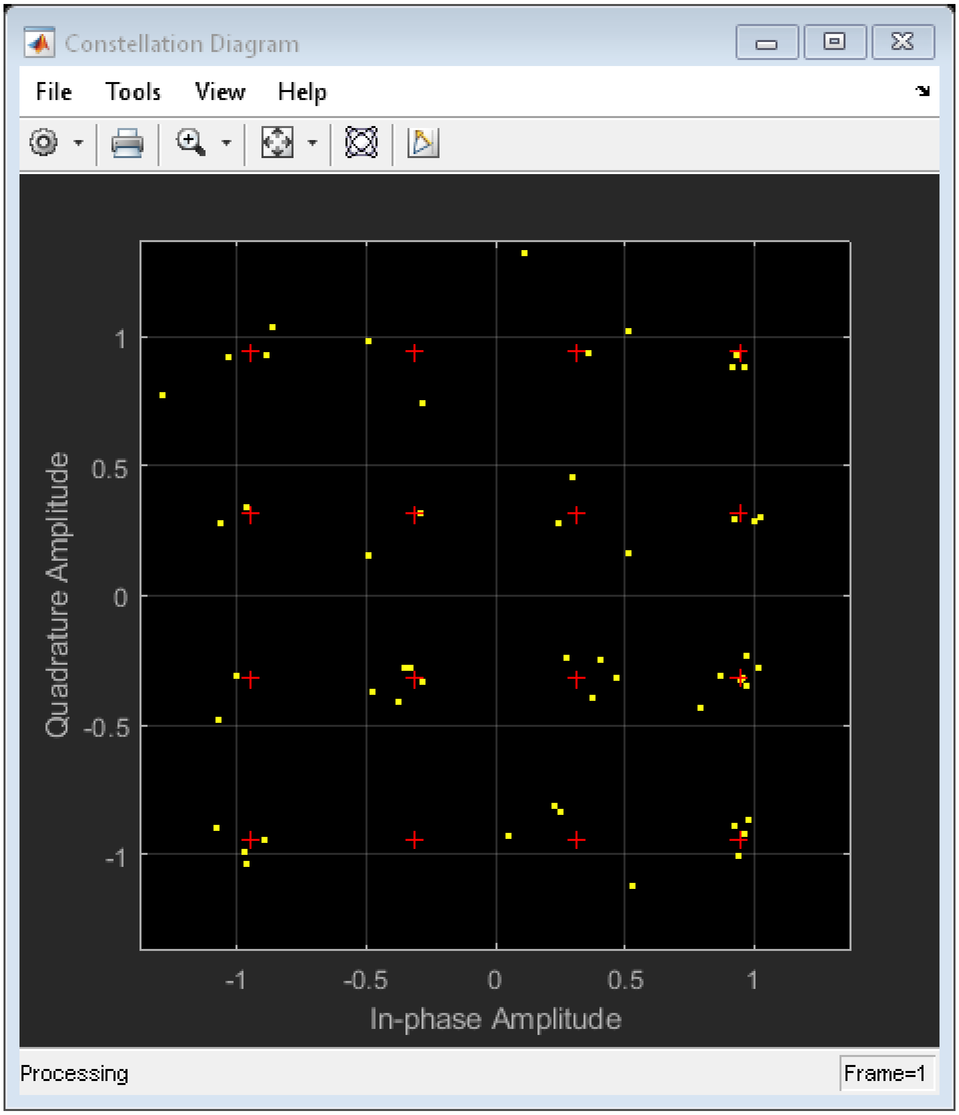
Convert path gains, `pg`, to scalar taps gains. Use the tap gains for equalization during signal recovery.

```
hImp = complex(zeros(1,nFFT));
hImp(:,sampIdx) = mean(pg,1);
```

```
hall = fftshift(fft(hImp.'),1);
dataIdx = double(setdiff((1:nFFT)',nullIdx));
h = hall(dataIdx);
```

Equalize the signal.

```
eqH = conj(h)./(conj(h).*h);
eqSig = eqH.*x;

cdScope = comm.ConstellationDiagram('ShowReferenceConstellation',true, ...
    'ReferenceConstellation',constSym);
cdScope(eqSig);
```



Demodulate the 16-QAM symbols to recover the signal. Compute the bit error rate.

```
rxSym = qamdemod(eqSig,M,'UnitAveragePower',true);

numErr = symerr(data,rxSym);
disp(['Number of symbol errors: ' num2str(numErr) ' out of ' num2str(length(data)) ' symbols.'])
```

```
Number of symbol errors: 2 out of 52 symbols.
```

OFDM Demodulation with Null and Pilot Packing

OFDM-demodulate data input that includes null and pilot packing.

Initialize input parameters, defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft     = 64;
cplen    = 16;
nSym     = 10;
nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';
numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
dataIn = complex(randn(numDataCarrs,nSym),randn(numDataCarrs,nSym));
pilots = repmat(pskmod((0:3).',4),1,nSym);
y2 = ofdmmod(dataIn,nfft,cplen,nullIdx,pilotIdx,pilots);
```

Demodulate the OFDM symbols. Compare the results to the original input data to show that there is negligible difference between the demodulated signal and the original data and pilot signals.

```
symOffset = cplen;
[x2,rxPilots] = ofdmdemod(y2,nfft,cplen,symOffset,nullIdx,pilotIdx);
max(x2-dataIn)
```

```
ans = 1×10 complex
10^-15 ×

   0.5551 + 0.2220i   0.2220 + 0.4441i   0.4441 - 0.2220i   0.4718 - 0.3331i  -0.1665 - 0.4441i
```

```
max(rxPilots-pilots)
```

```
ans = 1×10 complex
10^-15 ×

   0.0000 + 0.3331i   0.1837 - 0.2220i  -0.4441 - 0.2776i   0.2220 + 0.2220i   0.2220 - 0.1665i
```

Input Arguments

ofdmSig — Modulated OFDM symbols
2-D array of complex symbols

Modulated OFDM symbols, specified as a 2-D array of complex symbols.

- If cplen is a scalar, the array size is ((nfft + cplen) × N_{Sym})-by-N_{R}.
- If cplen is a row vector, the array size is ((nfft × N_{Sym}) + sum(cplen))-by-N_{R}.

  N_{Sym} is the number of symbols per antenna and N_{R} is the number of receive antennas.

Data Types: double | single

Complex Number Support: Yes

nfft — FFT length
integer greater than or equal to 8

FFT length, specified as an integer greater than or equal to 8. `nfft` is equivalent to the number of subcarriers used in the demodulation process.

Data Types: `double`

cplen — Cyclic prefix length
scalar | row vector of length N_{Sym}

Cyclic prefix length, specified as a scalar or as a row vector of length N_{Sym}.

- When you specify `cplen` as a scalar, the cyclic prefix length is the same for all symbols through all antennas.
- When you specify `cplen` as a row vector of length N_{Sym}, the cyclic prefix length can vary across symbols but remains the same length through all antennas.

Data Types: `double`

symOffset — Symbol sampling offset
`cplen` (default) | scalar | row vector

Symbol sampling offset, specified as values from 0 to `cplen`.

- If you do not specify `symOffset`, the default value is an offset equal to `cplen`.
- If you specify `symOffset` as a scalar, the same offset is used for all symbols.
- If you specify `symOffset` as a row vector, the offset value can be different for each symbol.

For information, see “Windowing and Symbol Offset” on page 2-615.

Data Types: `double`

nullidx — Indices of null subcarrier locations
column vector

Indices of null subcarrier locations, specified as a column vector with element values from 1 to `nfft`. If you specify `nullidx`, the number of rows in `outSym` is `(nfft-length(nullidx))`. For information, see “Subcarrier Allocation and Guard Bands” on page 2-614.

Data Types: `double`

pilotidx — Indices of pilot subcarrier locations
column vector

Indices of pilot subcarrier locations, specified as a column vector with element values from 1 to `nfft`. If you specify `pilotidx`, the number of rows in `outSym` is `(nfft-length(nullidx)-length(pilotidx))`.For information, see “Subcarrier Allocation and Guard Bands” on page 2-614.

Data Types: `double`

Output Arguments

`outSym` — Output demodulated symbols

3-D array

Output demodulated symbols, returned as an N_D-by-N_{Sym}-by-N_R array of symbols. N_D must equal `nfft – length(nullidx) – length(pilotidx)`. N_{Sym} is the number of OFDM symbols per antenna. N_R is the number of receive antennas. For information, see “OFDM Demodulation” on page 2-613.

`pilots` — Pilot subcarriers

3-D array

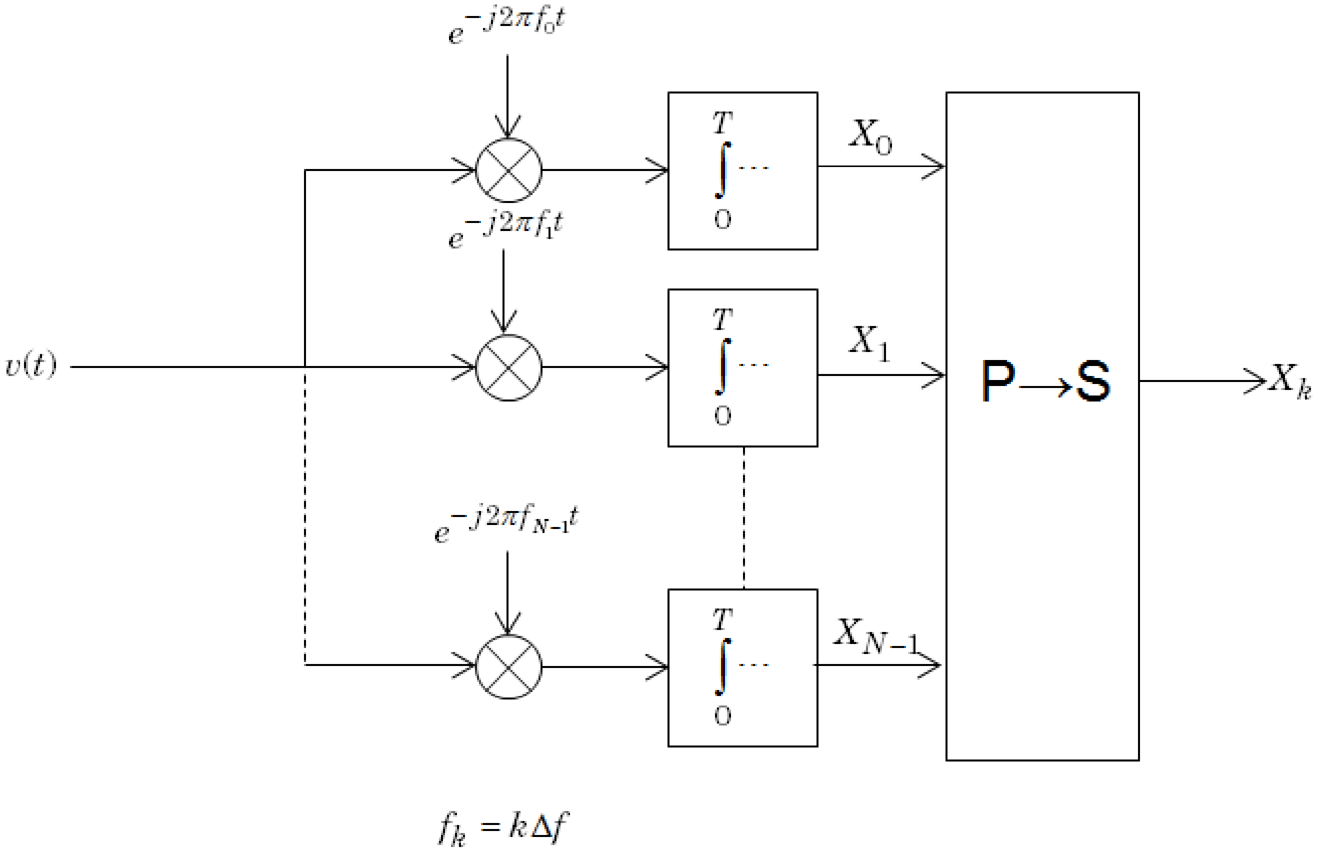
Pilot subcarriers, returned as an N_{Pilot}-by-N_{Sym}-by-N_R array of symbols. N_{Pilot} must equal the length of `pilotidx`. N_{Sym} is the number of OFDM symbols per antenna. N_R is the number of receive antennas. The function assumes that the pilot subcarrier locations are the same across each OFDM symbol and transmit antenna. Use the `comm.OFDMDemodulator` to vary pilot subcarrier locations across OFDM symbols or antennas.

More About

OFDM Demodulation

An OFDM demodulator demultiplexes a multi-subcarrier time-domain signal using orthogonal frequency division modulation.

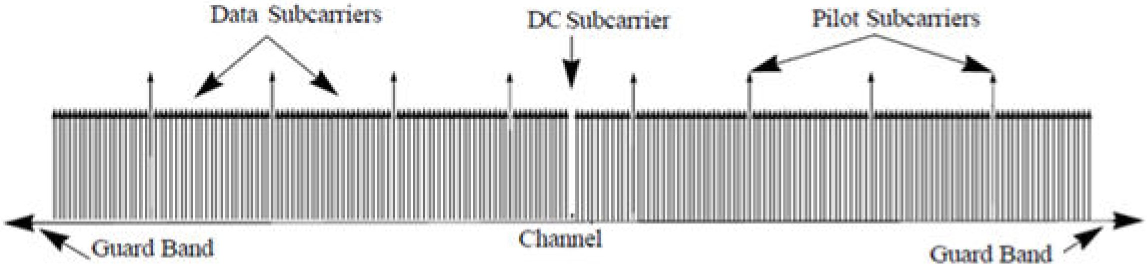
The OFDM demodulation uses an FFT operation that results in N parallel data streams. An OFDM demodulator consists of a bank of N correlators, with one correlator assigned to each OFDM subcarrier, followed by a parallel-to-serial conversion.



Subcarrier Allocation and Guard Bands

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, or guard band subcarriers.



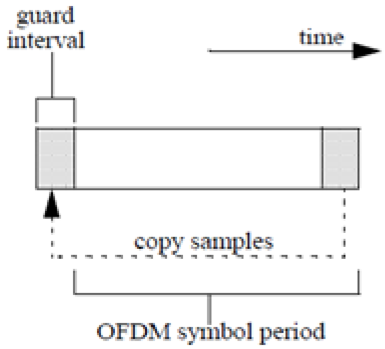
- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.

- The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
- The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

Similar to guard bands, guard intervals are used in OFDM to protect the integrity of transmitted signals by reducing intersymbol interference.

Assignment of guard intervals is analogous to the assignment of guard bands. You can model guard intervals to provide temporal separation between OFDM symbols. The guard intervals help preserve intersymbol orthogonality after the signal passes through time-dispersive channels. Guard intervals are created by using cyclic prefixes. Cyclic prefix insertion copies the last part of an OFDM symbol as the first part of the OFDM symbol.



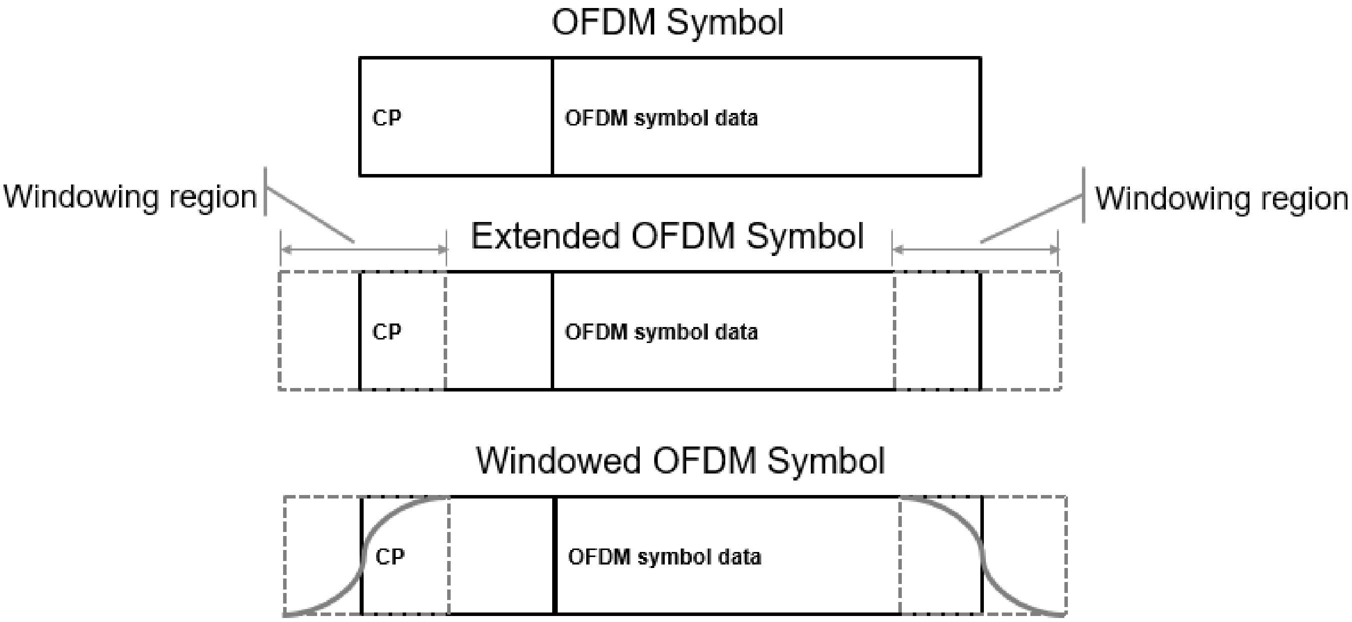
As long as the span of the time dispersion does not exceed the duration of the cyclic prefix, the benefit of cyclic prefix insertion is maintained.

Inserting a cyclic prefix results in a fractional reduction of user data throughput because the cyclic prefix occupies bandwidth that could be used for data transmission.

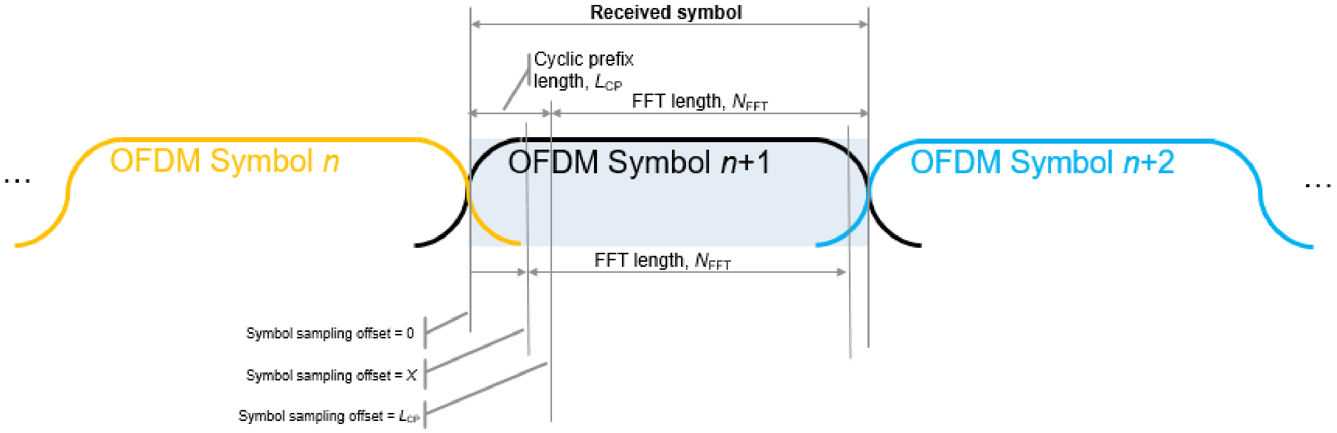
Windowing and Symbol Offset

To reduce intersymbol interference (ISI) introduced by signal windowing applied at the transmitter, the function applies a fractional symbol offset before demodulation of each OFDM symbol. Signal windowing is often applied to transmitted OFDM symbols to smooth the discontinuity between consecutive OFDM symbols. Windowing reduces intersymbol out-of-band emissions but increases ISI.

The windowed OFDM symbol consists of the cyclic prefix (CP), ODFM symbol data, plus windowing regions at the beginning and end of the symbol. The leading and trailing windowing shoulders have tails as shown in the figure.



To reduce ISI, you can align signal sample timing by specifying a symbol sampling offset that gets applied before OFDM symbol demodulation.


Received symbol
Cyclic prefix length, L_{CP}
FFT length, N_{FFT}
...
OFDM Symbol n
OFDM Symbol n+1
OFDM Symbol n+2
...
FFT length, N_{FFT}
Symbol sampling offset = 0
Symbol sampling offset = X
Symbol sampling offset = L_{CP}

Specify the symbol sampling offset as a value from 0 to L_{CP}.

- When the symbol sampling offset is a scalar from 0 to L_{CP}, the FFT window begins at the X+1 sample of the CP length.
- When the symbol sampling offset is zero, no offset is applied and the FFT window starts at the first sample of the symbol.
- When the symbol sampling offset is the cyclic prefix length, L_{CP}, the FFT window begins after the last CP sample. This offset is the default setting if symbol sampling offset is not specified.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

genqamdemod | ofdmmod | qamdemod

Objects

comm.GeneralQAMDemodulator | comm.OFDMDemodulator | comm.OQPSKDemodulator

Introduced in R2018a

ofdmmod

Modulate frequency-domain signal using orthogonal frequency division multiplexing (OFDM)

Syntax

```
ofdmSig = ofdmmod(inSym,nfft,cplen)
ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx)
ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx,pilotidx,pilots)
```

Description

`ofdmSig = ofdmmod(inSym,nfft,cplen)` performs OFDM modulation on the frequency-domain input data subcarriers, `inSym`, using an FFT size specified by `nfft` and cyclic prefix length specified by `cplen`. For information, see "OFDM Modulation" on page 2-622.

`ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx)` inserts null subcarriers into the frequency domain input data signal prior to performing OFDM modulation. The null subcarriers are inserted at index locations from 1 to `nfft`, as specified by `nullidx`. For this syntax, the number of rows in the input `inSym` must be `nfft` – `length(nullidx)`. Use null carriers to account for guard bands and DC subcarriers. For information, see "Subcarrier Allocation, Guard Bands and Guard Intervals" on page 2-623.

`ofdmSig = ofdmmod(inSym,nfft,cplen,nullidx,pilotidx,pilots)` inserts null and pilot subcarriers into the frequency domain input data symbols prior to performing OFDM modulation. The null subcarriers are inserted at the index locations specified by `nullidx`. The pilot subcarriers, `pilots`, are inserted at the index locations specified by `pilotidx`. For this syntax, the number of rows in the input `inSym` must be `nfft` – `length(nullidx)` – `length(pilotidx)`. The function assumes pilot subcarrier locations are the same across each OFDM symbol and transmit antenna.

Examples

OFDM Modulation Over Two Antennas

OFDM-modulate a fully packed input over two transmit antennas.

Initialize input parameters, generate random data, and perform OFDM modulation.

```
nfft  = 128;
cplen = 16;
nSym  = 5;
nt    = 2;
dataIn = complex(randn(nfft,nSym,nt),randn(nfft,nSym,nt));

y1 = ofdmmod(dataIn,nfft,cplen);
```

Apply OFDM Assigning Null Subcarriers

Apply OFDM modulation assigning null subcarriers.

Initialize input parameters and generate random data.

```
M = 16; % Modulation order for 16QAM
nfft  = 64;
cplen = 16;
nSym  = 10;
nullIdx  = [1:6 33 64-4:64]';
numDataCarrs = nfft-length(nullIdx);
inSig = randi([0 M-1],numDataCarrs,nSym);
```

QAM modulate data. Perform OFDM modulation.

```
qamSym = qammod(inSig,M,'UnitAveragePower',true);
outSig = ofdmmod(qamSym,nfft,cplen,nullIdx);
```

Perform OFDM Modulation Varying Cyclic Prefix per Symbol

Perform OFDM modulation to input frequency domain data signal varying cyclic prefix length applied to each symbol.

Initialize input parameters and generate random data.

```
M = 16; % Modulation order for 16QAM
nfft  = 64;
cplen = [4 8 10 7 2 2 4 11 16 3];
nSym  = 10;
nullIdx  = [1:6 33 64-4:64]';
numDataCarrs = nfft-length(nullIdx);
inSig = randi([0 M-1],numDataCarrs,nSym);
```

QAM modulate data. Perform OFDM modulation.

```
qamSym = qammod(inSig,M,'UnitAveragePower',true);
outSig = ofdmmod(qamSym,nfft,cplen,nullIdx);
```

Apply OFDM to QPSK Signal Spatially Multiplexed Over Two Antennas

Apply OFDM modulation to a QPSK signal that is spatially multiplexed over two transmit antennas.

Initialize input parameters and generate random data for each antenna.

```
M = 4; % Modulation order for QPSK
nfft  = 64;
cplen = 16;
nSym  = 5;
nt    = 2;
nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';
numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
```

```
pilots = repmat(pskmod((0:M-1).',M),1,nSym,2);

ant1 = randi([0 M-1],numDataCarrs,nSym);
ant2 = randi([0 M-1],numDataCarrs,nSym);
```

QPSK modulate data individually for each antenna. Perform OFDM modulation.

```
qpskSym(:,:,1) = pskmod(ant1,M);
qpskSym(:,:,2) = pskmod(ant2,M);
y1 = ofdmmod(qpskSym,nfft,cplen,nullIdx,pilotIdx,pilots);
```

OFDM Modulation with Null and Pilot Packing

OFDM-modulate data input, specifying null and pilot packing.

Initialize input parameters, defining locations for null and pilot subcarriers. Generate random data and perform OFDM modulation.

```
nfft     = 64;
cplen    = 16;
nSym     = 10;

nullIdx  = [1:6 33 64-4:64]';
pilotIdx = [12 26 40 54]';

numDataCarrs = nfft-length(nullIdx)-length(pilotIdx);
dataIn = complex(randn(numDataCarrs,nSym),randn(numDataCarrs,nSym));
pilots = repmat(pskmod((0:3).',4),1,nSym);

y2 = ofdmmod(dataIn,nfft,cplen,nullIdx,pilotIdx,pilots);
```

Input Arguments

`inSym` — Input data subcarriers

3-D array

Input data subcarriers, specified as an N_D-by-N_{Sym}-by-N_T array of symbols. The number of data subcarriers, N_D, must equal `nfft – length(nullidx) – length(pilotidx)`. N_{Sym} is the number of OFDM symbols per transmit antenna, N_T is the number of transmit antennas.

Input data symbols to an OFDM modulator are typically created with a baseband digital modulator, such as `qammod`.

Data Types: `double` | `single`
Complex Number Support: Yes

`nfft` — FFT length

integer greater than or equal to 8

FFT length, specified as an integer greater than or equal to 8. `nfft` is equivalent to the number of subcarriers used in the modulation process.

Data Types: `double`

cplen — Cyclic prefix length

scalar | row vector of length N_{Sym}

Cyclic prefix length, specified as a scalar or as a row vector of length N_{Sym}.

- When you specify `cplen` as a scalar, the cyclic prefix length is the same for all symbols through all antennas.
- When you specify `cplen` as a row vector of length N_{Sym}, the cyclic prefix length can vary across symbols but remains the same length through all antennas.

For more information, see “Subcarrier Allocation, Guard Bands and Guard Intervals” on page 2-623.

Data Types: `double`

nullidx — Indices of null subcarrier locations

column vector

Indices of null subcarrier locations, specified as a column vector with element values from 1 to `nfft`.

Data Types: `double`

pilotidx — Indices of pilot subcarrier locations

column vector

Indices of pilot subcarrier locations, specified as a column vector with element values from 1 to `nfft`.

Data Types: `double`

pilots — Pilot subcarriers

3-D array

Pilot subcarriers, specified as an N_{Pilot}-by-N_{Sym}-by-N_{T} array of symbols. N_{Pilot} must equal the length of `pilotidx`. N_{Sym} is the number of OFDM symbols per transmit antenna. N_{T} is the number of transmit antennas. The function assumes pilot subcarrier locations are the same across each OFDM symbol and transmit antenna. Use the `comm.OFDMModulator` to vary pilot subcarrier locations across OFDM symbols or antennas.

Data Types: `double` | `single`

Output Arguments

ofdmSig — Modulated OFDM symbols

2-D array of complex symbols

Modulated OFDM symbols, returned as a 2-D array of complex symbols.

- If `cplen` is a scalar, the array size is ((`nfft` + `cplen`) × N_{Sym})-by-N_{T}.
- If `cplen` is a row vector, the array size is ((`nfft` × N_{Sym}) + `sum(cplen)`)-by-N_{T}.

  N_{Sym} is the number of symbols per transmit antenna and N_{T} is the number of transmit antennas.

Data Types: `double` | `single`

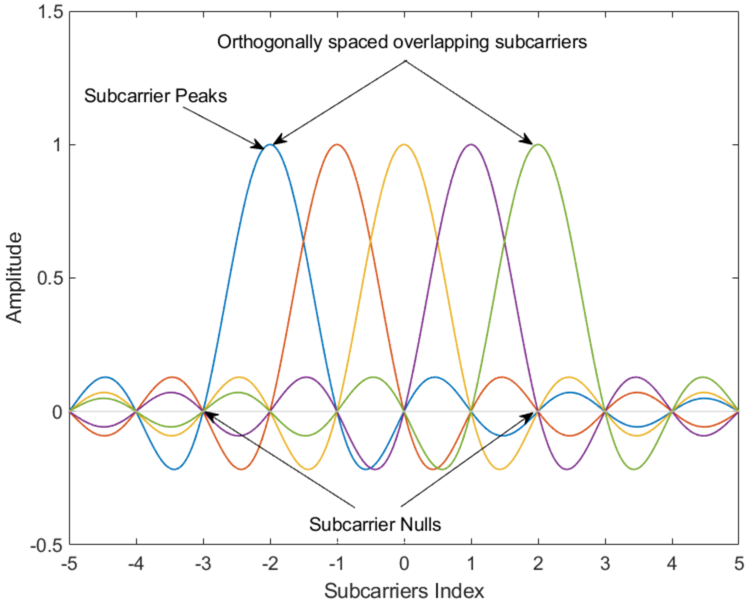
More About

OFDM Modulation

OFDM belongs to the class of multicarrier modulation schemes. Since the multiple data streams can be transmitted simultaneously with multiple carriers, OFDM is not influenced by noise to the same degree as single-carrier modulation.

OFDM operation divides a high-rate data stream into lower data rate substreams by decomposing the transmission frequency band into N contiguous individually modulated subcarriers. Multiple parallel and orthogonal subcarriers carry the samples with almost the same bandwidth as a wideband channel. By using narrow orthogonal subcarriers, the OFDM signal gains robustness over a frequency-selective fading channel and eliminates adjacent subcarrier interference. Intersymbol interference (ISI) is reduced because the lower data rate substreams have symbol durations larger than the channel delay spread.

The Frequency domain representation of orthogonal subcarriers in an OFDM waveform looks as follows:



The transmitter applies inverse fast Fourier transform (IFFT) to N symbols at a time. The output of the IFFT is the sum of the N orthogonal sinusoids:

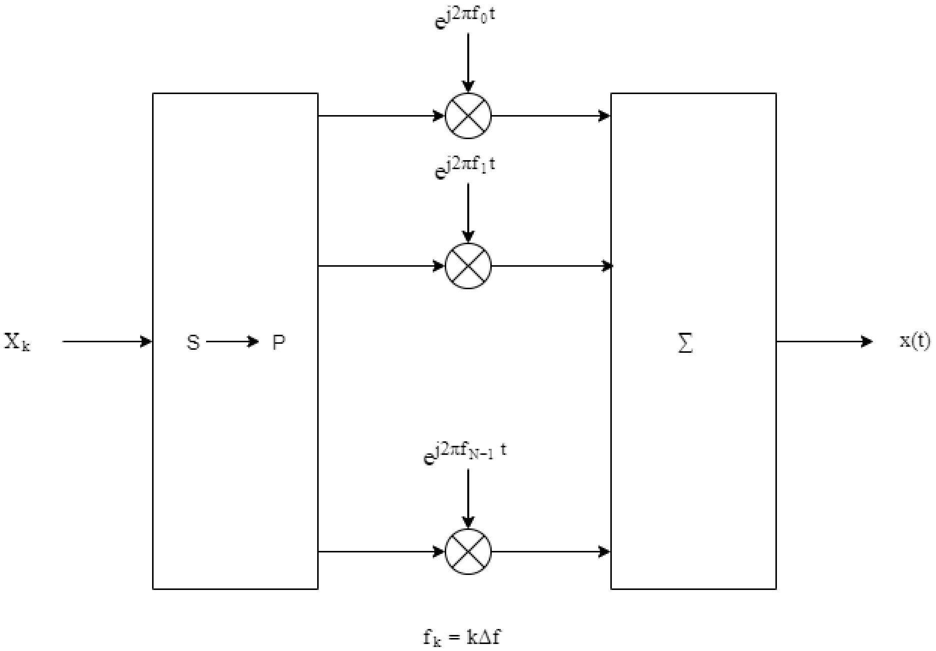
$$x(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi k\Delta f t}, \quad 0 \le t \le T,$$

where $\{X_k\}$ are data symbols, and T is the OFDM symbol time. The data symbols X_k are typically complex and can be from any digital modulation alphabet (for example, QPSK, 16-QAM, 64-QAM).

The subcarrier spacing is $\Delta f = 1/T$; ensuring that the subcarriers are orthogonal over each symbol period, as shown below:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi m\Delta ft}\right)^* \left(e^{j2\pi n\Delta ft}\right) dt = \frac{1}{T}\int_0^T e^{j2\pi(m-n)\Delta ft} dt = 0 \quad \text{for } m \neq n\,.$$

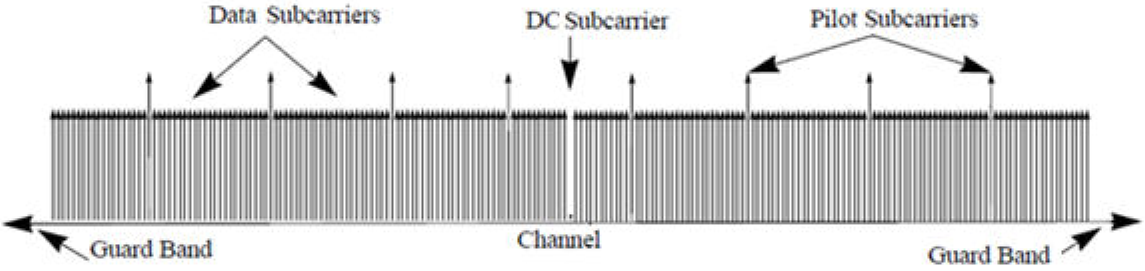
An OFDM modulator consists of a serial-to-parallel conversion followed by a bank of *N* complex modulators, individually corresponding to each OFDM subcarrier.



Subcarrier Allocation, Guard Bands and Guard Intervals

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, or guard band subcarriers.



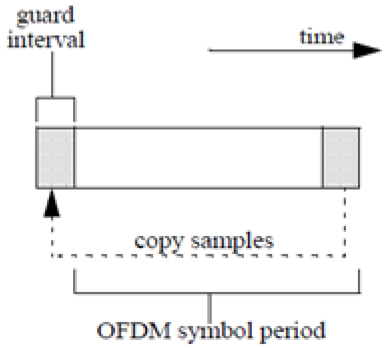
- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.

- The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
- The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

Similar to guard bands, guard intervals are used in OFDM to protect the integrity of transmitted signals by reducing intersymbol interference.

Assignment of guard intervals is analogous to the assignment of guard bands. You can model guard intervals to provide temporal separation between OFDM symbols. The guard intervals help preserve intersymbol orthogonality after the signal passes through time-dispersive channels. Guard intervals are created by using cyclic prefixes. Cyclic prefix insertion copies the last part of an OFDM symbol as the first part of the OFDM symbol.



As long as the span of the time dispersion does not exceed the duration of the cyclic prefix, the benefit of cyclic prefix insertion is maintained.

Inserting a cyclic prefix results in a fractional reduction of user data throughput because the cyclic prefix occupies bandwidth that could be used for data transmission.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`genqammod` | `ofdmdemod` | `qammod`

Objects
`comm.OFDMDemodulator` | `comm.OFDMModulator`

Introduced in R2018a

pamdemod

Pulse amplitude demodulation

Syntax

```
z = pamdemod(y,M)
z = pamdemod(y,M,ini_phase)
z = pamdemod(y,M,ini_phase,symbol_order)
```

Description

`z = pamdemod(y,M)` demodulates the complex envelope `y` of a pulse amplitude modulated signal. `M` is the alphabet size. The ideal modulated signal should have a minimum Euclidean distance of 2.

`z = pamdemod(y,M,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = pamdemod(y,M,ini_phase,symbol_order)` specifies how the function assigns binary words to corresponding integers. If `symbol_order` is set to `'bin'` (default), the function uses a natural binary-coded ordering. If `symbol_order` is set to `'gray'`, it uses a Gray-coded ordering.

Examples

Demodulate PAM Signal

Modulate and demodulate random integers using pulse amplitude modulation. Verify that the output data matches the original data.

Set the modulation order and generate 100 M-ary data symbols.

```
M = 12;
dataIn = randi([0 M-1],100,1);
```

Perform modulation and demodulation operations.

```
modData = pammod(dataIn,M);
dataOut = pamdemod(modData,M);
```

Compare the first five symbols.

```
[dataIn(1:5) dataOut(1:5)]
```

```
ans = 5×2

     9     9
    10    10
     1     1
    10    10
     7     7
```

Verify that there are no symbol errors in the entire sequence.

```
symErrors = symerr(dataIn,dataOut)

symErrors = 0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`pammod` | `pskdemod` | `pskmod` | `qamdemod` | `qammod`

Topics

"Digital Modulation"
"Comparing Theoretical and Empirical Error Rates"

Introduced before R2006a

pammod

Pulse amplitude modulation (PAM)

Syntax

```
y = pammod(x,M)
y = pammod(x,M,initphase)
y = pammod(x,M,initphase,symorder)
```

Description

`y = pammod(x,M)` returns the complex envelope of the modulation of the input message signal, `x`, using PAM and the alphabet size, `M`.

`y = pammod(x,M,initphase)` specifies the initial phase of the modulated signal.

`y = pammod(x,M,initphase,symorder)` specifies the symbol order modulation, which defines how the function assigns binary words to corresponding integers.

Examples

Modulate Data Symbols with PAM

Generate random data symbols and apply pulse amplitude modulation.

Set the modulation order.

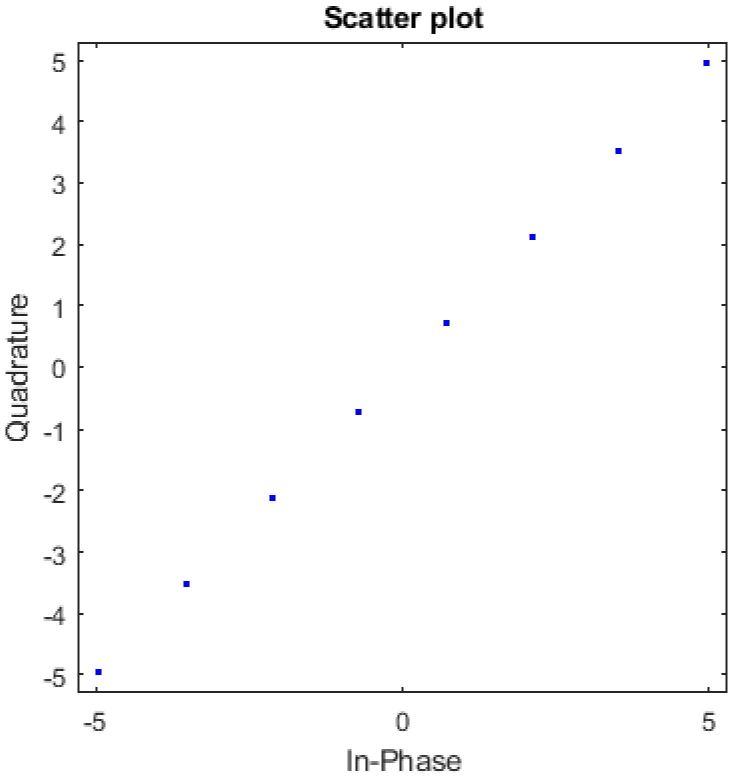
```
M = 8;
```

Generate random integers and apply PAM modulation having an initial phase of $\pi/4$.

```
data = randi([0 M-1],100,1);
modData = pammod(data,M,pi/4);
```

Display the PAM constellation diagram.

```
scatterplot(modData)
```



PAM Symbol Mapping

Plot PAM symbol mapping for Gray and natural binary encoded data.

Set the modulation order, and then create a data sequence containing a complete set of constellation points.

```
M = 8;
data = [0:M-1];
```

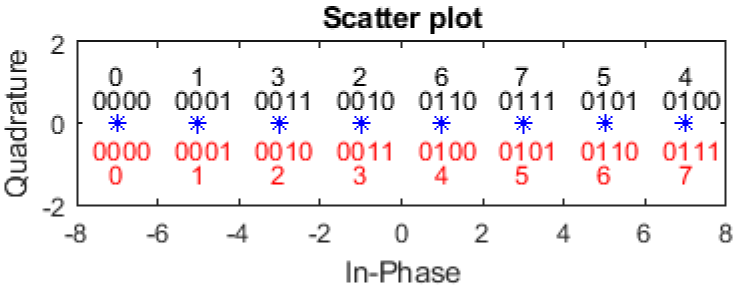
Modulate and demodulate Gray and natural binary encoded data.

```
symgray = pammod(data,M,0,'gray');
mapgray = pamdemod(symgray,M,0,'gray');

symbin = pammod(data,M,0,'bin');
mapbin = pamdemod(symbin,M,0,'bin');
```

Plot the constellation points using one of the symbol sets. For each constellation point, assign a label indicating the Gray and natural binary values for each symbol.

- For Gray binary symbol mapping, adjacent constellation points differ by a single binary bit and are not numerically sequential.
- For natural binary symbol mapping, adjacent constellation points follow the natural binary encoding and are sequential.

```
scatterplot(symgray,1,0,'b*');
for k = 1:M
    text(real(symgray(k))-0.6,imag(symgray(k))+0.6,...
        dec2base(mapgray(k),2,4));
    text(real(symgray(k))-0.2,imag(symgray(k))+1.2,...
        num2str(mapgray(k)));

    text(real(symbin(k))-0.6,imag(symbin(k))-0.6,...
        dec2base(mapbin(k),2,4),'Color',[1 0 0]);
    text(real(symbin(k))-0.2,imag(symbin(k))-1.2,...
        num2str(mapbin(k)),'Color',[1 0 0]);
end
axis([-M M -2 2])
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of integers in the range of [0, M – 1].

Example: randi([0 3],100,1)

Data Types: double

M — Modulation order

power of two

Modulation order, specified as a power of two.

Example: 4

Data Types: double

initphase — Initial phase

0 (default) | real-valued scalar | []

Initial phase of the modulated signal (in radians), specified as a real scalar.

Example: pi/4

Data Types: double

symorder — Symbol order binary vectors

'bin' (default) | 'gray'

Symbol order of binary vectors, specified as 'bin' or 'gray'. This argument specifies how the function assigns binary vectors to corresponding integers.

- If symorder is 'bin', the function uses a natural binary-coded ordering.
- If symorder is 'gray', the function uses a Gray-coded ordering.

Data Types: char | string

Output Arguments

y — Complex baseband representation of PAM-modulated signal

vector | matrix

Complex baseband representation of a PAM-modulated signal, returned as vector or matrix of complex values. The modulated signal has a minimum Euclidean distance of 2. The columns of y represent independent channels.

Data Types: double | single

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “Code Generation for Complex Data with Zero-Valued Imaginary Parts” (MATLAB Coder).

See Also

pamdemod | pskdemod | pskmod | qamdemod | qammod

Topics

“Digital Modulation”
“Comparing Theoretical and Empirical Error Rates”

Introduced before R2006a

channelDelay

Channel timing delay

Syntax

```
[delay,mag] = channelDelay(pathGains,pathFilters)
```

Description

`[delay,mag] = channelDelay(pathGains,pathFilters)` computes the channel timing delay by finding the peak of the channel impulse response. The function reconstructs the impulse response from a channel path gains array and a path filter impulse response matrix. The function returns the channel timing delay in samples, and the channel impulse response magnitude. For more information, see "Channel Delay and Magnitude Computation" on page 2-635.

Examples

Compute Timing Delay for 2-by-2 MIMO Channel

Configure a 2-by-2 MIMO channel. Use the `info` object function to retrieve the path filters.

```
chan = comm.MIMOChannel('SampleRate',1000,'PathDelays',[0 1.5e-3], ...
    'AveragePathGains',[1 0.8],'RandomStream','mt19937ar with seed', ...
    'Seed',10,'PathGainsOutputPort',true);
chanInfo = info(chan);
pathFilters = chanInfo.ChannelFilterCoefficients;
```

Compute the path gains by passing an impulse through the channel.

```
[~,pathGains] = chan(ones(1,2));
```

Compute the channel timing delay, specifying the retrieved path filters and computed path gains.

```
delay = channelDelay(pathGains,pathFilters)
```

```
delay = 6
```

Show Relative Timing Delay For Rayleigh Channel

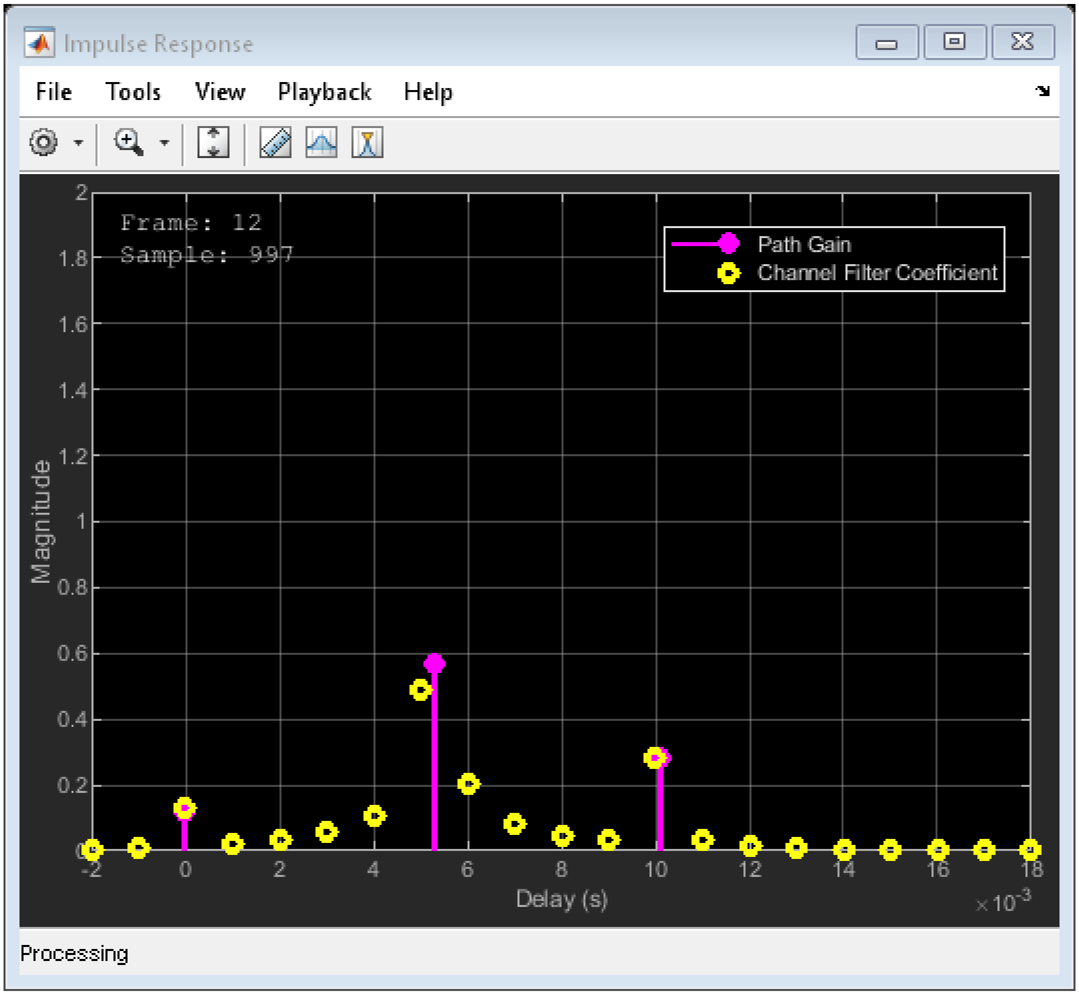
Compute and show the relative timing delay for a Rayleigh channel over time.

Create a `comm.RayleighChannel` System object configured with three paths and impulse response visualization enabled.

```
chan = comm.RayleighChannel;
chan.SampleRate = 1e3;
chan.PathDelays = [0 5.3e-3 10.1e-3];
chan.AveragePathGains = [0.1 1 0.5];
chan.PathGainsOutputPort = true;
chan.RandomStream = 'mt19937ar with seed';
chan.Seed = 1;
chan.Visualization = 'Impulse response';
chan.MaximumDopplerShift = 1;
```

Use the `info` object function to retrieve the Rayleigh channel path filters. In a loop, pass a static signal of all ones through the Rayleigh channel. The `channelDelay` function uses the channel path gains array from each pass through the channel and the path filter coefficients, `chanInfo.ChannelFilterCoefficients` (returned by the `info` function) to compute the relative channel timing delay. The impulse response varies for each iteration. The impulse response for the last iteration is shown here. The `delay` vector shows the relative channel timing delay computed for each iteration.

```
chanInfo = info(chan);
numIter = 12;
delay = zeros(1,numIter);
for p=1:numIter
    [~,pg] = chan(ones(1e3,1));
    delay(p) = channelDelay(pg,chanInfo.ChannelFilterCoefficients);
end
```



```
delay
```

```
delay = 1×12

    12     7    12     2    12     7    12     7     7     7     2     2
```

Input Arguments

`pathGains` — Channel path gains

4-D array

Channel path gains, specified as an N_{cs}-by-N_p-by-N_t-by-N_r array, where:

- N_{cs} is the number of channel snapshots.
- N_p is the number of paths.
- N_t is the number of transmit antennas.
- N_r is the number of receive antennas.

If any element in `pathGains` is `NaN`, the function assumes that no path exists between the transmitter and the receiver.

Data Types: `double` | `single`
Complex Number Support: Yes

`pathFilters` — Path filter impulse response

matrix

Path filter impulse response, specified as an N_p-by-N_h matrix. N_p is the number of paths, and N_h is the number of impulse response samples.

Data Types: `double` | `single`
Complex Number Support: Yes

Output Arguments

`delay` — Channel timing delay

integer

Channel timing delay in samples, returned as an integer. This value represents the number of samples of delay relative to the first sample of the channel impulse response reconstructed from the `pathGains` and `pathFilters` inputs. The function computes the channel timing delay by finding the peak of the composite channel impulse response. For more information, see “Channel Delay and Magnitude Computation” on page 2-635.

`mag` — Channel impulse response magnitude

matrix

Channel impulse response magnitude for each receive antenna, returned as an N_h-by-N_r matrix. N_h is the number of impulse response samples, and N_r is the number of receive antennas. For more information, see “Channel Delay and Magnitude Computation” on page 2-635.

More About

Channel Delay and Magnitude Computation

The computation of the channel delay and impulse response magnitudes uses the composite channel impulse response.

The composite channel impulse response results from averaging the impulse response across all channel snapshots as represented in the path gains array. The input path gains array must be of the format N_{cs}-by-N_{p}-by-N_{t}-by-N_{r} , where:

- N_{cs} is the number of channel snapshots.
- N_{p} is the number of paths.
- N_{t} is the number of transmit antennas.
- N_{r} is the number of receive antennas.

The channel timing delay, output as a single value, is relative to the first sample of the channel impulse response. The function computes this value by finding the peak of the composite channel impulse response. The composite channel impulse response is the summation of the impulse responses across all transmit and receive antennas.

The receive impulse response magnitudes are output as an N_{h}-by-N_{r} matrix. N_{h} is the number of impulse response samples, and N_{r} is the number of receive antennas. To compute the receive impulse response magnitudes,

1. Path gains are summed across all channel snapshots.
2. The contribution from each path is added to the channel impulse response across all transmit and receive antennas.
3. The transmit antenna paths are combined in the channel impulse response array, leaving a matrix of impulse response samples versus receive antennas.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2020a

plot (channel)

(To be removed) Plot channel characteristics with channel visualization tool

Note This function will be removed in a future release. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

Syntax

```
plot(h)
```

Description

`plot(h)`, where `h` is a channel object, launches the channel visualization tool. This GUI tool allows you to plot channel characteristics in various ways. See Channel Visualization for details.

ExamplesVisualize RF Impairments

Apply various RF impairments to a QAM signal. Observe the effects by using constellation diagrams, time-varying error vector magnitude (EVM) plots, and spectrum plots. Estimate the equivalent signal-to-noise ratio (SNR).

Initialization

Set the sample rate, modulation order, and SNR. Calculate the reference constellation points.

```
fs = 1000;
M = 16;
snrdB = 30;
refConst = qammod(0:M-1,M,'UnitAveragePower',true);
```

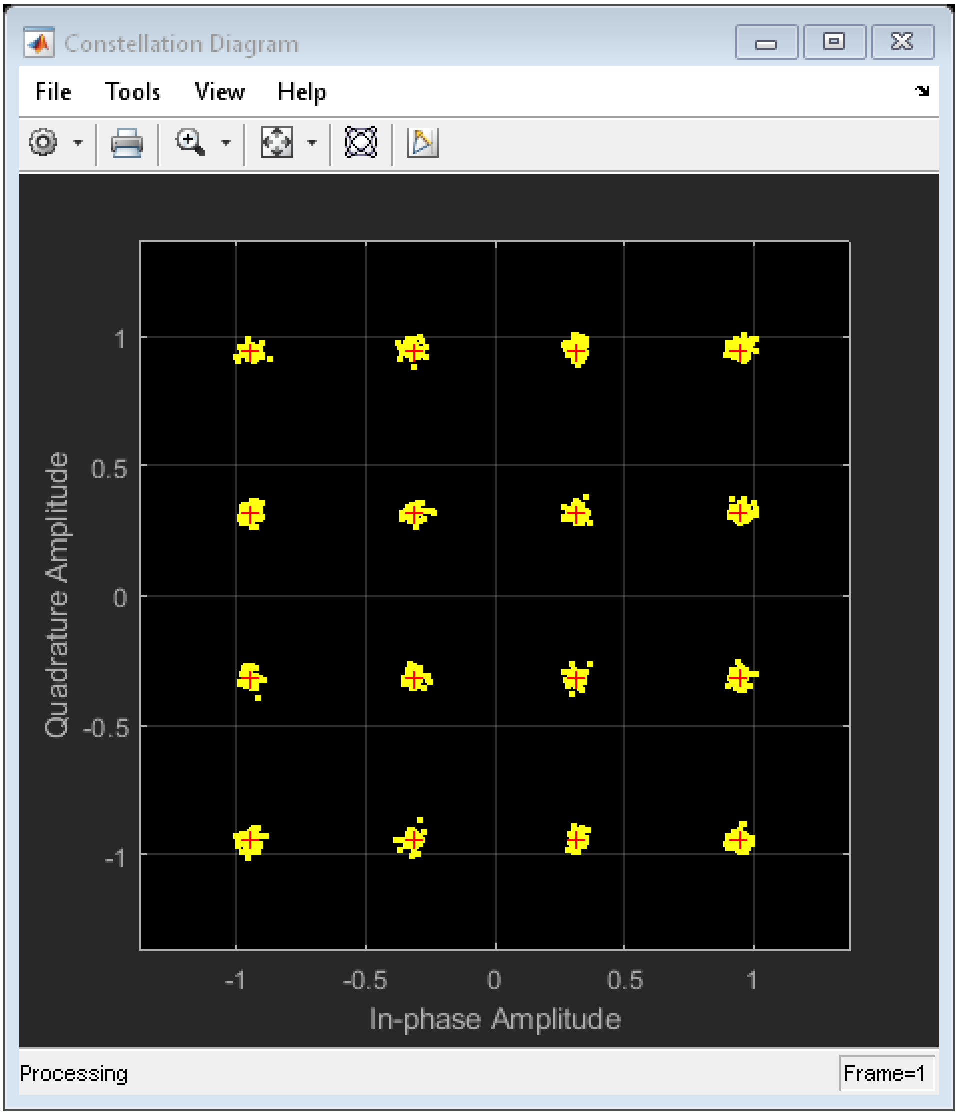
Create constellation diagram and time scope objects to visualize the impairment effects.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = timescope('YLimits',[0 40],'SampleRate',fs,'TimeSpanSource','property','TimeSpan',1,
    'ShowGrid',true,'YLabel','EVM (%)');
```

White Noise

Generate a 16-QAM signal, and pass it through an AWGN channel. Plot its constellation.

```
data = randi([0 M-1],1000,1);
modSig = qammod(data,M,'UnitAveragePower',true);
noisySig = awgn(modSig,snrdB);

constDiagram(noisySig)
```



Estimate the EVM of the noisy signal from the reference constellation points.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power');

rmsEVM = evm(noisySig)

rmsEVM = 3.1761
```

The modulation error rate (MER) closely corresponds to the SNR. Create an MER object, and estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(noisySig)

snrEst = 30.1057
```

The estimate is quite close to the specified SNR of 30 dB.

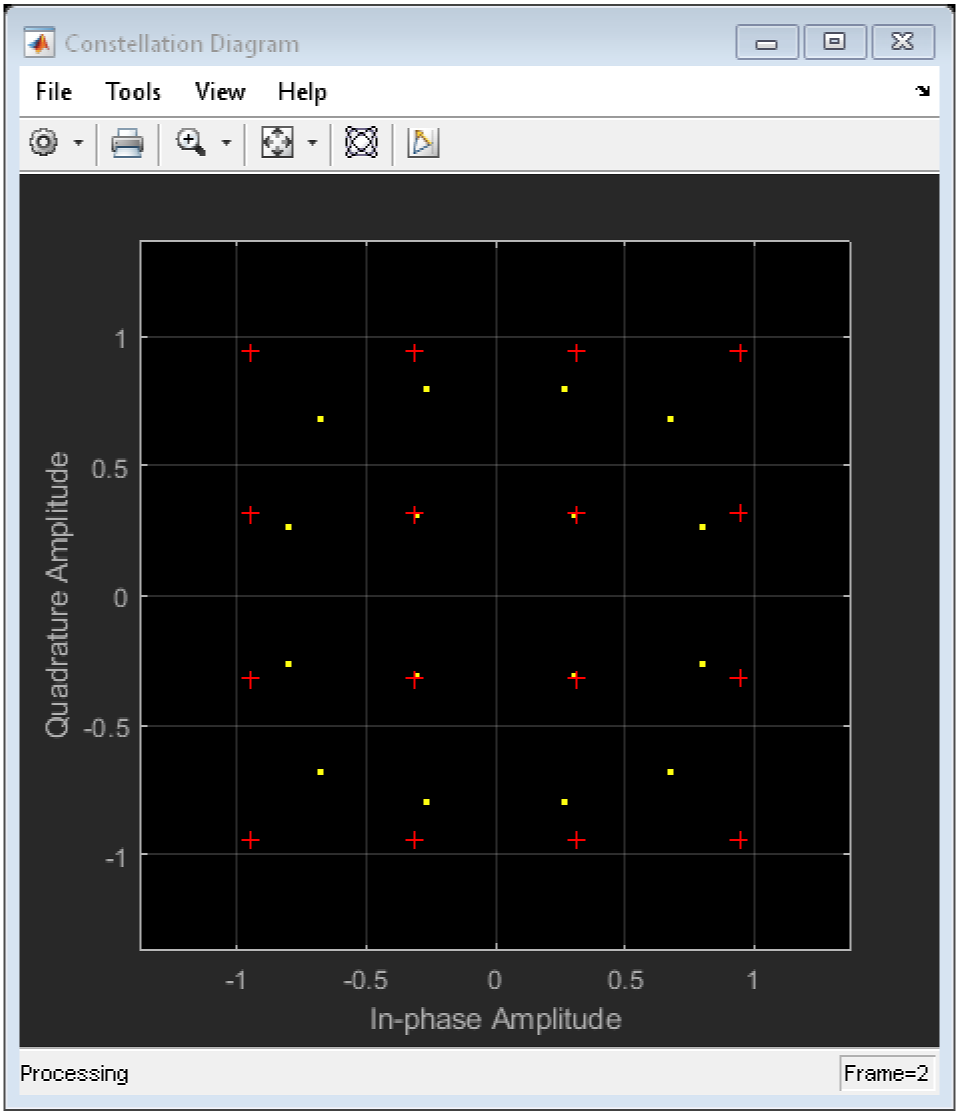
Amplifier Distortion

Create an amplifier using the memoryless nonlinearity object.

```
amp = comm.MemorylessNonlinearity('IIP3',38,'AMPMConversion',0);
```

Pass the modulated signal through the nonlinear amplifier and plot its constellation diagram.

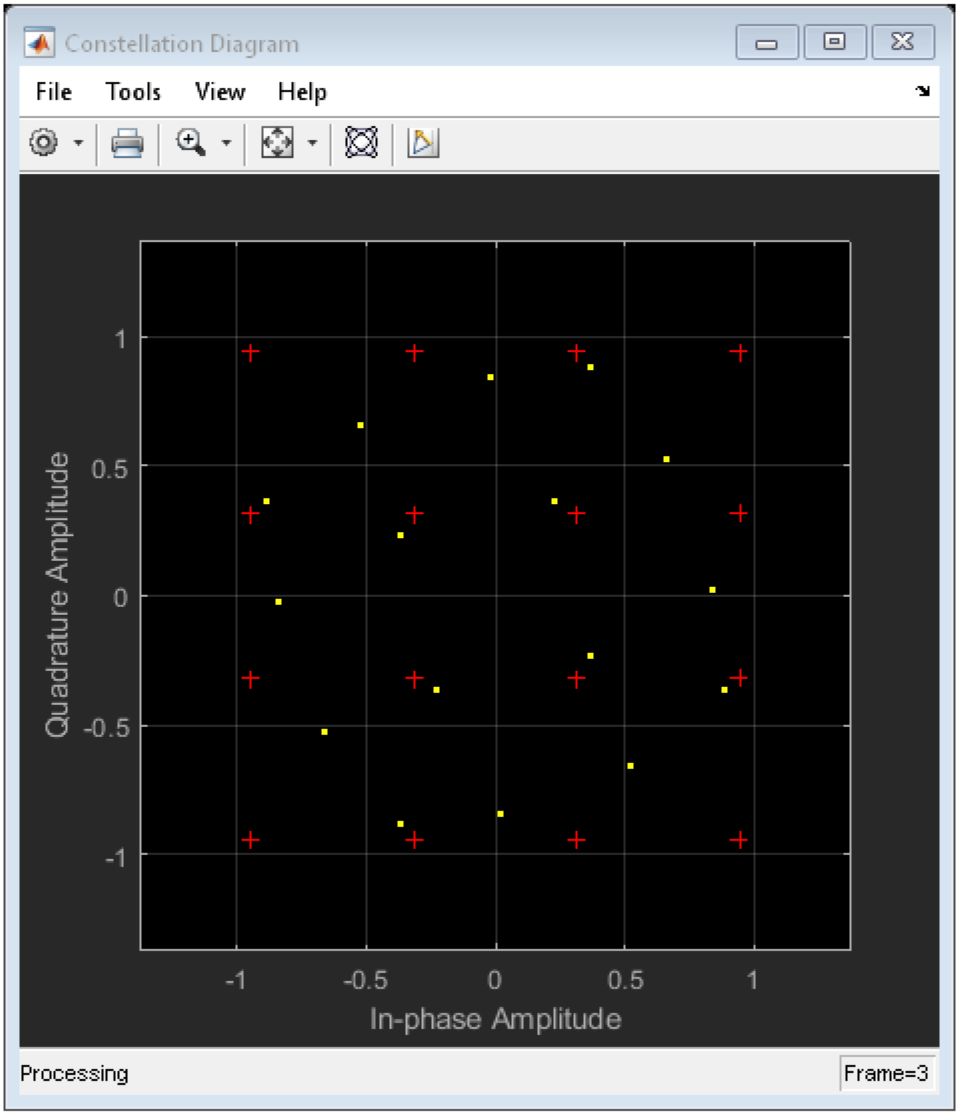
```
txSig = amp(modSig);
constDiagram(txSig)
```



The corner points of the constellation have moved toward the origin due to amplifier gain compression.

Introduce a small AM/PM conversion, and display the received signal constellation.

```
amp.AMPMConversion = 1;
txSig = amp(modSig);
constDiagram(txSig)
```

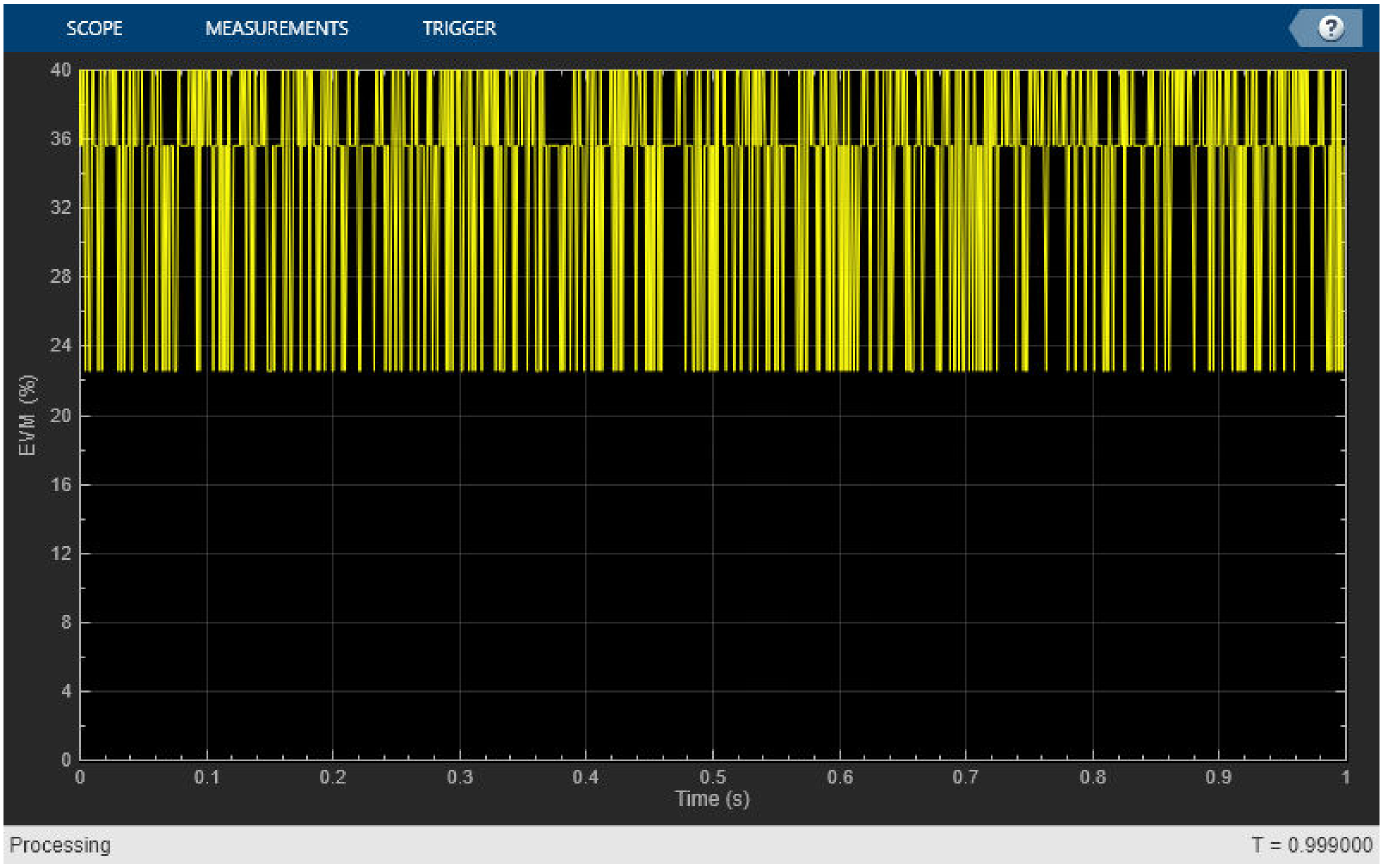


The constellation has rotated due to the AM/PM conversion. To compute the time-varying EVM, release the EVM object and set the `AveragingDimensions` property to 2. To estimate the EVM against an input signal, omit the `ReferenceSignalSource` property definition. This method produces more accurate results.

```
evm = comm.EVM('AveragingDimensions',2);
evmTime = evm(modSig,txSig);
```

Plot the time-varying EVM of the distorted signal.

```
timeScope(evmTime)
```



Compute the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 35.5812
```

Compute the MER.

```
mer = comm.MER;
snrEst = mer(modSig,txSig)
```

```
snrEst = 8.1390
```

The SNR (≈8 dB) is reduced from its initial value (∞) due to amplifier distortion.

Specify input power levels ranging from 0 to 40 dBm. Convert those levels to their linear equivalent in W. Initialize the output power vector.

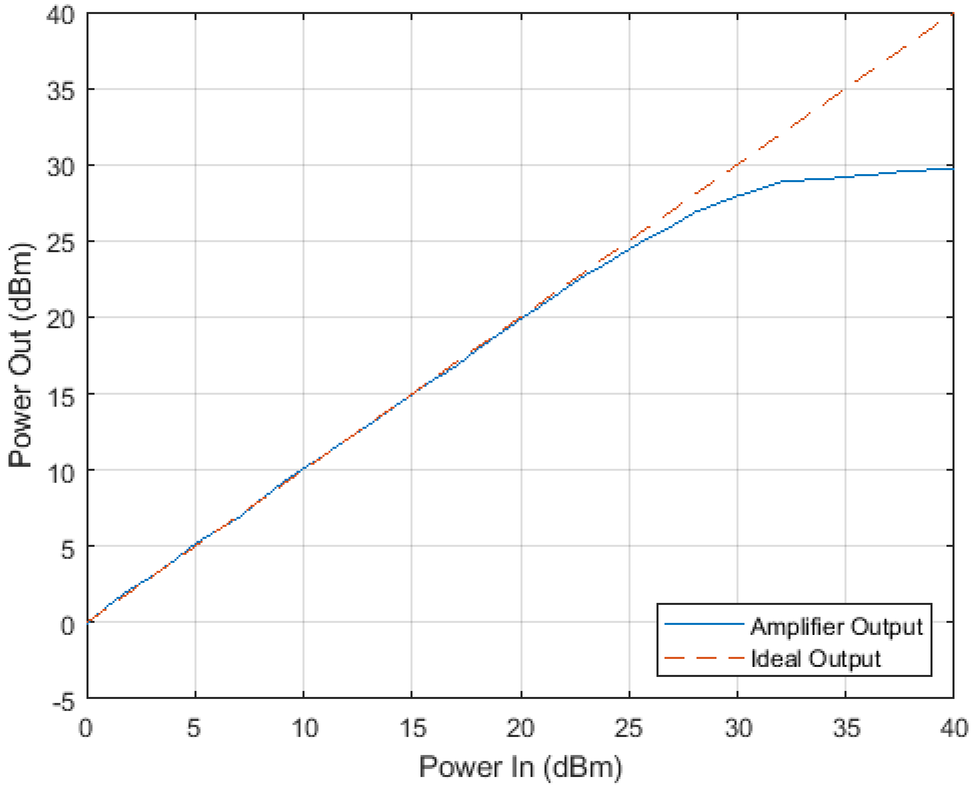
```
powerIn = 0:40;
pin = 10.^((powerIn-30)/10);
powerOut = zeros(length(powerIn),1);
```

Measure the amplifier output power for the range of input power levels.

```
for k = 1:length(powerIn)
    data = randi([0 15],1000,1);
    txSig = qammod(data,16,'UnitAveragePower',true)*sqrt(pin(k));
    ampSig = amp(txSig);
    powerOut(k) = 10*log10(var(ampSig))+30;
end
```

Plot the power output versus power input curve.

```
figure
plot(powerIn,powerOut,powerIn,powerIn,'--')
legend('Amplifier Output','Ideal Output','location','se')
xlabel('Power In (dBm)')
ylabel('Power Out (dBm)')
grid
```



The output power levels off at 30 dBm. The amplifier exhibits nonlinear behavior for input power levels greater than 25 dBm.

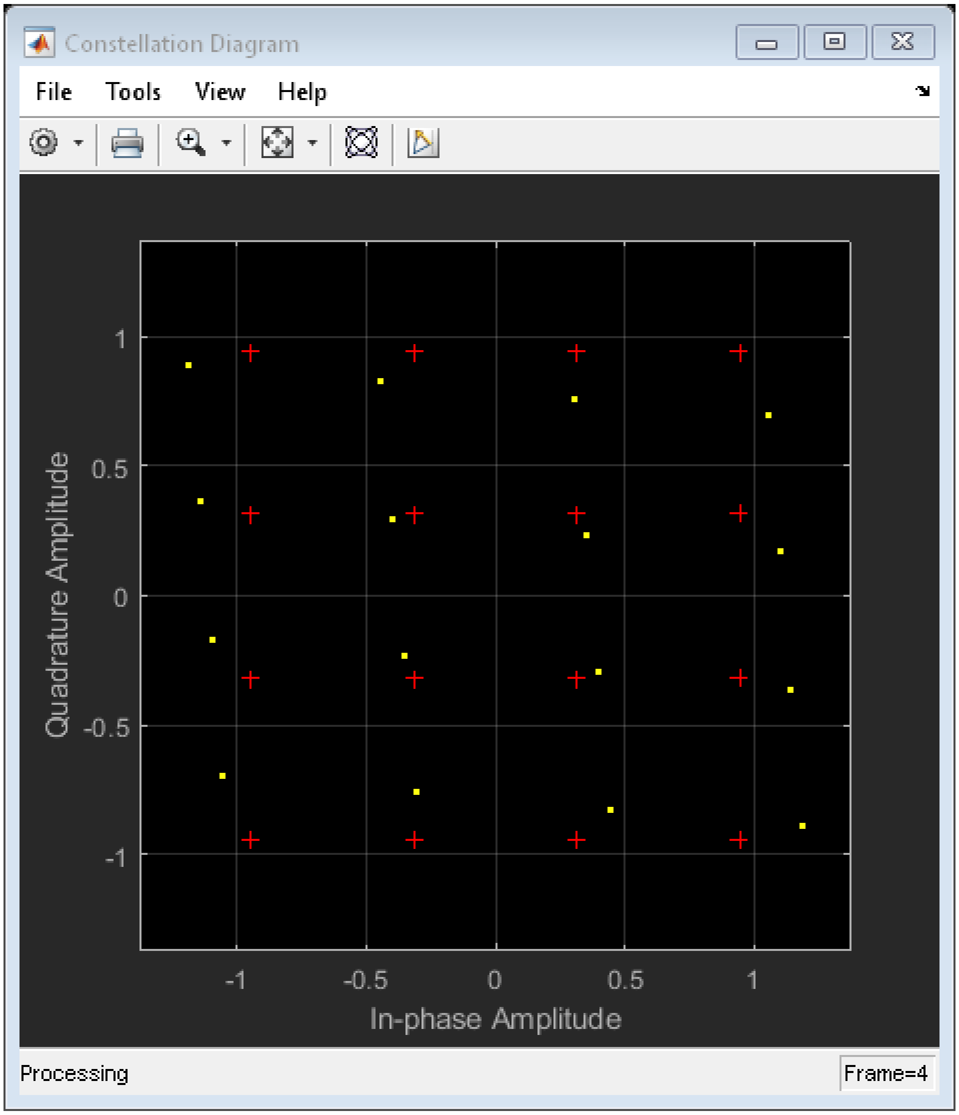
I/Q Imbalance

Apply an amplitude and phase imbalance to the modulated signal using the `iqimbal` function.

```
ampImb = 3;
phImb = 10;
rxSig = iqimbal(modSig,ampImb,phImb);
```

Plot the received constellation.

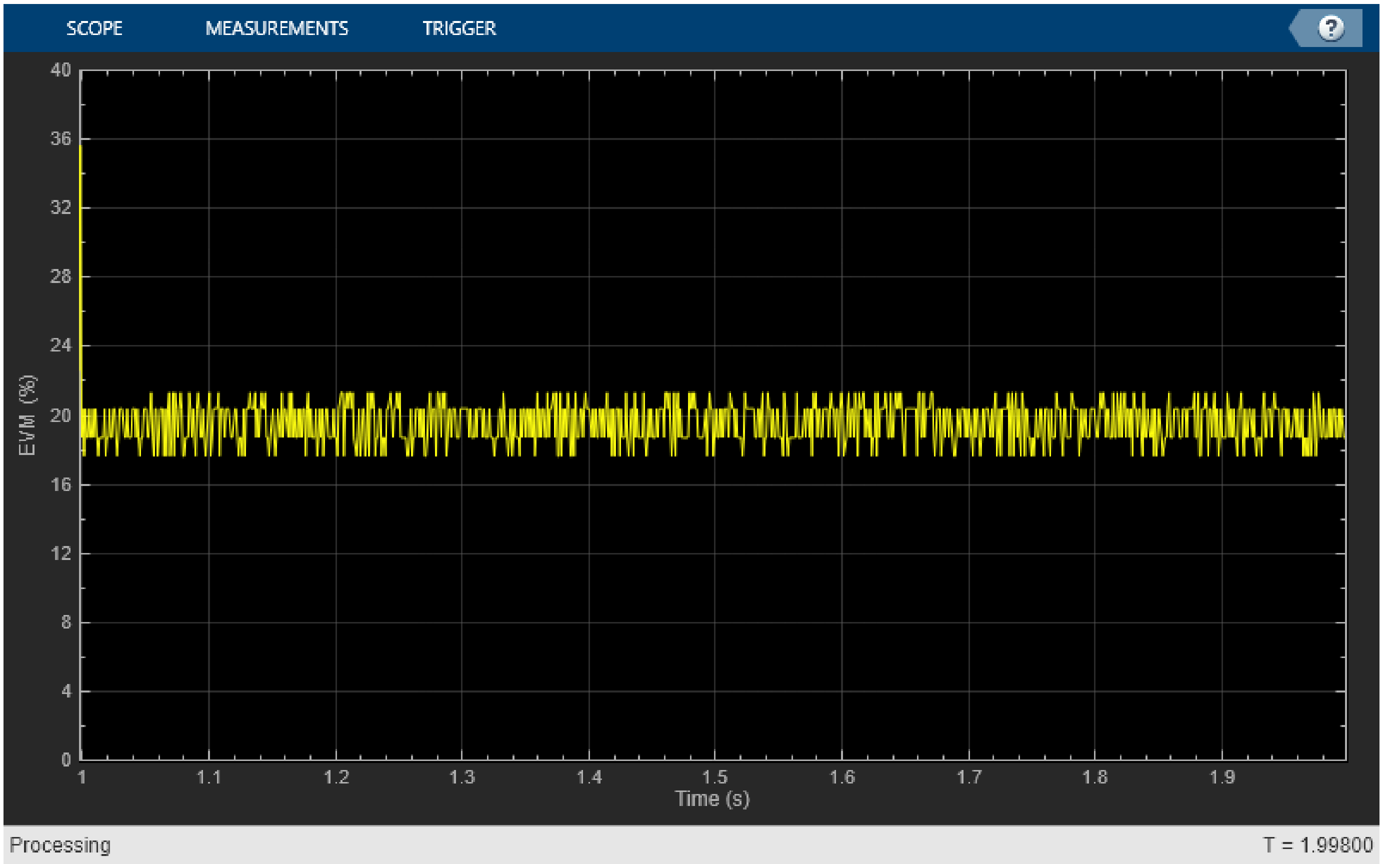
```
constDiagram(rxSig)
```



The magnitude and phase of the constellation has changed as a result of the I/Q imbalance.

Calculate and plot the time-varying EVM.

```
evmTime = evm(modSig,rxSig);
timeScope(evmTime)
```



The EVM exhibits a behavior that is similar to that experienced with a nonlinear amplifier though the variance is smaller.

Create a 100 Hz sine wave having a 1000 Hz sample rate.

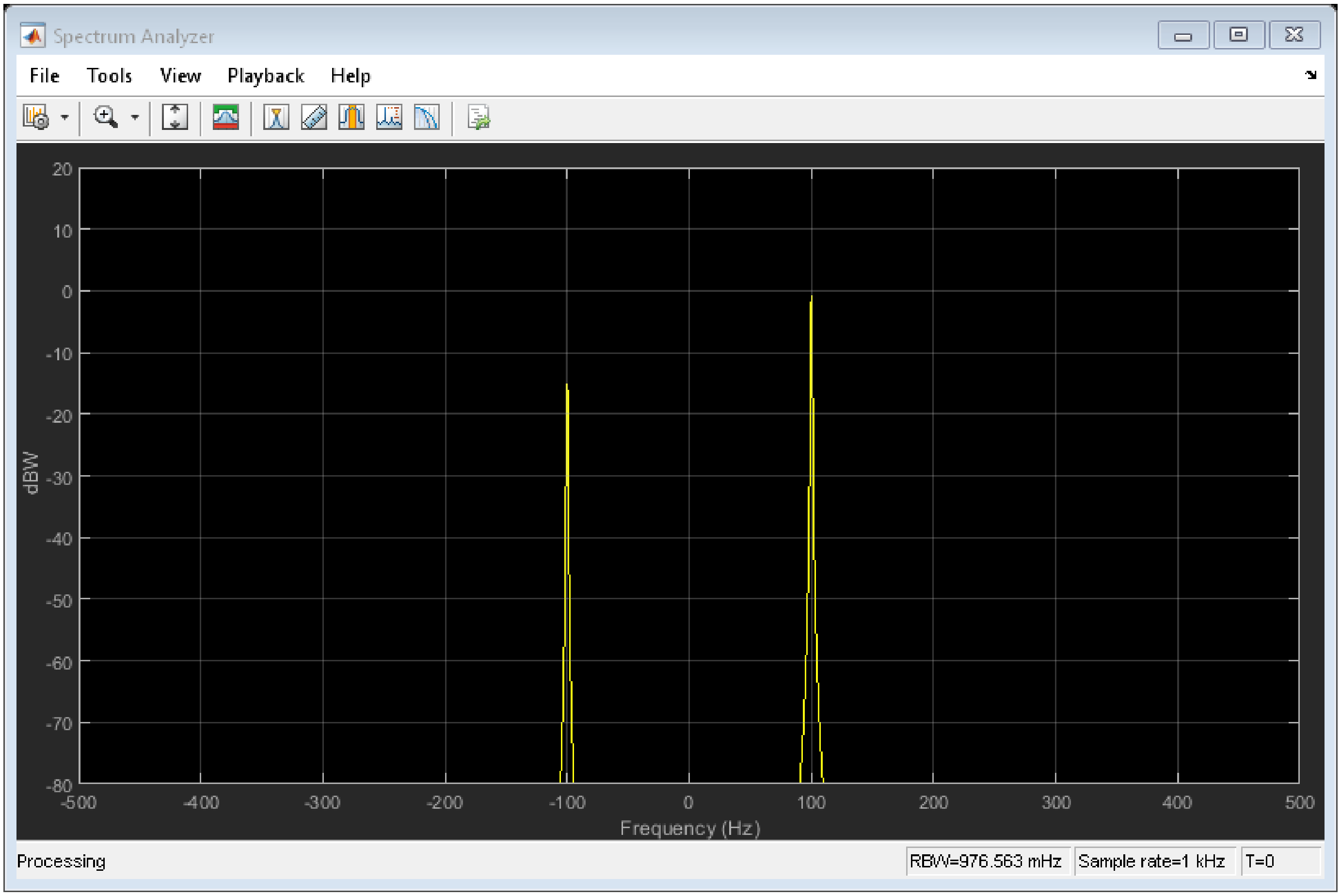
```
sinewave = dsp.SineWave('Frequency',100,'SampleRate',1000, ...
    'SamplesPerFrame',1e4,'ComplexOutput',true);

x = sinewave();
```

Apply the same 3 dB and 10 degree I/Q imbalance.

```
ampImb = 3;
phImb = 10;
y = iqimbal(x,ampImb,phImb);
```

Plot the spectrum of the imbalanced signal.

```
spectrum = dsp.SpectrumAnalyzer('SampleRate',1000,'PowerUnits','dBW');

spectrum(y)
```

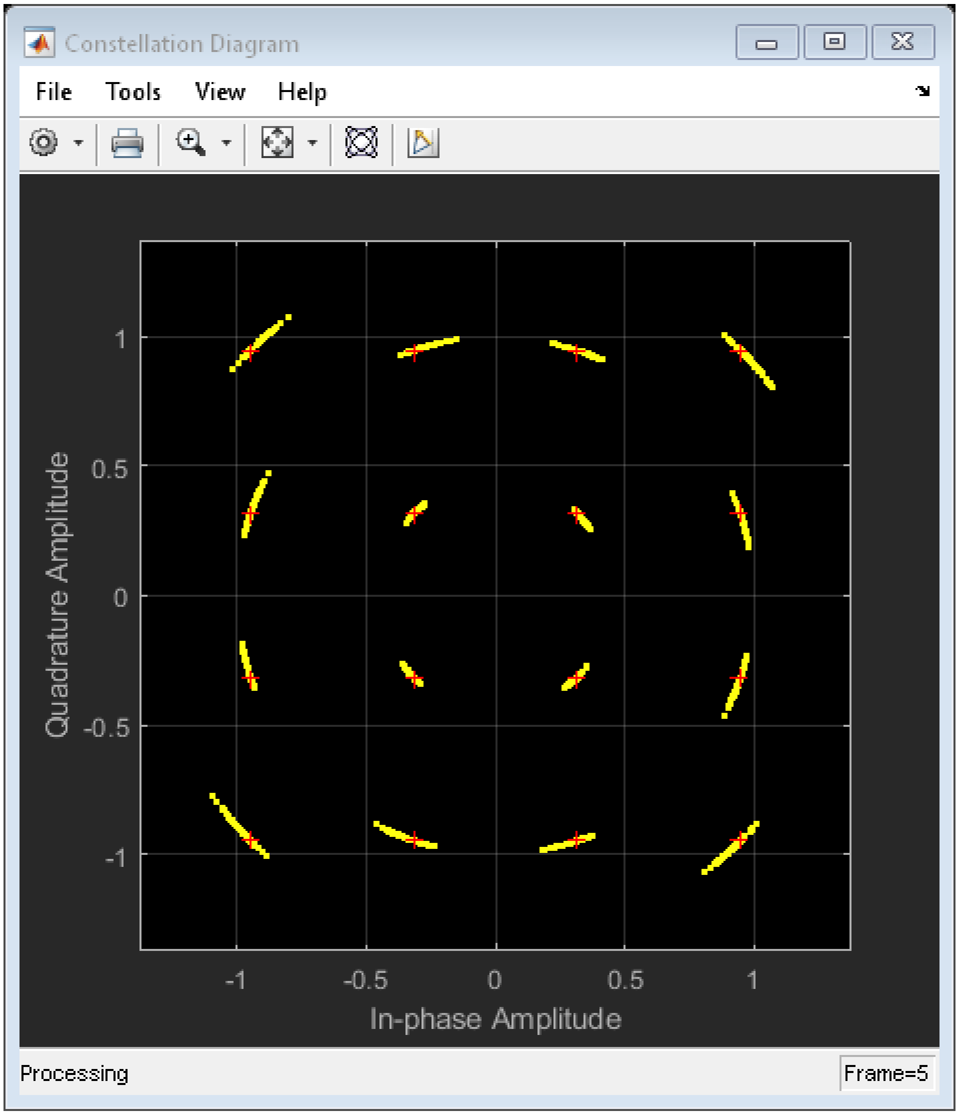


The I/Q imbalance introduces a second tone at -100 Hz, which is the inverse of the input tone.

Phase Noise

Apply phase noise to the transmitted signal. Plot the resulting constellation diagram.

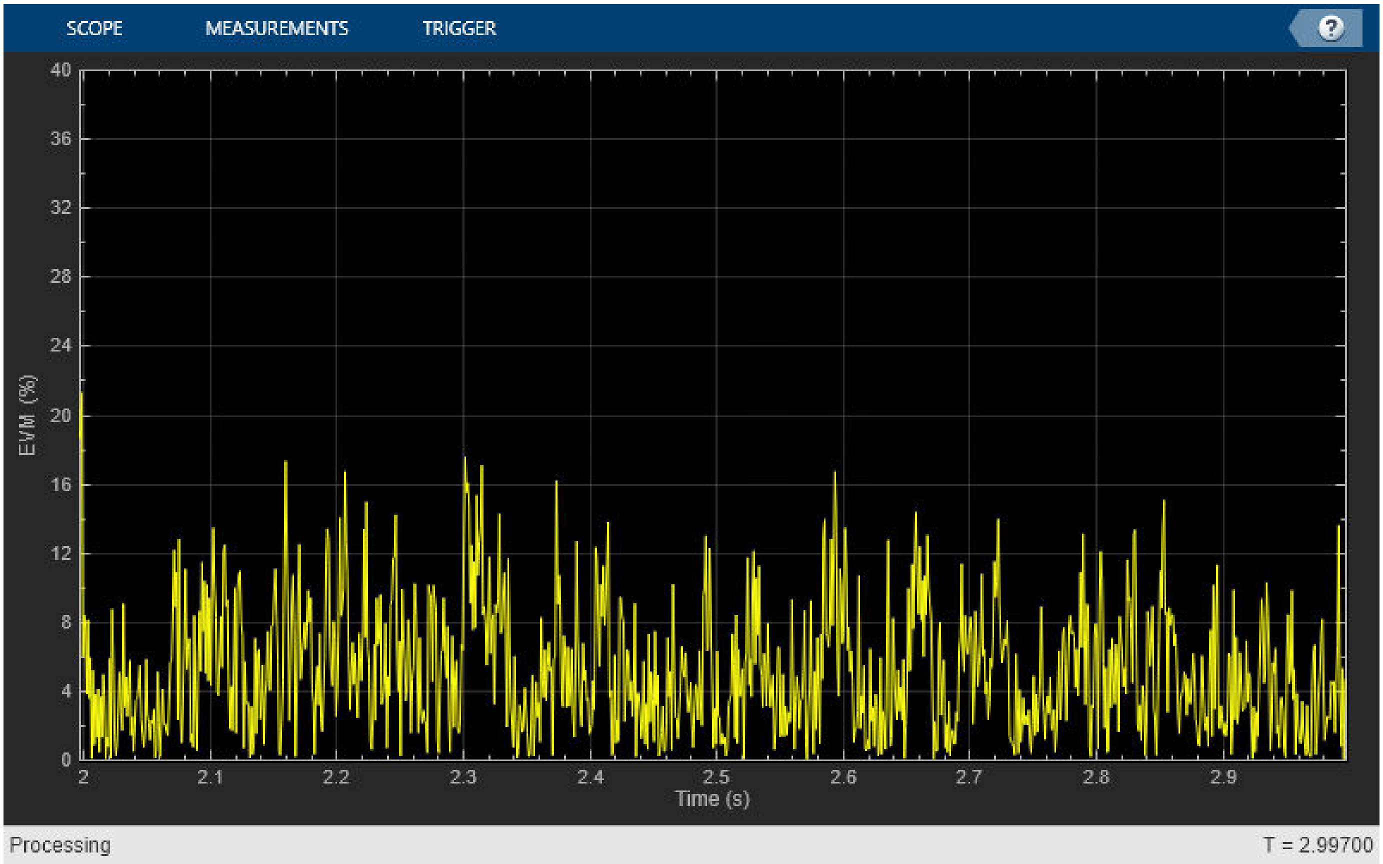
```
pnoise = comm.PhaseNoise('Level',-50,'FrequencyOffset',20,'SampleRate',fs);
pnoiseSig = pnoise(modSig);
constDiagram(pnoiseSig)
```



The phase noise introduces a rotational jitter.

Compute and plot the EVM of the received signal.

```
evmTime = evm(modSig,pnoiseSig);
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 6.2675
```

Filter Effects

Specify the samples per symbol parameter. Create a pair of raised cosine matched filters.

```
sps = 4;
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.2,'FilterSpanInSymbols',8, ...
    'OutputSamplesPerSymbol',sps,'Gain',sqrt(sps));

rxfilter = comm.RaisedCosineReceiveFilter('RolloffFactor',0.2,'FilterSpanInSymbols',8, ...
    'InputSamplesPerSymbol',sps,'Gain',1/sqrt(sps), ...
    'DecimationFactor',sps);
```

Determine the delay through the matched filters.

```
fltDelay = 0.5*(txfilter.FilterSpanInSymbols + rxfilter.FilterSpanInSymbols);
```

Pass the modulated signal through the matched filters.

```
filtSig = txfilter(modSig);
rxSig = rxfilter(filtSig);
```

To account for the delay through the filters, discard the first `fltDelay` samples.

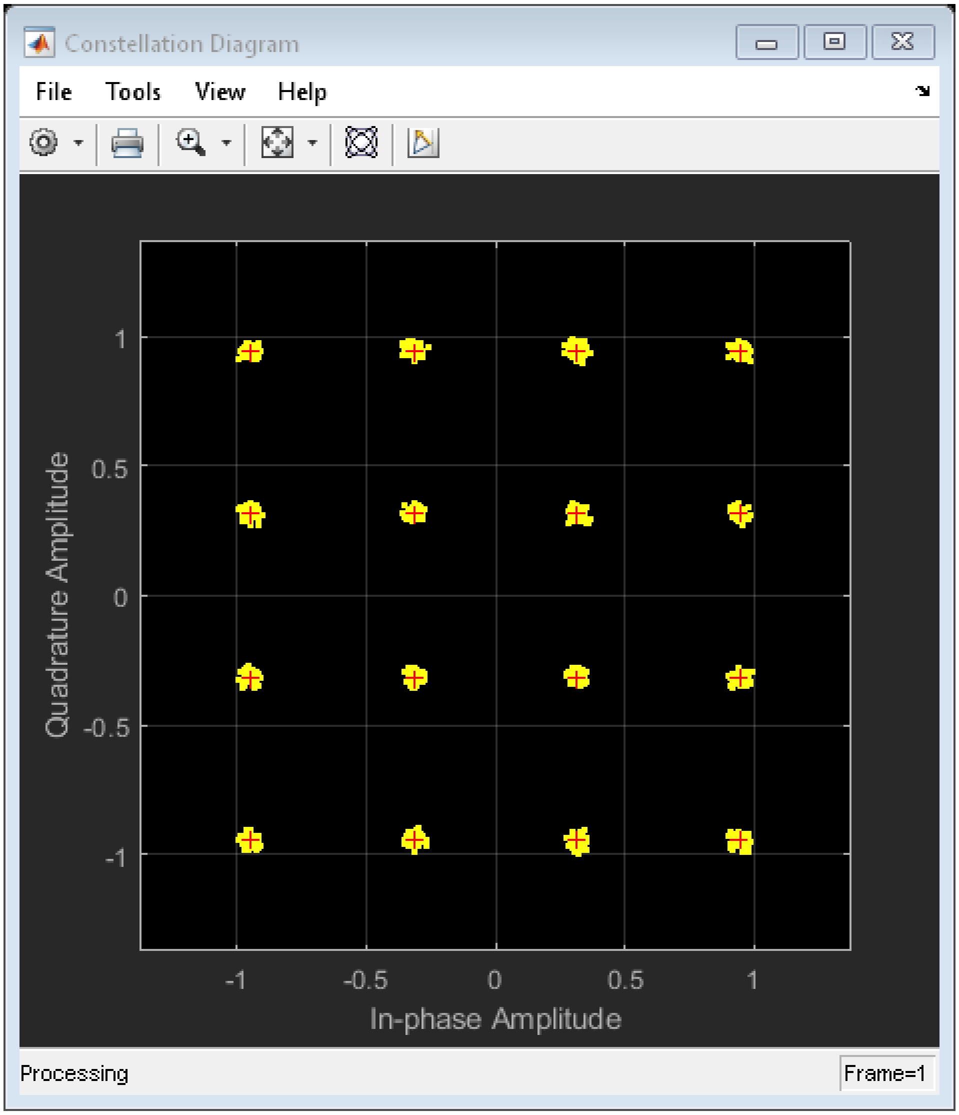
```
rxSig = rxSig(fltDelay+1:end);
```

To accommodate the change in the number of received signal samples, create new constellation diagram and time scope objects.

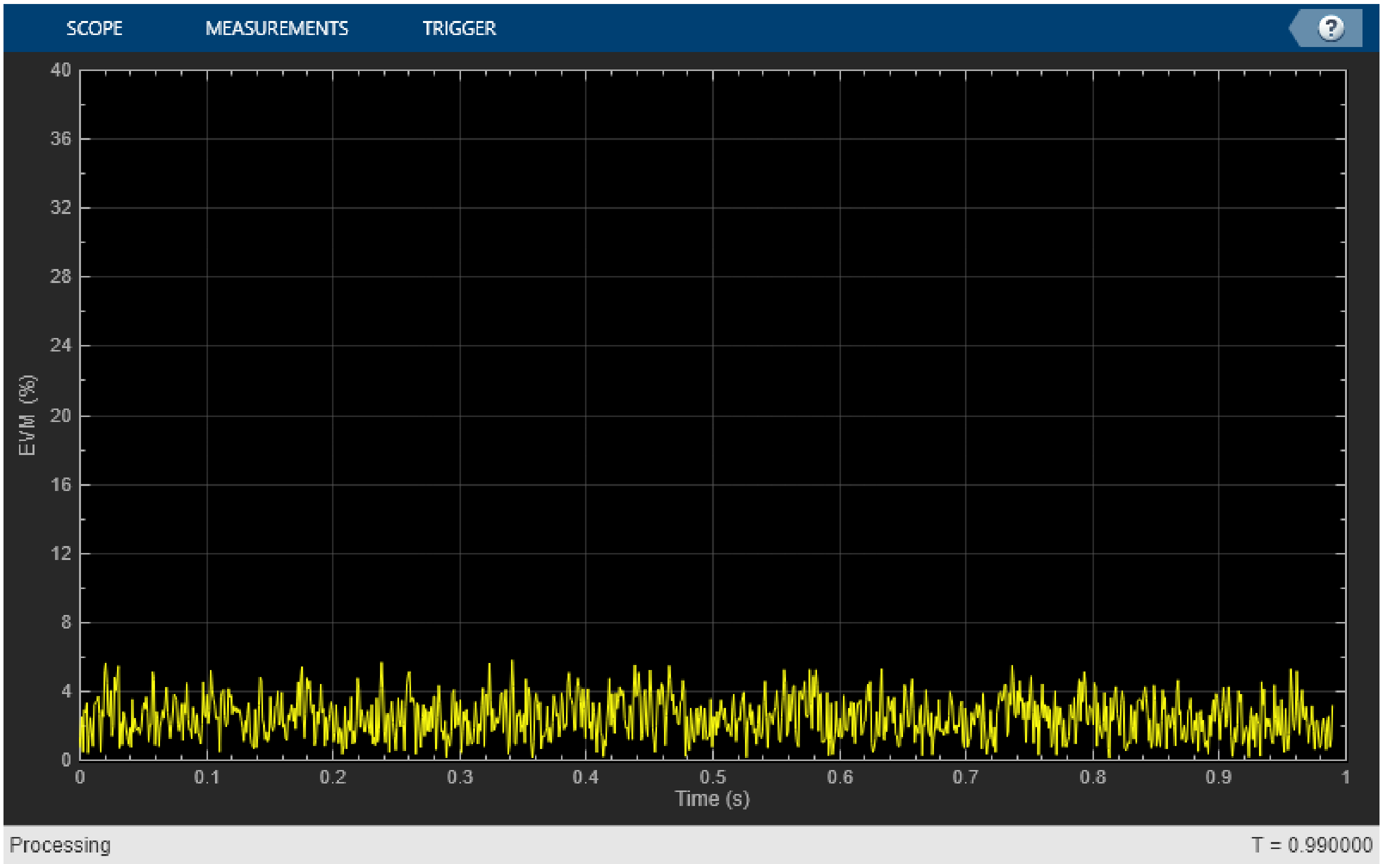
```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
timeScope = timescope('YLimits',[0 40],'SampleRate',fs,'TimeSpanSource','property','TimeSpan',1,
    'ShowGrid',true,'YLabel','EVM (%)');
```

Estimate EVM. Plot the received signal constellation diagram and the time-varying EVM.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
evmTime = evm(rxSig);
constDiagram(rxSig)
```



```
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 2.7174
```

Determine the equivalent SNR.

```
mer = comm.MER;
snrEst = mer(modSig(1:end-fltDelay),rxSig)
```

```
snrEst = 31.4719
```

Combined Effects

Combine the effects of the filters, nonlinear amplifier, AWGN, and phase noise. Display the constellation and EVM diagrams.

Create EVM, time scope and constellation diagram objects.

```
evm = comm.EVM('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst, ...
    'Normalization','Average constellation power','AveragingDimensions',2);
timeScope = timescope('YLimits',[0 40],'SampleRate',fs,'TimeSpanSource','property','TimeSpan',1,
    'ShowGrid',true,'YLabel','EVM (%)');
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refConst);
```

Specify the nonlinear amplifier and phase noise objects.

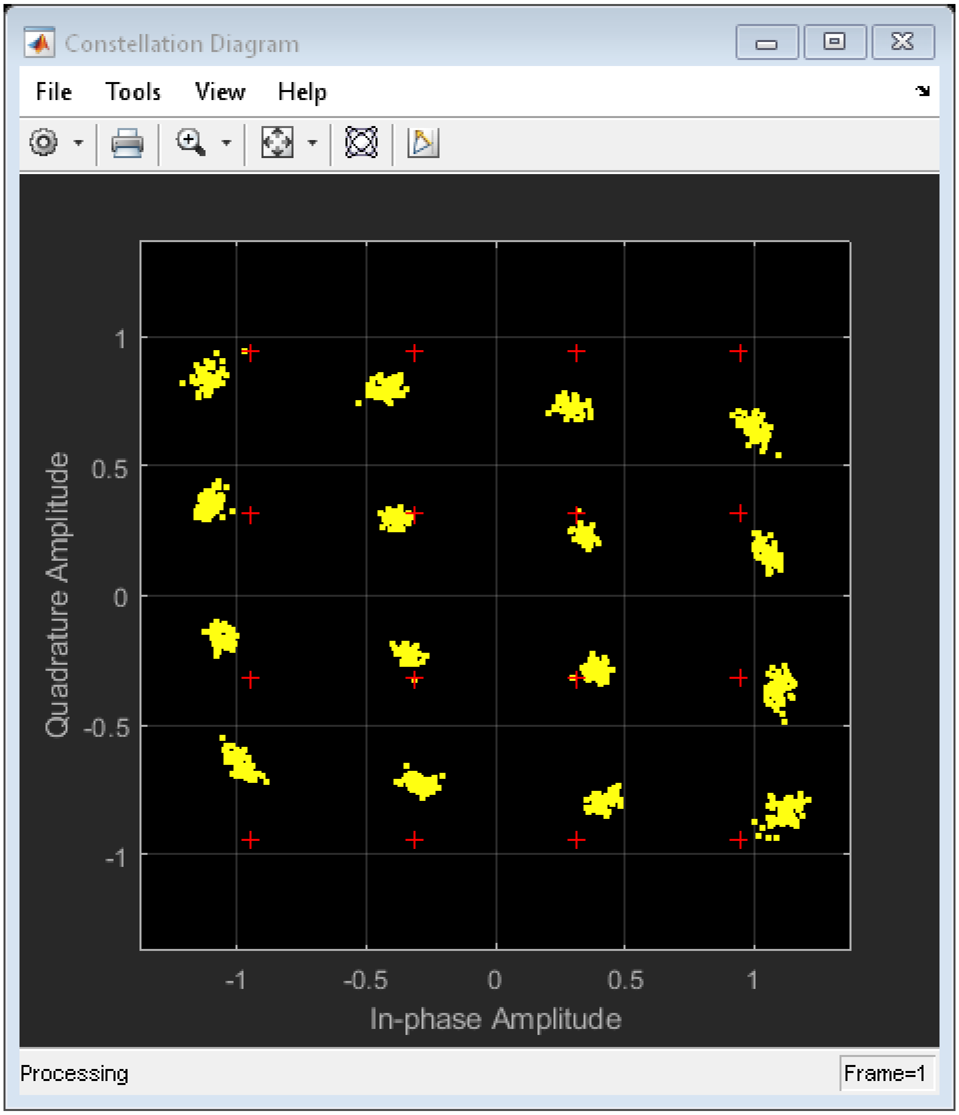
```
amp = comm.MemorylessNonlinearity('IIP3',45,'AMPMConversion',0);
pnoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',fs);
```

Filter and then amplify the modulated signal.

```
txfiltOut = txfilter(modSig);
txSig = amp(txfiltOut);
```

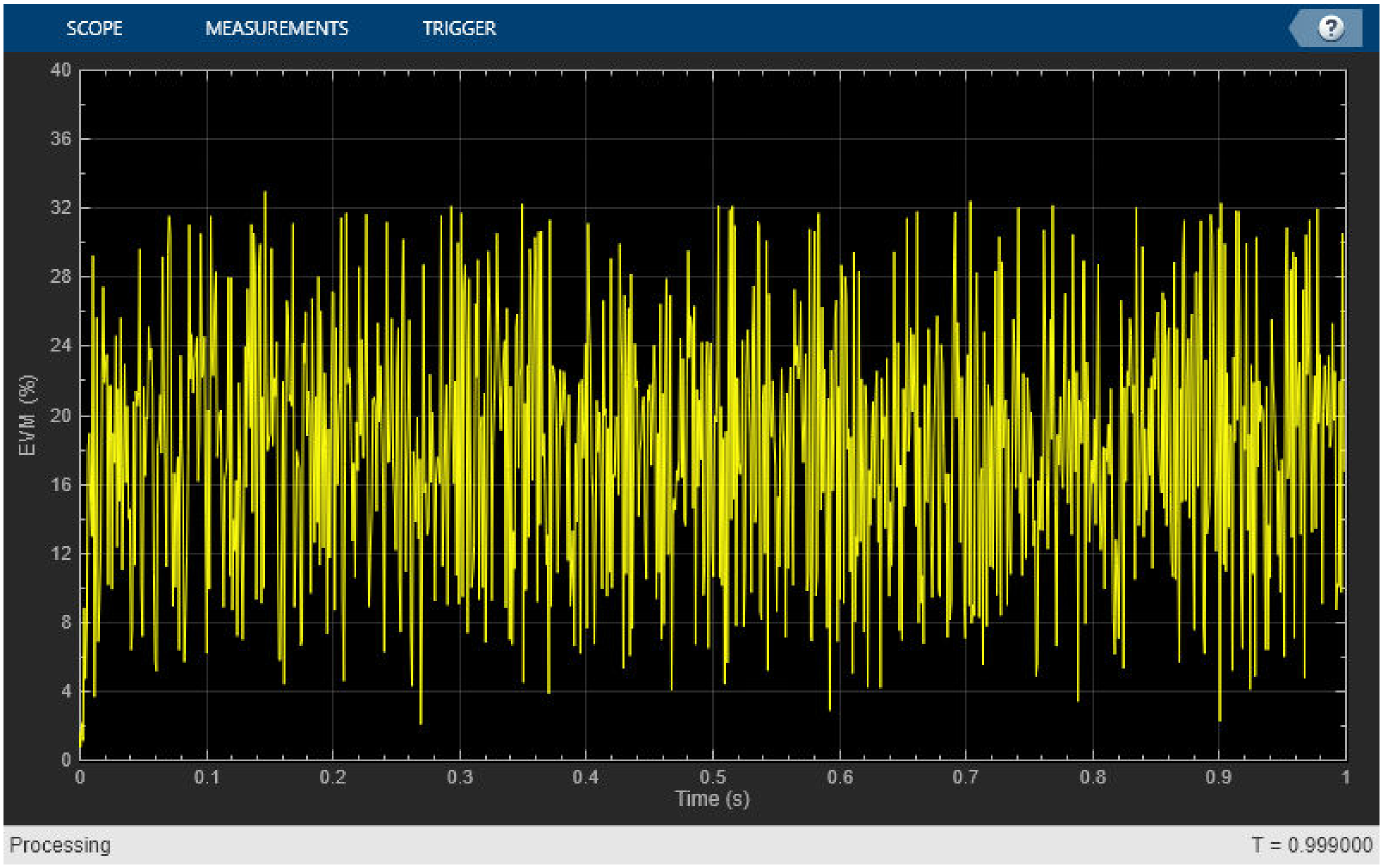
Add phase noise. Pass the impaired signal through the AWGN channel. Plot the constellation diagram.

```
rxSig = awgn(txSig,snrdB);
iqImbalSig = iqimbal(rxSig,ampImb,phImb);
pnoiseSig = pnoise(iqImbalSig);
rxfiltOut = rxfilter(pnoiseSig);
constDiagram(rxfiltOut)
```



Calculate the time-varying EVM. Plot the result.

```
evmTime = evm(rxfiltOut);
timeScope(evmTime)
```



Determine the RMS EVM.

```
evmRMS = sqrt(mean(evmTime.^2))
```

```
evmRMS = 19.4977
```

Estimate the SNR.

```
mer = comm.MER('ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
snrEst = mer(rxfiltOut)
```

```
snrEst = 14.1829
```

This value is approximately 6 dB worse than the specified value of 30 dB, which means that the effects of the other impairments are significant and will degrade the bit error rate performance.

More About

Use the Channel Visualization Tool

You can use a graphical plotting function that helps you visualize the characteristics of a fading channel. For example, the following code opens the channel visualization tool showing a three-path Rayleigh channel through which a random signal is passed:

```
% Three-Path Rayleigh channel
h = rayleighchan(1/100000, 130, [0 1.5e-5 3.2e-5], [0, -3, -3]);
tx = randi([0 1],500,1);          % Random bit stream
hmod = comm.DBPSKModulator;       % Create DBPSKModulator
dpskSig = step(hmod,tx);          % DPSK signal
h.StoreHistory = true;            % Allow states to be stored
y = filter(h, dpskSig);           % Run signal through channel
plot(h);                          % Call Channel Visualization Tool
```

See “Fading Channels” for a description of fading channels and objects.

Compatibility Considerations

plot has been removed

Errors starting in R2020b

plot (channel) has been removed. Use comm.RicianChannel instead.

See Also

comm.RayleighChannel | comm.RicianChannel

Introduced before R2006a

plotPhaseNoiseFilter

Plot response of phase noise filter block

Syntax

```
plotPhaseNoiseFilter(blockname)
```

Description

`plotPhaseNoiseFilter(blockname)` plots the response of the phase noise filter associated with the Phase Noise block specified by the variable `blockname`.

Examples

View Filter Response of Phase Noise Block

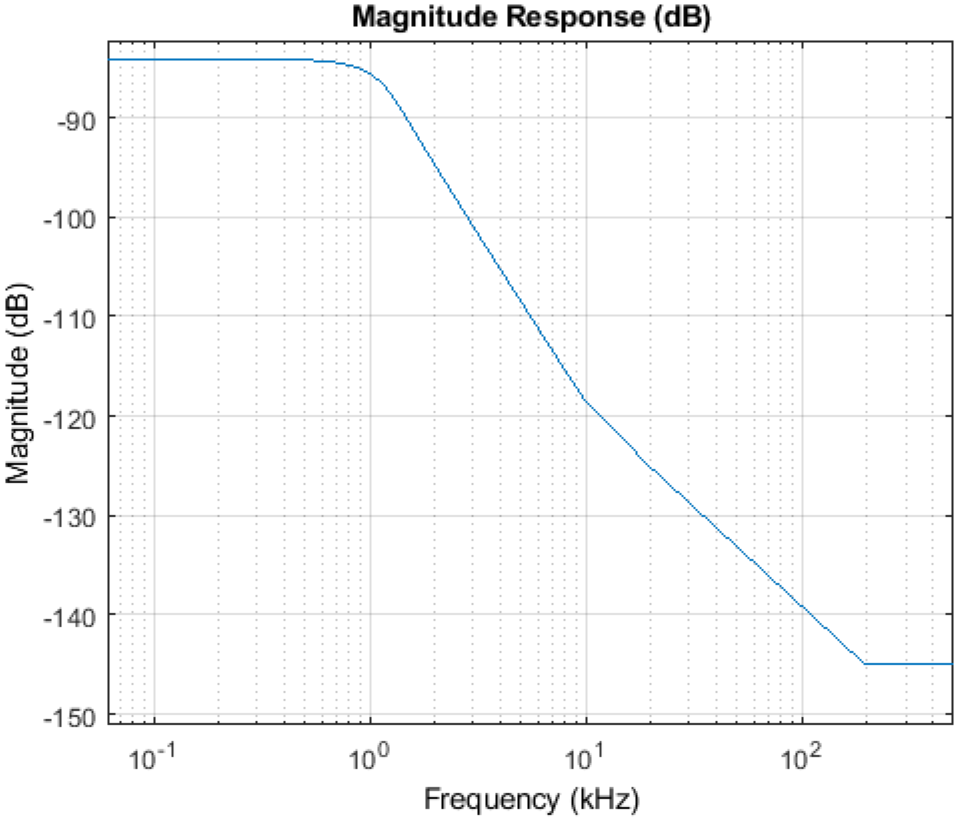
This example shows how to use the `plotPhaseNoiseFilter` function to view the filter response of a Phase Noise block in a Simulink model.

Load a Simulink model that contains a Phase Noise block. The `load_system` (Simulink) command loads a model into memory without making its model window visible. The function will also work with models whose window is visible. The example, `slex_phasenoise`, contains a `Phase Noise` block.

```
load_system('slex_phasenoise')
```

Run the `plotPhaseNoiseFilter` function to view the filter response of the block Phase Noise.

```
plotPhaseNoiseFilter('slex_phasenoise/Phase Noise')
```



Input Arguments

`blockname` — Phase noise block name
character vector

The name of a Phase Noise block in a Simulink model

Example: `plotPhaseNoiseFilter('Model Name/Phase Noise')`

Data Types: `char`

See Also

Phase Noise

Introduced in R2014b

pmdemod

Phase demodulation

Syntax

```
z = pmdemod(y,Fc,Fs,phasedev)
z = pmdemod(y,Fc,Fs,phasedev,ini_phase)
```

Description

`z = pmdemod(y,Fc,Fs,phasedev)` demodulates the phase-modulated signal `y` at the carrier frequency `Fc` (hertz). `z` and the carrier signal have sampling rate `Fs` (hertz), where `Fs` must be at least `2*Fc`. The `phasedev` argument is the phase deviation of the modulated signal, in radians.

`z = pmdemod(y,Fc,Fs,phasedev,ini_phase)` specifies the initial phase of the modulated signal, in radians.

Examples

Recover Phase Modulated Signal from AWGN Channel

Set the sample rate. To plot the signals, create a time vector.

```
fs = 50;
t = (0:2*fs+1)'/fs;
```

Create a sinusoidal input signal.

```
x = sin(2*pi*t) + sin(4*pi*t);
```

Set the carrier frequency and phase deviation.

```
fc = 10;
phasedev = pi/2;
```

Modulate the input signal.

```
tx = pmmod(x,fc,fs,phasedev);
```

Pass the signal through an AWGN channel.

```
rx = awgn(tx,10,'measured');
```

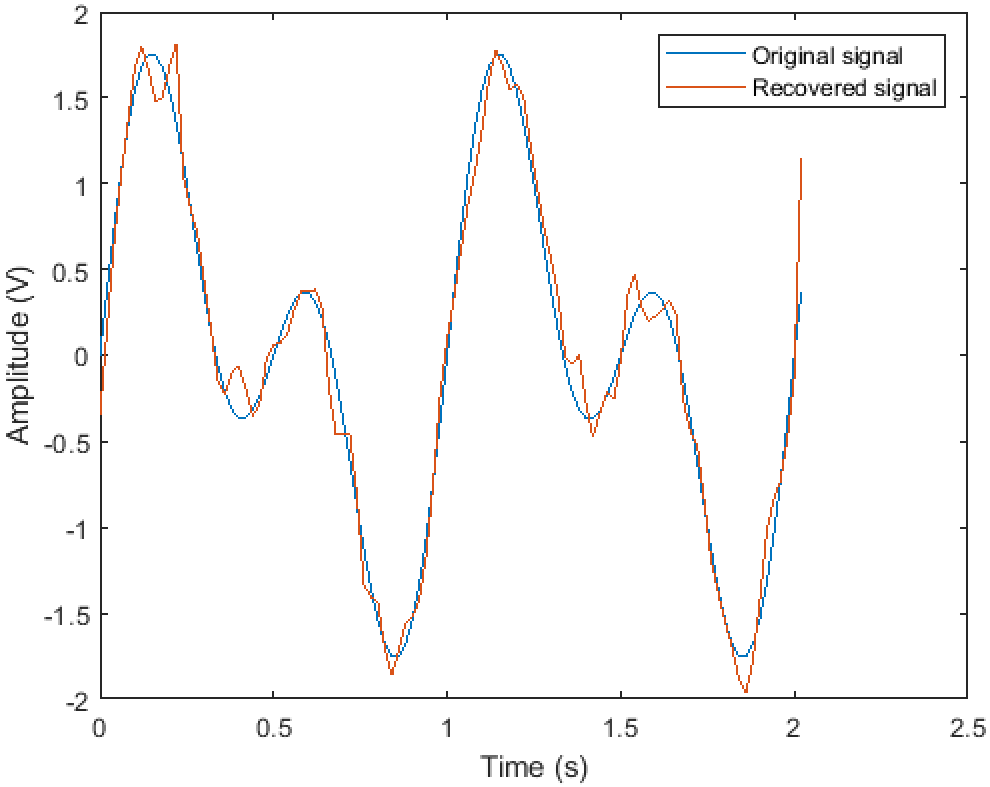
Demodulate the noisy signal.

```
y = pmdemod(rx,fc,fs,phasedev);
```

Plot the original and recovered signals.

```
figure; plot(t,[x y]);
legend('Original signal','Recovered signal');
```

```
xlabel('Time (s)')
ylabel('Amplitude (V)')
```



See Also

`fmdemod` | `fmmod` | `pmmod`

Topics

"Digital Modulation"

Introduced before R2006a

pmmod

Phase modulation

Syntax

```
y = pmmod(x,Fc,Fs,phasedev)
y = pmmod(x,Fc,Fs,phasedev,ini_phase)
```

Description

`y = pmmod(x,Fc,Fs,phasedev)` modulates the message signal `x` using phase modulation.

`y = pmmod(x,Fc,Fs,phasedev,ini_phase)` specifies the initial phase of the modulated signal in radians.

Examples

Recover Phase Modulated Signal from AWGN Channel

Set the sample rate. To plot the signals, create a time vector.

```
fs = 50;
t = (0:2*fs+1)'/fs;
```

Create a sinusoidal input signal.

```
x = sin(2*pi*t) + sin(4*pi*t);
```

Set the carrier frequency and phase deviation.

```
fc = 10;
phasedev = pi/2;
```

Modulate the input signal.

```
tx = pmmod(x,fc,fs,phasedev);
```

Pass the signal through an AWGN channel.

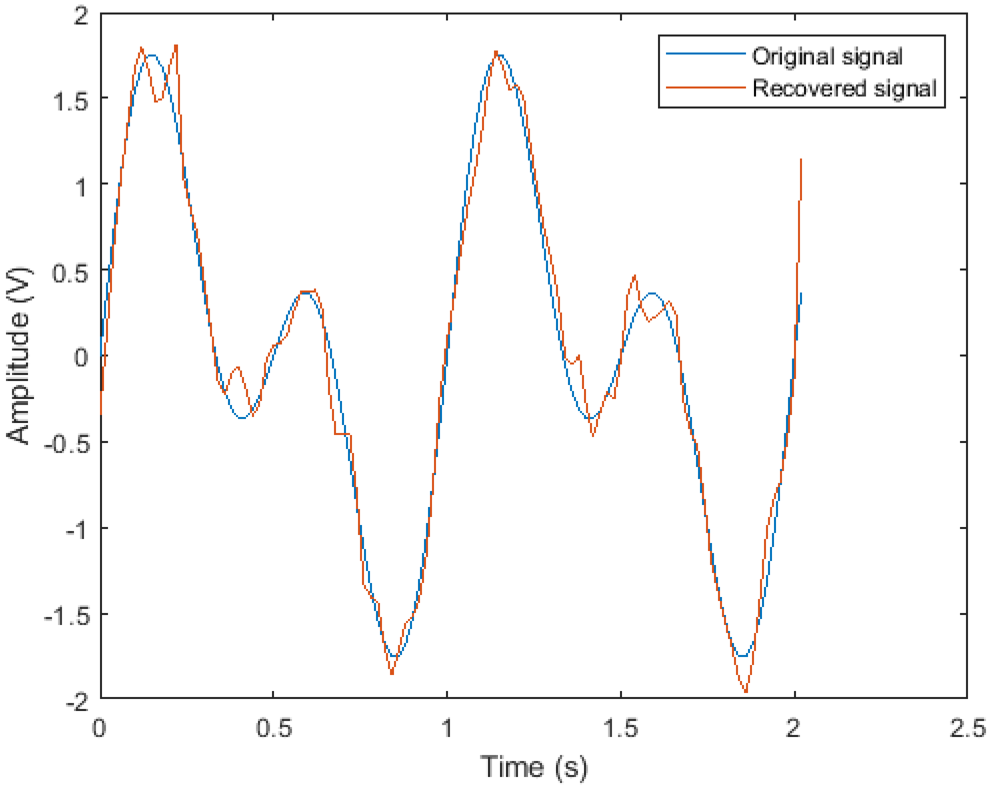
```
rx = awgn(tx,10,'measured');
```

Demodulate the noisy signal.

```
y = pmdemod(rx,fc,fs,phasedev);
```

Plot the original and recovered signals.

```
figure; plot(t,[x y]);
legend('Original signal','Recovered signal');
xlabel('Time (s)')
ylabel('Amplitude (V)')
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. If `x` is a matrix, `pmmod` processes the columns independently.

Example: `sin(2*pi*t) + sin(6*pi*t)`

Data Types: `double`

Fc — Carrier frequency

positive scalar

Carrier frequency, specified as a positive scalar.

Data Types: `double`

Fs — Sample rate

positive scalar

Sample rate, specified as a positive scalar. `Fs` must be at least 2*`Fc`.

Data Types: `double`

ini_phase — Initial phase

0 (default) | scalar | [ ]

Initial phase of the modulated signal (in radians), specified as a real scalar.

Example: pi/4

Data Types: double

phasedev — Phase deviation

positive scalar

Phase deviation, specified as a positive scalar in radians.

Data Types: double

Output Arguments

y — PM-modulated output signal

vector | matrix

Complex baseband representation of a PM-modulated signal, returned as vector or matrix of complex values. The columns of y represent independent channels.

Data Types: double | single

See Also

fmdemod | fmmod | pmdemod

Topics

“Digital Modulation”

Introduced before R2006a

poly2trellis

Convert convolutional code polynomials to trellis description

Syntax

```
trellis = poly2trellis(ConstraintLength,CodeGenerator)
trellis = poly2trellis(ConstraintLength,CodeGenerator,FeedbackConnection)
```

Description

`trellis = poly2trellis(ConstraintLength,CodeGenerator)` returns the trellis structure description corresponding to the conversion for a rate *K* / *N* feedforward encoder. *K* is the number of input bit streams to the encoder, and *N* is the number of output connections. `ConstraintLength` specifies the delay for the input bit streams to the encoder. `CodeGenerator` specifies the output connections for the input bit streams to the encoder.

The poly2trellis function accepts a polynomial description of a convolutional encoder and returns the corresponding trellis structure description. This output can be used as an input to the `convenc` and `vitdec` functions. It can also be used as a mask parameter value for the Convolutional Encoder, Viterbi Decoder, and APP Decoder blocks.

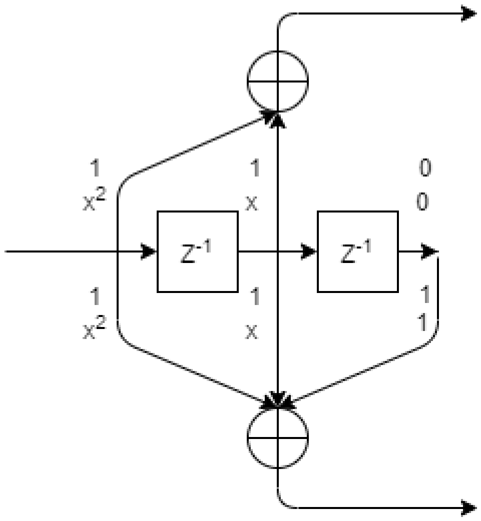
Note When used with a feedback polynomial, poly2trellis makes a feedback connection to the input of the trellis.

`trellis = poly2trellis(ConstraintLength,CodeGenerator,FeedbackConnection)` returns the trellis structure description corresponding to the conversion for a rate *K* / *N* feedback encoder. *K* is the number of input bit streams to the encoder, and *N* is the number of output connections. `ConstraintLength` specifies the delay for the input bit streams to the encoder. `CodeGenerator` specifies the output connections for the input bit streams to the encoder. `FeedbackConnection` specifies the feedback connection for each of the *K* input bit streams to the encoder.

Examples

Use Trellis Structure for Rate 1/2 Feedforward Convolutional Encoder

Use a trellis structure to configure the rate 1/2 feedforward convolutional code in this diagram.



Create a trellis structure, setting the constraint length to 3 and specifying the code generator as a vector of octal values. The diagram indicates the binary values and polynomial form, indicating the left-most bit is the most-significant-bit (MSB). The binary vector [1 1 0] represents octal 6 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 1 1] represents octal 7 and corresponds to the lower row of binary digits in the diagram. These binary digits indicate connections from the outputs of the registers to the two adders in the diagram.

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

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 4
          nextStates: [4x2 double]
             outputs: [4x2 double]
```

Generate random binary data. Convolutionally encode the data, by using the specified trellis structure. Decode the coded data by using the Viterbi algorithm with the specified trellis structure, 34 for its traceback depth, truncated operation mode, and hard decisions.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
tbdepth = 34;
decodedData = vitdec(codedData,trellis,tbdepth,'trunc','hard');
```

Verify the decoded data has zero bit errors.

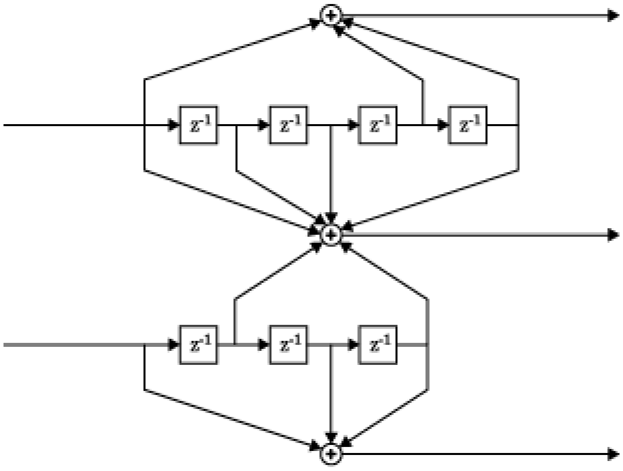
```
biterr(data,decodedData)
```

```
ans = 0
```

Trellis Structure for 2/3 Feedforward Convolutional Encoder

Create a trellis structure for a rate 2/3 feedforward convolutional code and display a portion of the next states of the trellis. See `convenc` for an example using this encoder.

The diagram shows a rate 2/3 encoder with two input streams, three output streams, and seven shift registers.



Create a trellis structure. Set the constraint length of the upper path to 5 and the constraint length of the lower path to 4. The octal representation of the code generator matrix corresponds to the taps from the upper and lower shift registers.

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

```
trellis = struct with fields:
     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

The structure field `numInputSymbols` equals 4 because two bit streams can produce four different input symbols. The structure field `numOutputSymbols` equals 8 because three bit streams produce eight different output symbols. Because the encoder has seven total shift registers, the number of possible states is 2^7 = 128, as shown by the `nextStates` field.

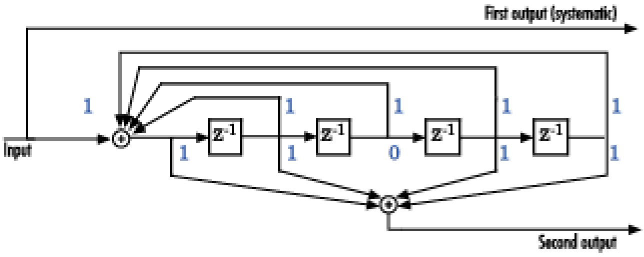
Display the first five rows of the 128-by-4 `trellis.nextStates` matrix.

```
trellis.nextStates(1:5,:)
```

```
ans = 5×4

     0    64     8    72
     0    64     8    72
     1    65     9    73
     1    65     9    73
     2    66    10    74
```

Use Trellis Structure for Rate 1/2 Feedback Convolutional Encoder

Create a trellis structure to represent the rate 1/2 systematic convolutional encoder with feedback shown in this diagram.



This encoder has 5 for its constraint length, [37 33] as its generator polynomial matrix, and 37 for its feedback connection polynomial.

The first generator polynomial is octal 37. The second generator polynomial is octal 33. The feedback polynomial is octal 37. The first generator polynomial matches the feedback connection polynomial because the first output corresponds to the systematic bits.

The binary vector [1 1 1 1 1] represents octal 37 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 1 0 1 1] represents octal 33 and corresponds to the lower row of binary digits in the diagram. These binary digits indicate connections from the outputs of the registers to the two adders in the diagram. The initial 1 corresponds to the input bit.

Convert the polynomial to a trellis structure by using the `poly2trellis` function. When used with a feedback polynomial, `poly2trellis` makes a feedback connection to the input of the trellis.

```
trellis = poly2trellis(5,[37 33],37)
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 16
          nextStates: [16x2 double]
             outputs: [16x2 double]
```

Generate random binary data. Convolutionally encode the data by using the specified trellis structure. Decode the coded data by using the Viterbi algorithm with the specified trellis structure, 34 for its traceback depth, truncated operation mode, and hard decisions.

```
data = randi([0 1],70,1);
codedData = convenc(data,trellis);
tbdepth = 34; % Traceback depth for Viterbi decoder
decodedData = vitdec(codedData,trellis,tbdepth,'trunc','hard');
```

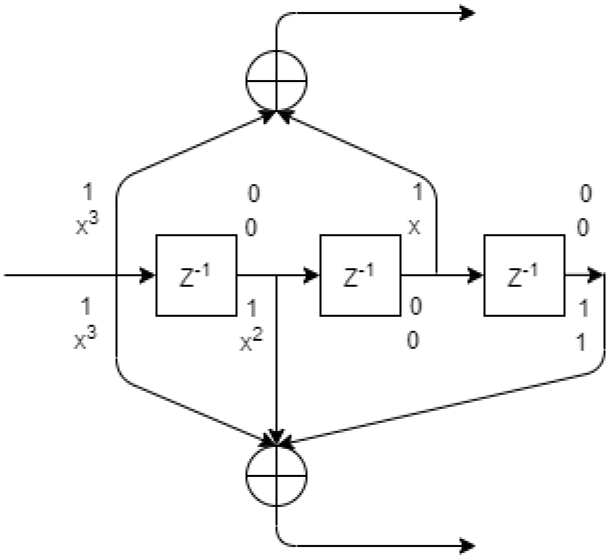
Verify the decoded data has zero bit errors.

```
biterr(data,decodedData)
```

```
ans = 0
```

Specifying Code Generators in Polynomial Form

Demonstrate alternative forms of specifying code generators for a trellis structure are equivalent.



Use a trellis structure to configure the rate 1/2 feedforward convolutional code in this diagram. The diagram indicates the binary values and polynomial form, indicating the left-most bit is the most-significant-bit (MSB).

Set the constraint length to 4. Use a cell array of polynomial character vectors to specify code generators. For more information, see “Character Representation of Polynomials”. When using character representation to specify the code generator, you can specify the polynomial in ascending or descending order, but the `poly2trellis` function always assigns registers in descending order with the left-most register for the MSB.

```
trellis_poly = poly2trellis(4,{'x3 + x','x3 + x2 + 1'})
```

```
trellis_poly = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 8
          nextStates: [8x2 double]
             outputs: [8x2 double]
```

The binary vector [1 0 1 0] represents octal 12 and corresponds to the upper row of binary digits in the diagram. The binary vector [1 1 0 1] represents octal 15 and corresponds to the lower row of binary digits in the diagram. Use octal representation to specify the code generators for an equivalent trellis structure.

```
trellis = poly2trellis(4,[12 15])
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 8
          nextStates: [8x2 double]
```

```
            outputs: [8x2 double]
```

Use `isequal` to confirm the two trellises are equal.

```
isequal(trellis,trellis_poly)
```

```
ans = logical
   1
```

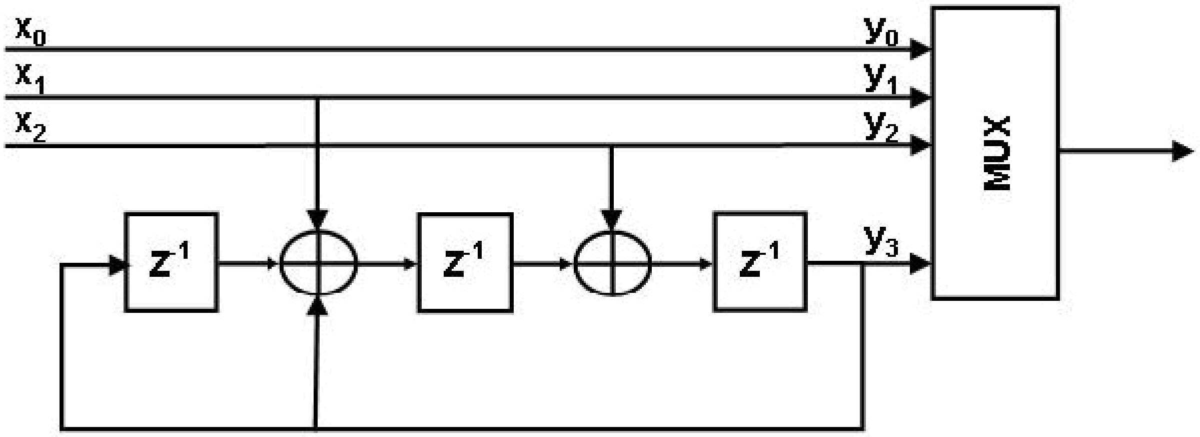
Create User Defined Trellis Structure

This example demonstrates creation of a nonstandard trellis structure for a convolutional encoder with uncoded bits and feedback. The encoder cannot be created using `poly2trellis` because the peculiar specifications for the encoder do not match the input requirements of `poly2trellis`.

You can manually create the trellis structure, and then use it as the input trellis structure for an encoder and decoder. The Convolutional Encoder and Viterbi Decoder blocks used in the "Convolutional Encoder with Uncoded Bits and Feedback" model load the trellis structure created here using a `PreLoadFcn` callback.

Convolutional Encoder

Create a rate 3/4 convolutional encoder with feedback connection whose MSB bit remains uncoded.



Declare variables according to the specifications.

```
K = 3;
N = 4;
constraintLength = 4;
```

Create trellis structure

A trellis is represented by a structure with the following fields:

- `numInputSymbols` – Number of input symbols

- `numOutputSymbols` – Number of output symbols
- `numStates` – Number of states
- `nextStates` – Next state matrix
- `outputs` – Output matrix

For more information about these structure fields, see `istrellis`.

Reset any previous occurrence of `myTrellis` structure.

```
clear myTrellis;
```

Define the trellis structure fields.

```
myTrellis.numInputSymbols = 2^K;
myTrellis.numOutputSymbols = 2^N;
myTrellis.numStates  = 2^(constraintLength-1);
```

Create nextStates Matrix

The `nextStates` matrix is a [`numStates` x `numInputSymbols`] matrix. The (*i*,*j*) element of the next state matrix is the resulting final state index that corresponds to a transition from the initial state *i* for an input equal to *j*.

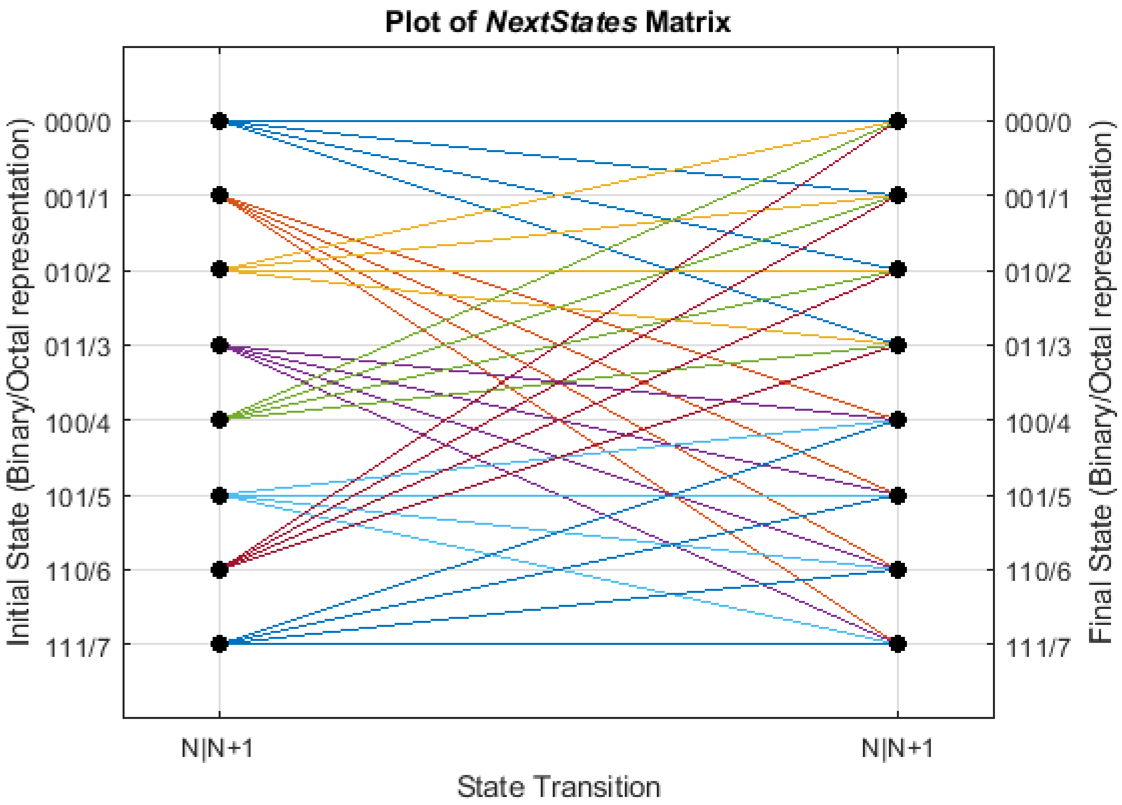
```
myTrellis.nextStates = [0  1  2  3  0  1  2  3; ...
                        6  7  4  5  6  7  4  5; ...
                        1  0  3  2  1  0  3  2; ...
                        7  6  5  4  7  6  5  4; ...
                        2  3  0  1  2  3  0  1; ...
                        4  5  6  7  4  5  6  7; ...
                        3  2  1  0  3  2  1  0; ...
                        5  4  7  6  5  4  7  6]
```

```
myTrellis = struct with fields:
     numInputSymbols: 8
    numOutputSymbols: 16
           numStates: 8
          nextStates: [8x8 double]
```

Plot nextStates Matrix

Use the `commcnv_plotnextstates` helper function to plot the `nextStates` matrix to illustrate the branch transitions between different states for a given input.

```
commcnv_plotnextstates(myTrellis.nextStates);
```



Create `outputs` Matrix

The `outputs` matrix is a [`numStates` x `numInputSymbols`] matrix. The (*i*,*j*) element of the output matrix is the output symbol in octal format given a current state *i* for an input equal to *j*.

```
outputs =  [0  2  4  6  10  12  14  16; ...
            1  3  5  7  11  13  15  17; ...
            0  2  4  6  10  12  14  16; ...
            1  3  5  7  11  13  15  17; ...
            0  2  4  6  10  12  14  16; ...
            1  3  5  7  11  13  15  17; ...
            0  2  4  6  10  12  14  16; ...
            1  3  5  7  11  13  15  17]
```

```
outputs = 8×8

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

Use `oct2dec` to display these values in decimal format.

```
outputs_dec = oct2dec(outputs)
```

```
outputs_dec = 8×8

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

Copy outputs matrix into the `myTrellis` structure.

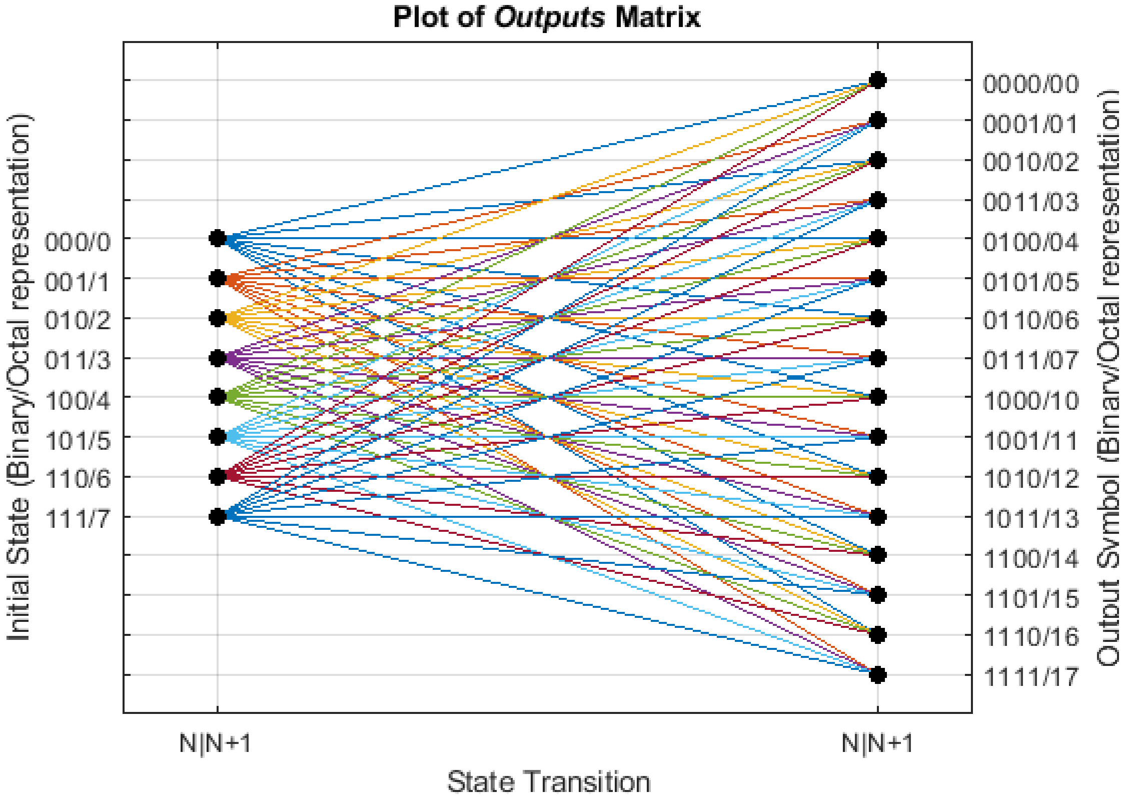
```
myTrellis.outputs = outputs
```

```
myTrellis = struct with fields:
     numInputSymbols: 8
    numOutputSymbols: 16
           numStates: 8
          nextStates: [8x8 double]
             outputs: [8x8 double]
```

Plot `outputs` Matrix

Use the `commcnv_plotoutputs` helper function to plot the `outputs` matrix to illustrate the possible output symbols for a given state depending on the input symbol.

```
commcnv_plotoutputs(myTrellis.outputs, myTrellis.numOutputSymbols);
```



Check Resulting Trellis Structure

```
istrellis(myTrellis)
```

```
ans = logical
   1
```

A return value of 1 confirms the trellis structure is valid.

Implement Soft-Decision Decoding

Decode with 3-bit soft decisions partitioned so that values near 0 map to 0, and values near 1 map to 7. If your application requires better decoding performance, refine the partition to obtain finer quantization.

The example decodes the code and computes the bit error rate. When comparing the decoded data with the original message, the example must take the decoding delay into account. The continuous operation mode of the Viterbi decoder causes a delay equal to the traceback length, so `msg(1)` corresponds to `decoded(tblen+1)` rather than to `decoded(1)`.

System Setup

Initialize runtime variables for the message data, trellis, bit error rate computations, and traceback length.

```
stream = RandStream.create('mt19937ar', 'seed',94384);
prevStream = RandStream.setGlobalStream(stream);
msg = randi([0 1],4000,1); % Random data

trellis = poly2trellis(7,[171 133]); % Define trellis

ber = zeros(3,1); % Store BER values
tblen = 48; % Traceback length
```

Create an AWGN channel System object, a Viterbi decoder System object, and an error rate calculator System object. Account for the receive delay caused by the traceback length of the Viterbi decoder.

```
awgnChan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)','SNR',6);
vitDec = comm.ViterbiDecoder(trellis,'InputFormat','Soft', ...
    'SoftInputWordLength',3,'TracebackDepth',tblen,'TerminationMethod','Continuous');
errorCalc = comm.ErrorRate('ReceiveDelay', tblen);
```

Run Coding and Decoding

Convolutionally code the message, pass in through an AWGN filter, quantize the noisy message for soft-decision decoding. Perform Viterbi decoding using the trellis generated using `poly2trellis`.

```
code = convenc(msg,trellis);
awgnChan.SignalPower = (code'*code)/length(code);
ncode = awgnChan(code);
```

Use `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.

```
qcode = quantiz(ncode,[0.001,0.1,0.3,0.5,0.7,0.9,0.999]);
decoded = vitDec(qcode);
```

Compute bit error rate.

```
ber = errorCalc(msg,decoded);
ratio = ber(1)
```

```
ratio = 0.0013
```

```
number = ber(2)
```

```
number = 5
```

```
RandStream.setGlobalStream(prevStream);
```

Input Arguments

`ConstraintLength` — Constraint length

row vector

Constraint length, specified as a 1-by-*K* row vector defining the delay for each of the *K* input bit streams to the encoder.

Data Types: `double`

`CodeGenerator` — Code generator

matrix | cell array of character vector | string array

Code generator, specified as a *K*-by-*N* matrix of octal numbers, a *K*-by-*N* cell array of polynomial character vectors, or a *K*-by-*N* string array. `CodeGenerator` specifies the *N* output connections for each of the *K* input bit streams to the encoder.

When using character representation to specify the code generator, you can specify the polynomial in ascending or descending order, but the `poly2trellis` function always assigns registers in descending order with the left-most register for the most-significant-bit (MSB). For more information, see “Specifying Code Generators in Polynomial Form” on page 2-662.

Data Types: `double` | `char` | `string`

`FeedbackConnection` — Feedback connection

row vector

Feedback connection, specified as a 1-by-*K* row vector of octal numbers defining the feedback connection for each of the *K* input bit streams to the encoder.

Data Types: `double`

Output Arguments

`trellis` — Trellis description

structure

Trellis description, returned as a structure with these fields. For more about this structure, see the `istrellis` function.

Trellis Structure Fields for Rate K/N Code

`numInputSymbols` — Number of input symbols

scalar

Number of input symbols, returned as a scalar with a value of 2^K. This value represents the number of input symbols to the encoder and *K* represents the number of input bit streams.

`numOutputSymbols` — Number of output symbols

scalar

Number of output symbols, returned as a scalar with a value of 2^N. This value represents the number of output symbols from the encoder and *N* represents the number of output bit streams.

`numStates` — Number of states

scalar

Number of states in the encoder, returned as a scalar.

`nextStates` — Next states

matrix

Next states for all combinations of current states and current inputs, returned as a `numStates`-by-2^K matrix, where *K* represents the number of input bit streams.

`outputs` — Outputs

matrix

Outputs for all combinations of current states and current inputs, returned as a `numStates`-by-2^K matrix, K represents the number of input bit streams. The elements of this matrix are octal numbers.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Inputs must be constants, of which there can be at most 3 (`ConstraintLength`, `CodeGenerator`, `FeedbackConnection`).

See Also

Functions

`convenc` | `istrellis` | `vitdec`

Topics

"Convolutional Codes"
"Character Representation of Polynomials"

Introduced before R2006a

primpoly

Find primitive polynomials for 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, `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 first example below illustrates the formats that `primpoly` uses in the Command Window and in the output argument `pr`. The subsequent examples illustrate the display options and the use of the *`opt`* argument.

```
pr = primpoly(4)

pr1 = primpoly(5,'max','nodisplay')

pr2 = primpoly(5,'min')

pr3 = primpoly(5,2)

pr4 = primpoly(5,3);
```

The output is below.

```
Primitive polynomial(s) =

D^4+D^1+1

pr =

    19


pr1 =

    61


Primitive polynomial(s) =

D^5+D^2+1

pr2 =

    37


No primitive polynomial satisfies the given constraints.

pr3 =

     []


Primitive polynomial(s) =

D^5+D^2+1
D^5+D^3+1
```

See Also

`gf` | `isprimitive`

Topics

"Galois Field Computations"

Introduced before R2006a

pskdemod

Phase shift keying demodulation

Syntax

```
z = pskdemod(y,M)
z = pskdemod(y,M,ini_phase)
z = pskdemod(y,M,ini_phase,symorder)
```

Description

`z = pskdemod(y,M)` demodulates the complex envelope, `y`, of a PSK-modulated signal having modulation order `M`.

`z = pskdemod(y,M,ini_phase)` specifies the initial phase of the PSK-modulated signal.

`z = pskdemod(y,M,ini_phase,symorder)` specifies the symbol order of the PSK-modulated signal.

Examples

Compare Phase Noise Effects on PSK and PAM Signals

Compare PSK and PAM modulation schemes to demonstrate that PSK is more sensitive to phase noise. This is the expected result because the PSK constellation is circular while the PAM constellation is linear.

Specify the number of symbols and the modulation order parameters. Generate random data symbols.

```
len = 10000;
M = 16;
msg = randi([0 M-1],len,1);
```

Modulate `msg` using both PSK and PAM to compare the two methods.

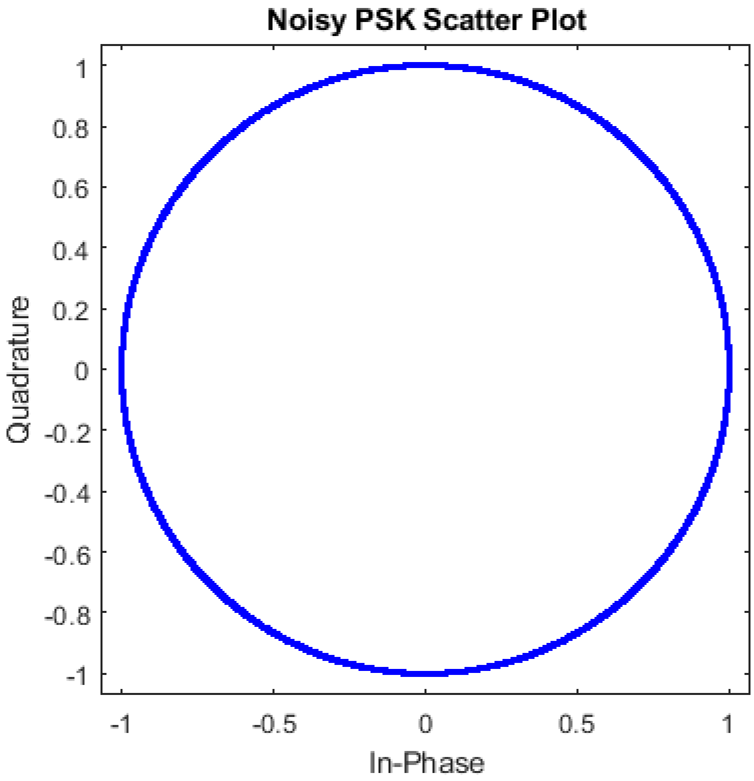
```
txpsk = pskmod(msg,M);
txpam = pammod(msg,M);
```

Perturb the phase of the modulated signals by applying a random phase rotation.

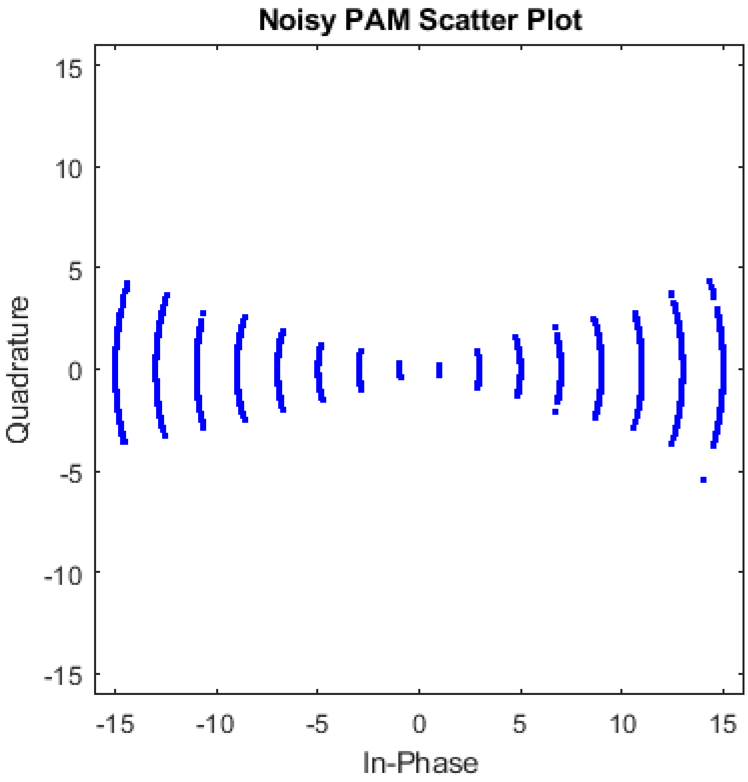
```
phasenoise = randn(len,1)*.015;
rxpsk = txpsk.*exp(2i*pi*phasenoise);
rxpam = txpam.*exp(2i*pi*phasenoise);
```

Create scatter plots of the received signals.

```
scatterplot(rxpsk);
title('Noisy PSK Scatter Plot')
```



```
scatterplot(rxpam);
title('Noisy PAM Scatter Plot')
```



Demodulate the received signals.

```
recovpsk = pskdemod(rxpsk,M);
recovpam = pamdemod(rxpam,M);
```

Compute the number of symbol errors for each modulation scheme. The PSK signal experiences a much greater number of symbol errors.

```
numerrs_psk = symerr(msg,recovpsk);
numerrs_pam = symerr(msg,recovpam);
[numerrs_psk numerrs_pam]
```

```
ans = 1×2

   343      1
```

Modulate and Demodulate QPSK Signal in AWGN

Generate random symbols.

```
dataIn = randi([0 3],1000,1);
```

QPSK modulate the data.

```
txSig = pskmod(dataIn,4,pi/4);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,10);
```

Demodulate the received signal and compute the number of symbol errors.

```
dataOut = pskdemod(rxSig,4,pi/4);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 2
```

PSK Symbol Mapping

Plot PSK symbol mapping for Gray and natural binary encoded data.

Set the modulation order, and then create a data sequence containing a complete set of constellation points.

```
M = 8;
data = (0:M-1);
phz = 0;
```

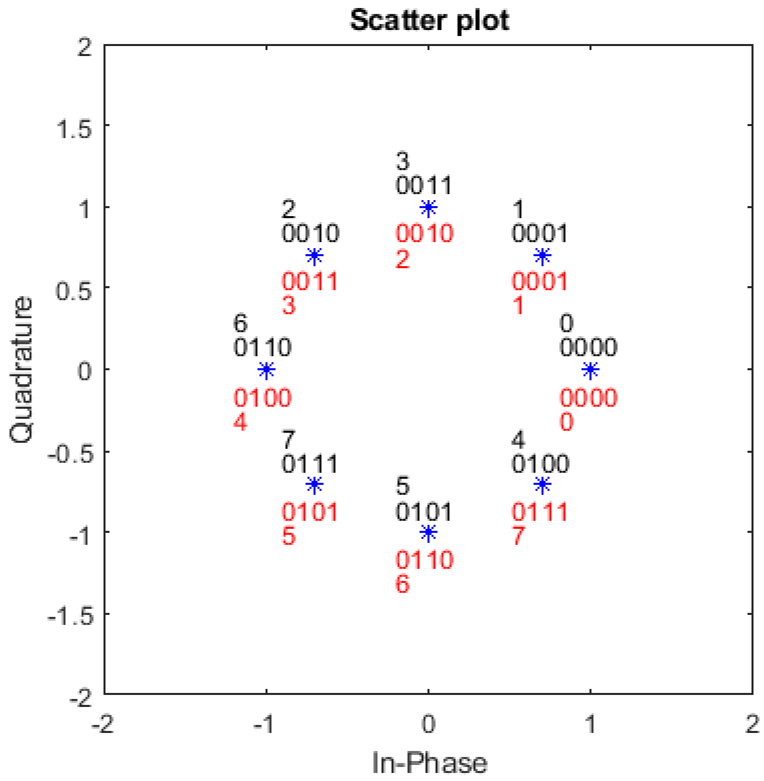
Modulate and demodulate the data using Gray and natural binary encoded data.

```
symgray = pskmod(data,M,phz,'gray');
mapgray = pskdemod(symgray,M,phz,'gray');

symbin = pskmod(data,M,phz,'bin');
mapbin = pskdemod(symbin,M,phz,'bin');
```

Plot the constellation points using one of the symbol sets. For each constellation point, assign a label indicating the Gray and natural binary values for each symbol.

- For Gray binary symbol mapping, adjacent constellation points differ by a single binary bit and are not numerically sequential.
- For natural binary symbol mapping, adjacent constellation points follow the natural binary encoding and are sequential.

```
scatterplot(symgray,1,0,'b*');
for k = 1:M
    text(real(symgray(k))-0.2,imag(symgray(k))+.15,...
        dec2base(mapgray(k),2,4));
     text(real(symgray(k))-0.2,imag(symgray(k))+.3,...
         num2str(mapgray(k)));

    text(real(symbin(k))-0.2,imag(symbin(k))-.15,...
        dec2base(mapbin(k),2,4),'Color',[1 0 0]);
    text(real(symbin(k))-0.2,imag(symbin(k))-.3,...
        num2str(mapbin(k)),'Color',[1 0 0]);
end
axis([-2 2 -2 2])
```



Input Arguments

y — PSK-modulated input signal

vector | matrix

PSK-modulated input signal, specified as a real or complex vector or matrix. If y is a matrix, the function processes the columns independently.

Data Types: double
Complex Number Support: Yes

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double

ini_phase — Initial phase

0 (default) | scalar | [ ]

Initial phase of the PSK modulation, specified in radians as a real scalar.

If ini_phase is empty, then pskdemod uses an initial phase of 0.

Example: `pi/4`

Data Types: `double`

`symorder` — Symbol order

`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

Output Arguments

`z` — PSK-demodulated output signal

vector | matrix

PSK-demodulated output signal, returned as a vector or matrix having the same number of columns as input signal `y`.

See Also

`comm.PSKDemodulator` | `modnorm` | `pskmod`

Topics

"Phase Modulation"

Introduced before R2006a

pskmod

Phase shift keying modulation

Syntax

```
y = pskmod(x,M)
y = pskmod(x,M,ini_phase)
y = pskmod(x,M,ini_phase,symorder)
```

Description

`y = pskmod(x,M)` modulates the input signal, `x`, using phase shift keying (PSK) with modulation order `M`.

`y = pskmod(x,M,ini_phase)` specifies the initial phase of the PSK-modulated signal.

`y = pskmod(x,M,ini_phase,symorder)` specifies the symbol order of the PSK-modulated signal.

Examples

Modulate PSK Signal

Modulate and plot the constellations of QPSK and 16-PSK signals.

QPSK

Set the modulation order to 4.

```
M = 4;
```

Generate random data symbols.

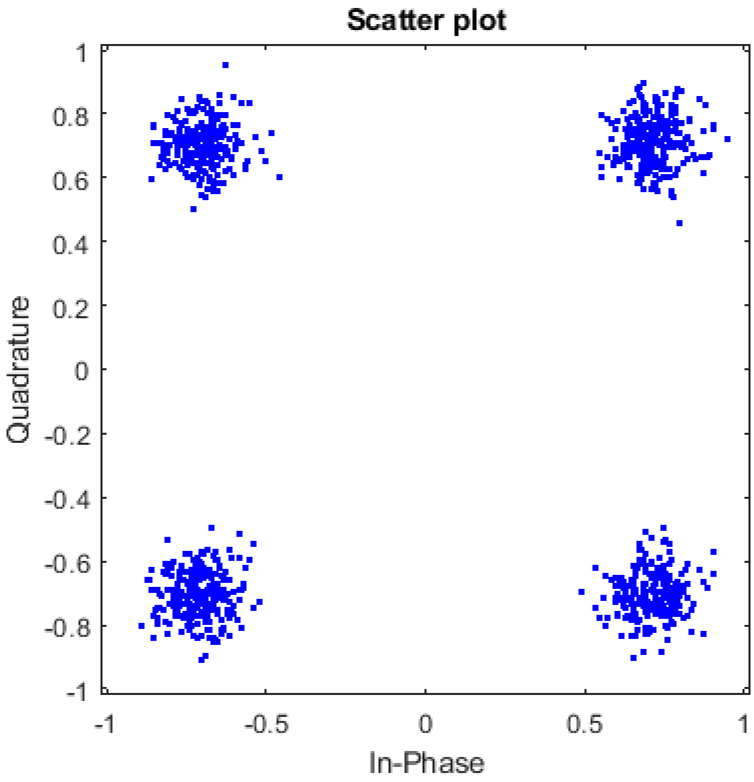
```
data = randi([0 M-1],1000,1);
```

Modulate the data symbols.

```
txSig = pskmod(data,M,pi/M);
```

Pass the signal through white noise and plot its constellation.

```
rxSig = awgn(txSig,20);
scatterplot(rxSig)
```



16-PSK

Change the modulation order from 4 to 16.

```
M = 16;
```

Generate random data symbols.

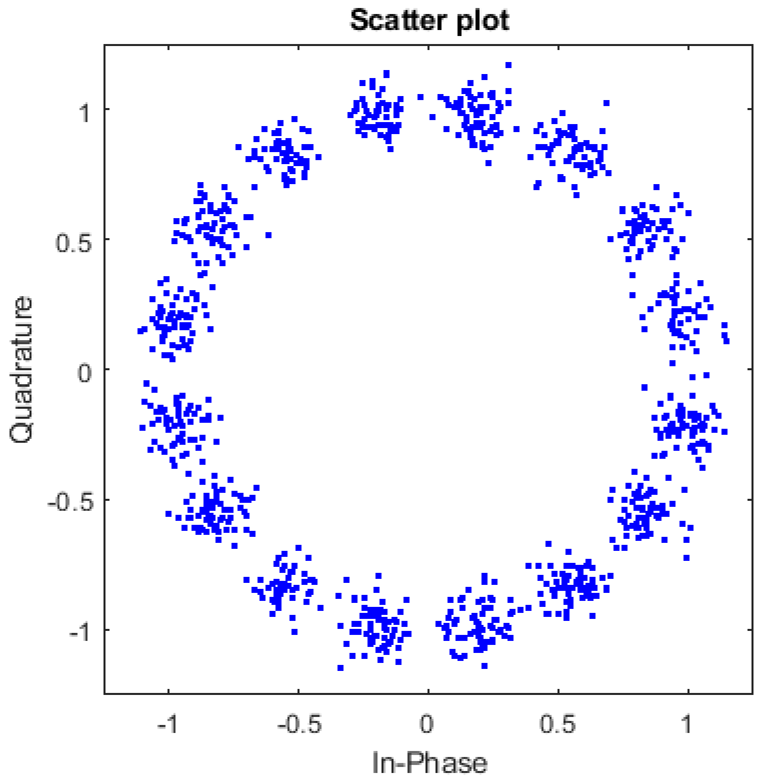
```
data = randi([0 M-1],1000,1);
```

Modulate the data symbols.

```
txSig = pskmod(data,M,pi/M);
```

Pass the signal through white noise and plot its constellation.

```
rxSig = awgn(txSig,20);
scatterplot(rxSig)
```



Modulate and Demodulate QPSK Signal in AWGN

Generate random symbols.

```
dataIn = randi([0 3],1000,1);
```

QPSK modulate the data.

```
txSig = pskmod(dataIn,4,pi/4);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,10);
```

Demodulate the received signal and compute the number of symbol errors.

```
dataOut = pskdemod(rxSig,4,pi/4);
numErrs = symerr(dataIn,dataOut)
```

```
numErrs = 2
```

PSK Symbol Mapping

Plot PSK symbol mapping for Gray and natural binary encoded data.

Set the modulation order, and then create a data sequence containing a complete set of constellation points.

```
M = 8;
data = (0:M-1);
phz = 0;
```

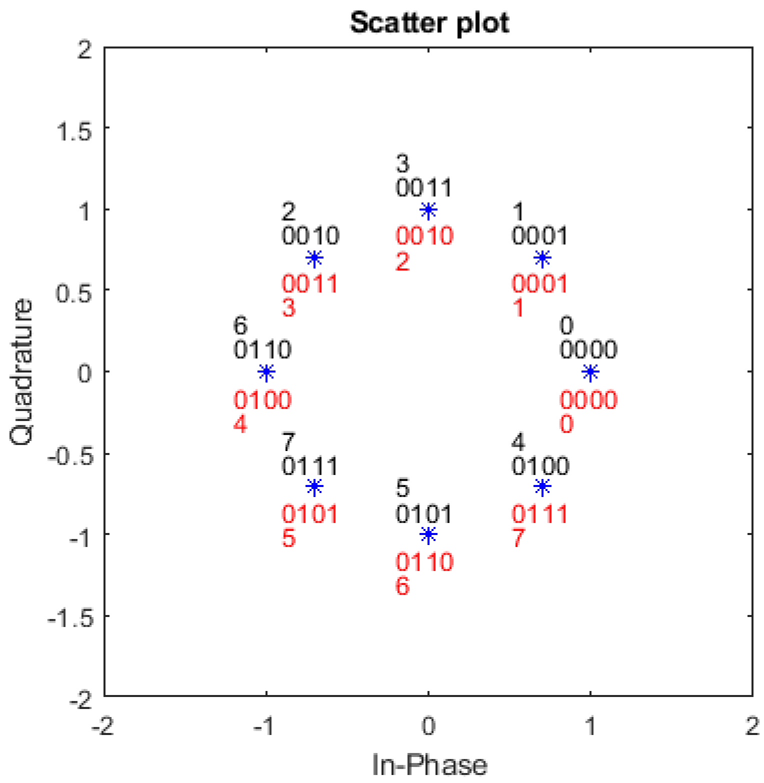
Modulate and demodulate the data using Gray and natural binary encoded data.

```
symgray = pskmod(data,M,phz,'gray');
mapgray = pskdemod(symgray,M,phz,'gray');

symbin = pskmod(data,M,phz,'bin');
mapbin = pskdemod(symbin,M,phz,'bin');
```

Plot the constellation points using one of the symbol sets. For each constellation point, assign a label indicating the Gray and natural binary values for each symbol.

- For Gray binary symbol mapping, adjacent constellation points differ by a single binary bit and are not numerically sequential.
- For natural binary symbol mapping, adjacent constellation points follow the natural binary encoding and are sequential.

```
scatterplot(symgray,1,0,'b*');
for k = 1:M
    text(real(symgray(k))-0.2,imag(symgray(k))+.15,...
        dec2base(mapgray(k),2,4));
     text(real(symgray(k))-0.2,imag(symgray(k))+.3,...
         num2str(mapgray(k)));

    text(real(symbin(k))-0.2,imag(symbin(k))-.15,...
        dec2base(mapbin(k),2,4),'Color',[1 0 0]);
    text(real(symbin(k))-0.2,imag(symbin(k))-.3,...
        num2str(mapbin(k)),'Color',[1 0 0]);
end
axis([-2 2 -2 2])
```



Input Arguments

x — Input signal

vector | matrix

Input signal, specified as a vector or matrix of positive integers. The elements of x must have values in the range of [0, M – 1].

Example: randi([0 3],100,1)

Data Types: double

M — Modulation order

integer power of two

Modulation order, specified as an integer power of two.

Example: 2 | 4 | 16

Data Types: double

ini_phase — Initial phase

0 (default) | scalar | [ ]

Initial phase of the PSK modulation, specified in radians as a real scalar.

If you specify ini_phase as empty, then pskmod uses an initial phase of 0.

Example: `pi/4`

Data Types: `double`

`symorder` — Symbol order

`'bin'` (default) | `'gray'`

Symbol order, specified as `'bin'` or `'gray'`. This argument specifies how the function assigns binary vectors to corresponding integers.

- If `symorder` is `'bin'`, the function uses a natural binary-coded ordering.
- If `symorder` is `'gray'`, the function uses a Gray-coded ordering.

Data Types: `char`

Output Arguments

y — PSK-modulated output signal

vector | matrix

Complex baseband representation of a PSK-modulated signal, returned as vector or matrix of complex values. The columns of `y` represent independent channels.

See Also

`comm.PSKModulator` | `modnorm` | `pskdemod`

Topics

“Phase Modulation”

Introduced before R2006a

qamdemod

Quadrature amplitude demodulation

Syntax

```
z = qamdemod(y,M)
z = qamdemod(y,M,symOrder)
z = qamdemod( ___ ,Name,Value)
```

Description

`z = qamdemod(y,M)` returns a demodulated signal, `z`, given quadrature amplitude modulation (QAM) signal `y` of modulation order `M`.

`z = qamdemod(y,M,symOrder)` returns a demodulated signal, `z`, and specifies the symbol order for the demodulation.

`z = qamdemod( ___ ,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'OutputType','bit'` sets the type of output signal to bits.

Examples

Demodulate 8-QAM Signal

Demodulate an 8-QAM signal and plot the points corresponding to symbols 0 and 3.

Generate random 8-ary data symbols.

```
data = randi([0 7],1000,1);
```

Modulate `data` by applying 8-QAM.

```
txSig = qammod(data,8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,18,'measured');
```

Demodulate the received signal using an initial phase of $\pi/8$.

```
rxData = qamdemod(rxSig.*exp(-1i*pi/8),8);
```

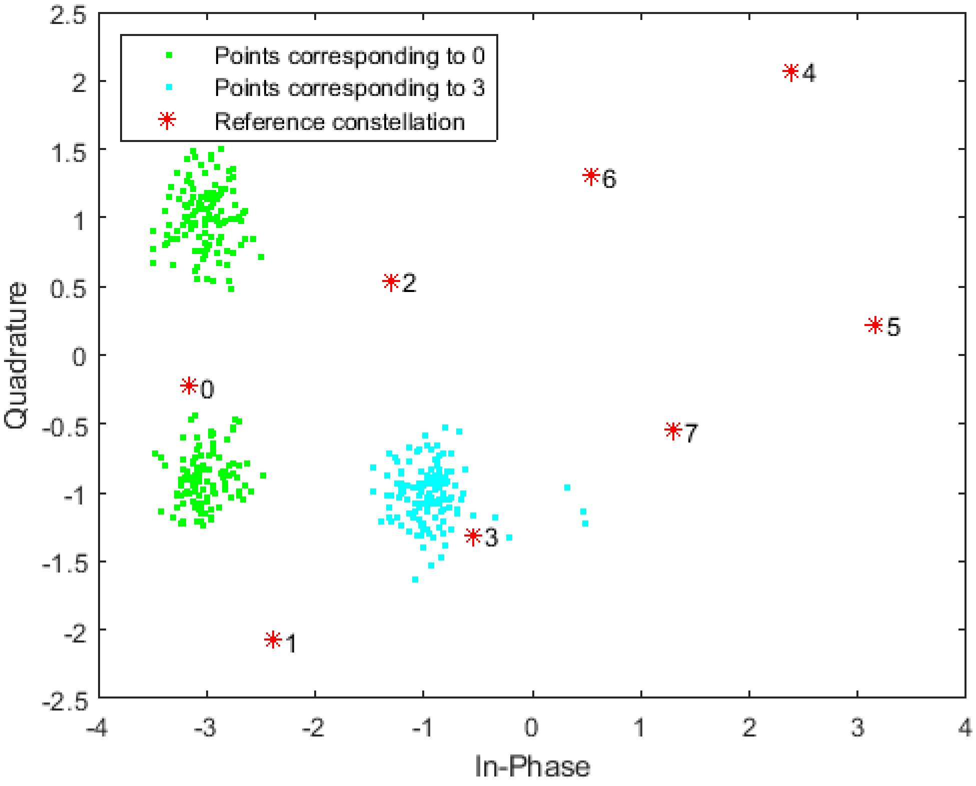
Generate the reference constellation points.

```
refpts = qammod((0:7)',8) .* exp(1i*pi/8);
```

Plot the received signal points corresponding to symbols 0 and 3 and overlay the reference constellation. The received data corresponding to those symbols is displayed.

```
plot(rxSig(rxData==0),'g.');
hold on
```

```
plot(rxSig(rxData==3),'c.');
plot(refpts,'r*')
text(real(refpts)+0.1,imag(refpts),num2str((0:7)'))
xlabel('In-Phase')
ylabel('Quadrature')
legend('Points corresponding to 0','Points corresponding to 3', ...
    'Reference constellation','location','nw');
```



QAM Demodulation with WLAN Symbol Mapping

Modulate and demodulate random data by using 16-QAM with WLAN symbol mapping. Verify that the input data symbols match the demodulated symbols.

Generate a 3-D array of random symbols.

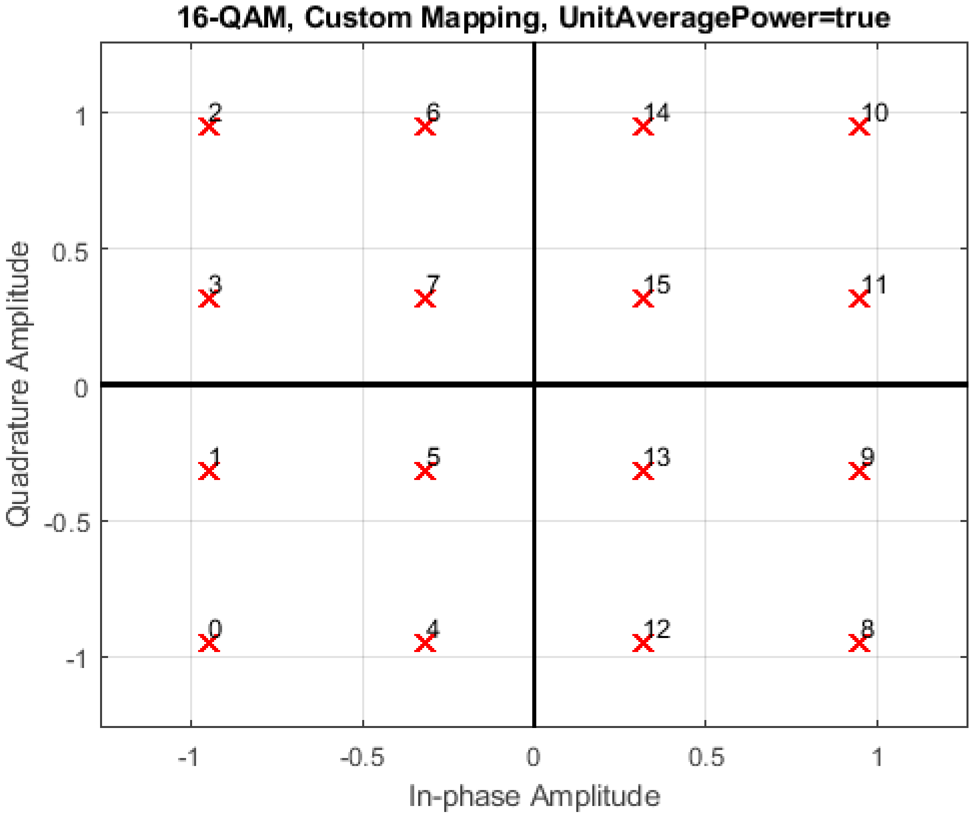
```
x = randi([0,15],20,4,2);
```

Create a custom symbol mapping for the 16-QAM constellation based on WLAN standards.

```
wlanSymMap = [2 3 1 0 6 7 5 4 14 15 13 12 10 11 9 8];
```

Modulate the data, and set the constellation to have unit average signal power. Plot the constellation.

```
y = qammod(x,16,wlanSymMap,'UnitAveragePower', true,'PlotConstellation',true);
```



Demodulate the received signal.

```
z = qamdemod(y,16,wlanSymMap,'UnitAveragePower',true);
```

Verify that the demodulated signal is equal to the original data.

```
isequal(x,z)
```

```
ans = logical
   1
```

Demodulate QAM Fixed-Point Signal

Demodulate a fixed-point QAM signal and verify that the data is recovered correctly.

Set the modulation order as 64, and determine the number of bits per symbol.

```
M = 64;
bitsPerSym = log2(M);
```

Generate random bits. When operating in bit mode, the length of the input data must be an integer multiple of the number of bits per symbol.

```
x = randi([0 1],10*bitsPerSym,1);
```

Modulate the input data using a binary symbol mapping. Set the modulator to output fixed-point data. The numeric data type is signed with a 16-bit word length and a 10-bit fraction length.

```
y = qammod(x,M,'bin','InputType','bit','OutputDataType', ...
    numerictype(1,16,10));
```

Demodulate the 64-QAM signal. Verify that the demodulated data matches the input data.

```
z = qamdemod(y,M,'bin','OutputType','bit');
s = isequal(x,double(z))
```

```
s = logical
   1
```

Estimate BER for Hard and Soft Decision Viterbi Decoding

Estimate bit error rate (BER) performance for hard-decision and soft-decision Viterbi decoders in AWGN. Compare the performance to that of an uncoded 64-QAM link.

Set the simulation parameters.

```
clear; close all
rng default
M = 64;                 % Modulation order
k = log2(M);            % Bits per symbol
EbNoVec = (4:10)';      % Eb/No values (dB)
numSymPerFrame = 1000;  % Number of QAM symbols per frame
```

Initialize the BER results vectors.

```
berEstSoft = zeros(size(EbNoVec));
berEstHard = zeros(size(EbNoVec));
```

Set the trellis structure and traceback depth for a rate `1/2`, constraint length `7`, convolutional code.

```
trellis = poly2trellis(7,[171 133]);
tbl = 32;
rate = 1/2;
```

The main processing loops performs these steps:

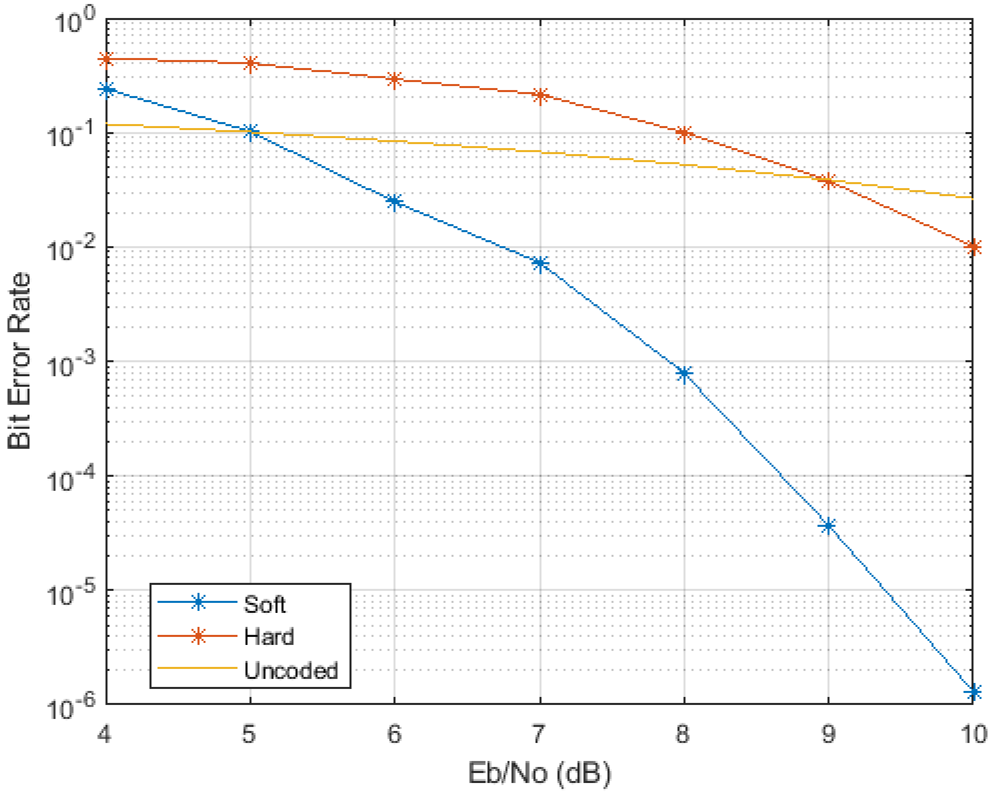
- Generate binary data
- Convolutionally encode the data
- Apply QAM modulation to the data symbols. Specify unit average power for the transmitted signal
- Pass the modulated signal through an AWGN channel
- Demodulate the received signal using hard decision and approximate LLR methods. Specify unit average power for the received signal
- Viterbi decode the signals using hard and unquantized methods
- Calculate the number of bit errors

The `while` loop continues to process data until either 100 errors are encountered or 10^7 bits are transmitted.

```
for n = 1:length(EbNoVec)
    % Convert Eb/No to SNR
    snrdB = EbNoVec(n) + 10*log10(k*rate);
    % Noise variance calculation for unity average signal power.
    noiseVar = 10.^(-snrdB/10);
    % Reset the error and bit counters
    [numErrsSoft,numErrsHard,numBits] = deal(0);

    while numErrsSoft < 100 && numBits < 1e7
        % Generate binary data and convert to symbols
        dataIn = randi([0 1],numSymPerFrame*k,1);

        % Convolutionally encode the data
        dataEnc = convenc(dataIn,trellis);

        % QAM modulate
        txSig = qammod(dataEnc,M,'InputType','bit','UnitAveragePower',true);

        % Pass through AWGN channel
        rxSig = awgn(txSig,snrdB,'measured');

        % Demodulate the noisy signal using hard decision (bit) and
        % soft decision (approximate LLR) approaches.
        rxDataHard = qamdemod(rxSig,M,'OutputType','bit','UnitAveragePower',true);
        rxDataSoft = qamdemod(rxSig,M,'OutputType','approxllr', ...
            'UnitAveragePower',true,'NoiseVariance',noiseVar);

        % Viterbi decode the demodulated data
        dataHard = vitdec(rxDataHard,trellis,tbl,'cont','hard');
        dataSoft = vitdec(rxDataSoft,trellis,tbl,'cont','unquant');

        % Calculate the number of bit errors in the frame. Adjust for the
        % decoding delay, which is equal to the traceback depth.
        numErrsInFrameHard = biterr(dataIn(1:end-tbl),dataHard(tbl+1:end));
        numErrsInFrameSoft = biterr(dataIn(1:end-tbl),dataSoft(tbl+1:end));

        % Increment the error and bit counters
        numErrsHard = numErrsHard + numErrsInFrameHard;
        numErrsSoft = numErrsSoft + numErrsInFrameSoft;
        numBits = numBits + numSymPerFrame*k;

    end

    % Estimate the BER for both methods
    berEstSoft(n) = numErrsSoft/numBits;
    berEstHard(n) = numErrsHard/numBits;
end
```

Plot the estimated hard and soft BER data. Plot the theoretical performance for an uncoded 64-QAM channel.

```
semilogy(EbNoVec,[berEstSoft berEstHard],'-*')
hold on
semilogy(EbNoVec,berawgn(EbNoVec,'qam',M))
legend('Soft','Hard','Uncoded','location','best')
grid
xlabel('Eb/No (dB)')
ylabel('Bit Error Rate')
```



As expected, the soft decision decoding produces the best results.

Soft-Decision OQPSK Modulation-Demodulation

Use the `qamdemod` function to simulate soft decision output for OQPSK-modulated signals.

Generate an OQPSK modulated signal.

```
sps = 4;
msg = randi([0 1],1000,1);
oqpskMod = comm.OQPSKModulator('SamplesPerSymbol',sps,'BitInput',true);
oqpskSig = oqpskMod(msg);
```

Add noise to the generated signal.

```
impairedSig = awgn(oqpskSig,15);
```

Perform Soft-Decision Demodulation

Create QPSK equivalent signal to align in-phase and quadrature.

```
impairedQPSK = complex(real(impairedSig(1+sps/2:end-sps/2)), imag(impairedSig(sps+1:end)));
```

Apply matched filtering to the received OQPSK signal.

```
halfSinePulse = sin(0:pi/sps:(sps)*pi/sps);
matchedFilter = dsp.FIRDecimator(sps,halfSinePulse,'DecimationOffset',sps/2);
filteredQPSK = matchedFilter(impairedQPSK);
```

To perform soft demodulation of the filtered OQPSK signal use the `qamdemod` function. Align symbol mapping of `qamdemod` with the symbol mapping used by the `comm.OQPSKModulator`, then demodulate the signal.

```
oqpskModSymbolMapping = [1 3 0 2];
demodulated = qamdemod(filteredQPSK,4,oqpskModSymbolMapping,'OutputType','llr');
```

Input Arguments

y — Input signal

scalar | vector | matrix | 3-D array

Input signal that resulted QAM, specified as a scalar, vector, matrix, or 3-D array of complex values. Each column in the matrix and 3-D array is considered as an independent channel.

Data Types: `single` | `double` | `fi`
Complex Number Support: Yes

`M` — Modulation order

scalar integer

Modulation order, specified as a power-of-two scalar integer. The modulation order specifies the number of points in the signal constellation.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

`symOrder` — Symbol order

`'gray'` (default) | `'bin'` | vector

Symbol order, specified as one of these options:

- `'gray'` — Use “Gray Code” on page 2-694 ordering.
- `'bin'` — Use natural binary-coded ordering.
- Vector — Use custom symbol ordering. The vector must be of length `M`. Vectors must use unique elements whose values range from 0 to `M` – 1. The first element corresponds to the upper left point of the constellation, with subsequent elements running down column-wise from left to right.

Data Types: `char` | `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `z = qamdemod(y,M,symOrder,'OutputType','bit')`

`UnitAveragePower` — Unit average power flag

`false` or `0` (default) | `true` or `1`

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a numeric or logical `0` (`false`) or `1` (`true`). When this flag is `1` (`true`), the function scales the constellation to the average power of one watt referenced to 1 ohm. When this flag is 0 (false), the function scales the constellation so that the QAM constellation points are separated by a minimum distance of two.

OutputType — Type of output

'integer' (default) | 'bit' | 'llr' | 'approxllr'

Type of output, specified as the comma-separated pair consisting of 'OutputType' and 'integer', 'bit', 'llr', or 'approxllr'.

Data Types: char

NoiseVariance — Noise variance

1 (default) | positive scalar | vector of positive values

Noise variance, specified as the comma-separated pair consisting of 'NoiseVariance' and one of these options:

- Positive scalar — The same noise variance value is used on all input elements.
- Vector of positive values — The vector length must be equal to the number of elements in the last dimension of the input signal. Each element of the vector specifies the noise variance for all the elements of the input along the corresponding last dimension.

Tip When 'OutputType' is ‘llr’, if the demodulation computation outputs Inf or -Inf value, it is likely because the specified noise variance values are smaller than the signal-to-noise ratio (SNR). Because the Log-Likelihood algorithm computes exponentials using finite precision arithmetic, the computation of exponentials with large or small numbers can yield positive or negative infinity. For more details, see “Exact LLR Algorithm”.

To avoid this issue, set 'OutputType' to ‘approxllr’ instead. The approximate LLR algorithm does not compute exponentials.

Dependencies

To enable this name-value pair argument, set 'OutputType' is 'llr' or 'approxllr'.

Data Types: double

PlotConstellation — Option to plot constellation

false or 0 (default) | true or 1

Option to plot constellation, specified as the comma-separated pair consisting of 'PlotConstellation' and a numeric or logical 0 (false) or 1 (true) To plot the QAM constellation, set 'PlotConstellation' to true.

Output Arguments

z — Demodulated output signal

scalar | vector | matrix | 3-D array

Demodulated output signal, returned as a scalar, vector, matrix, or 3-D array. The data type is the same as that of the input signal, y. The value and dimension of this output vary depending on the specified 'OutputType' value, as shown in this table.

'OutputType'	Return Value of qamdemod	Dimensions of Output
'integer'	Demodulated integer values from 0 to (M – 1)	z has the same dimensions as input y.
'bit'	Demodulated bits	The number of rows in z is $\log_2$(M) times the number of rows in y. Each demodulated symbol is mapped to a group of $\log_2$(M) bits, where the first bit represents the most significant bit (MSB) and the last bit represents the least significant bit (LSB).
'llr'	Log-likelihood ratio value for each bit calculated using the Exact Log Likelihood algorithm. For more details, see “Exact LLR Algorithm”.	
'approxllr'	Approximate log-likelihood ratio value for each bit. The values are calculated using the Approximate Log Likelihood algorithm. For more details, see “Approximate LLR Algorithm”.	

More About

Gray Code

A Gray code, also known as a reflected binary code, is a system where the bit patterns in adjacent constellation points differ by only one bit.

Compatibility Considerations

Initial Phase Input Removed

Errors starting in R2018b

Starting in R2018b, you can no longer offset the initial phase for the QAM constellation using the qamdemod function.

Instead, use the genqamdemod function to offset the initial phase of the QAM signal being demodulated. Alternatively, you can multiply the modulated input of qamdemod by the desired initial phase, as shown in this code

```
z = qamdemod(y.*exp(-1i*initPhase,M))
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

genqamdemod | genqammod | modnorm | pamdemod | qammod

Topics

“Compute Symbol Error Rate”
“Exact LLR Algorithm”

"Approximate LLR Algorithm"

Introduced before R2006a

qammod

Quadrature amplitude modulation (QAM)

Syntax

```
y = qammod(x,M)
y = qammod(x,M,symOrder)
y = qammod( ___ ,Name,Value)
```

Description

`y = qammod(x,M)` modulates input signal `x` by using QAM with the specified modulation order `M`. Output `y` is the modulated signal.

`y = qammod(x,M,symOrder)` specifies the symbol order.

`y = qammod( ___ ,Name,Value)` specifies options using name-value pair arguments in addition to any of the input argument combinations from previous syntaxes.

Examples

Modulate Data Using QAM

Modulate data using QAM and display the result in a scatter plot.

Set the modulation order to 16 and create a data vector containing each of the possible symbols.

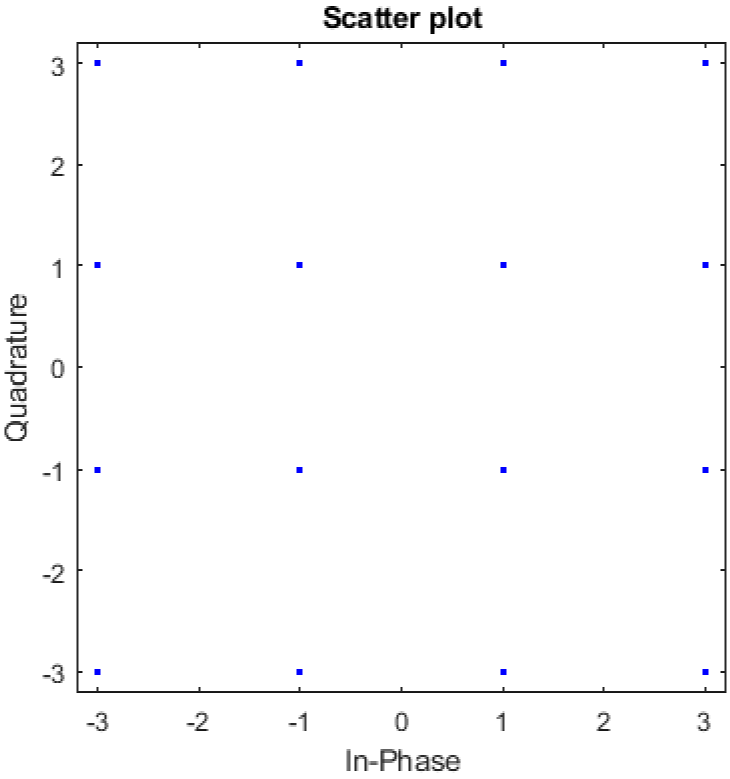
```
M = 16;
x = (0:M-1)';
```

Modulate the data using the `qammod` function.

```
y = qammod(x,M);
```

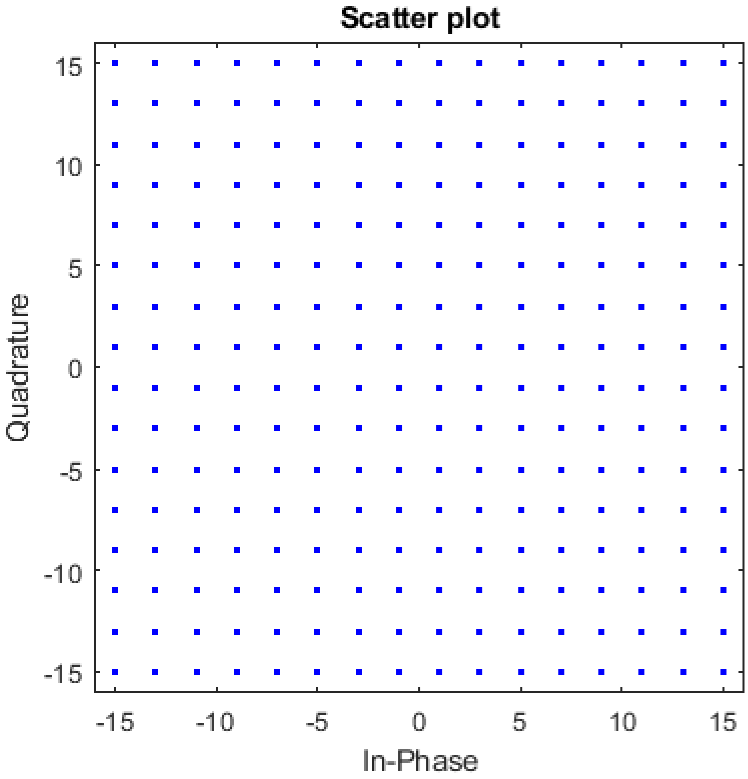
Display the modulated signal constellation using the `scatterplot` function.

```
scatterplot(y)
```



Set the modulation order to 256, and display the scatter plot of the modulated signal.

```
M = 256;
x = (0:M-1)';
y = qammod(x,M);
scatterplot(y)
```



Normalize QAM Signal by Average Power

Modulate random data symbols using QAM. Normalize the modulator output so that it has an average signal power of 1 W.

Set the modulation order and generate random data.

```
M = 64;
x = randi([0 M-1],1000,1);
```

Modulate the data. Use the `'UnitAveragePower'` name-value pair to set the output signal to have an average power of 1 W.

```
y = qammod(x,M,'UnitAveragePower',true);
```

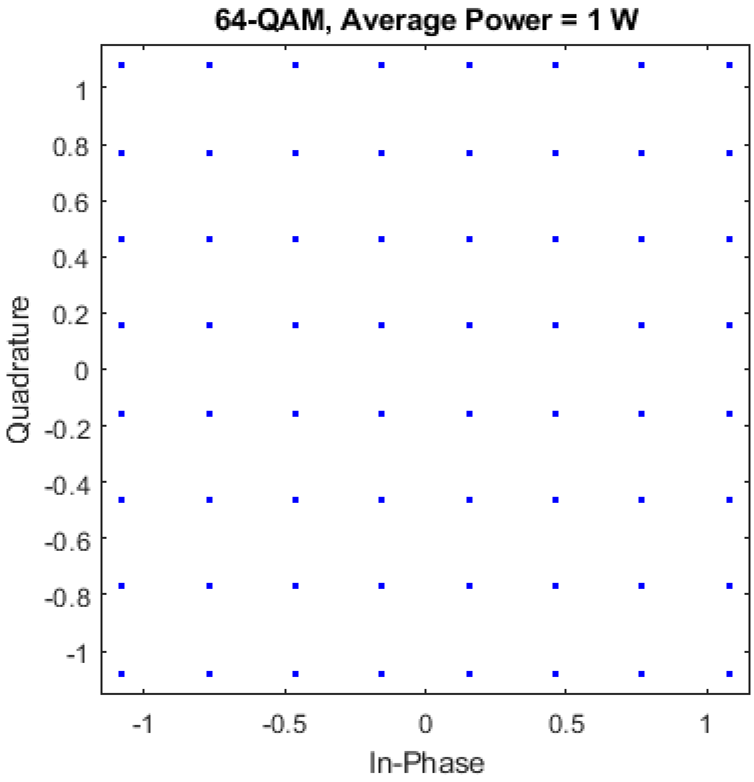
Confirm that the signal has unit average power.

```
avgPower = mean(abs(y).^2)
```

```
avgPower = 1.0070
```

Plot the resulting constellation.

```
scatterplot(y)
title('64-QAM, Average Power = 1 W')
```



QAM Symbol Ordering

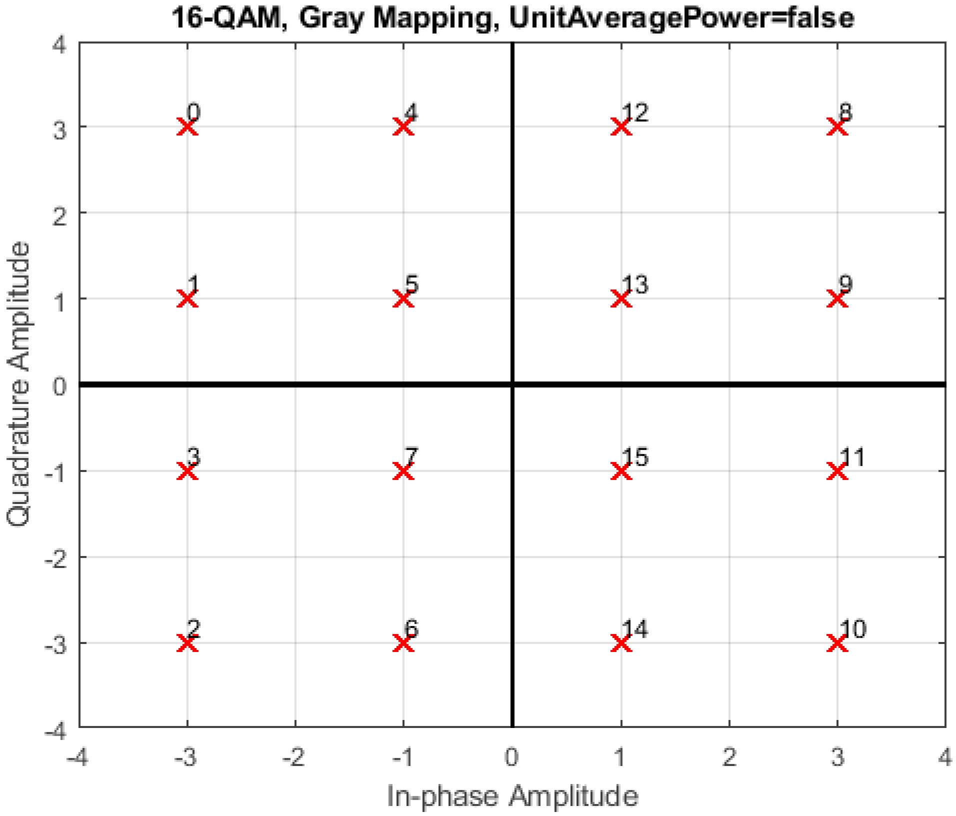
Plot QAM constellations for Gray, binary, and custom symbol mappings.

Set the modulation order, and create a data sequence that includes a complete set of symbols for the modulation scheme.

```
M = 16;
d = [0:M-1];
```

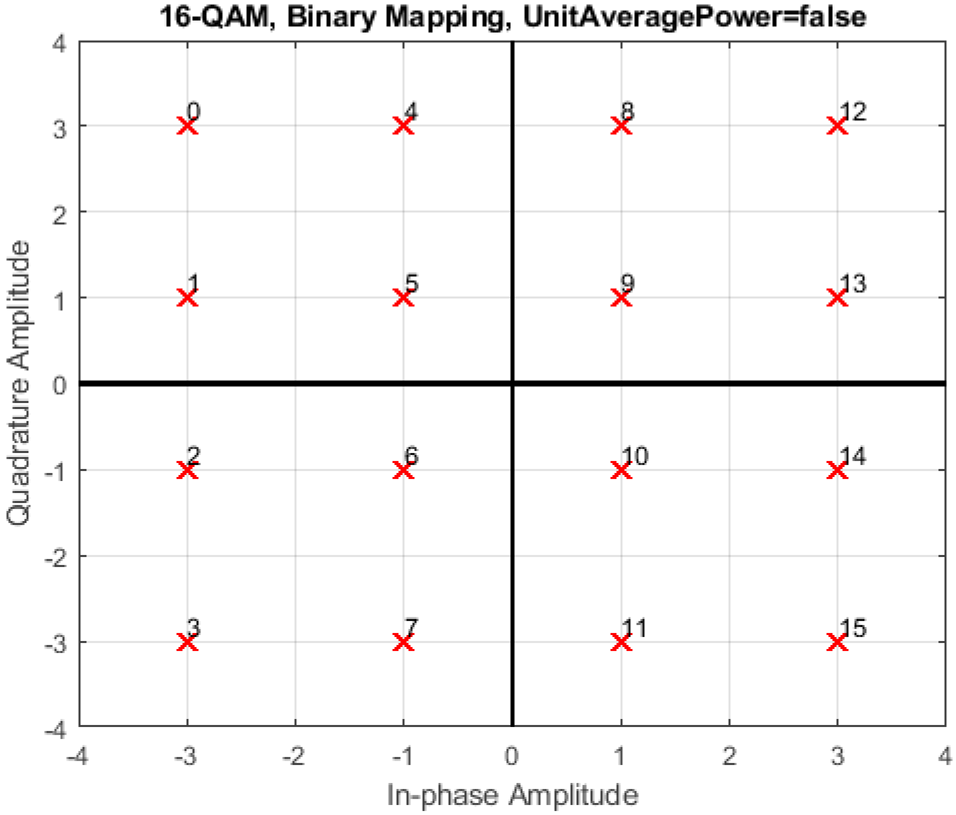
Modulate the data, and plot its constellation. The default symbol mapping uses Gray ordering. The ordering of the points is not sequential.

```
y = qammod(d,M,'PlotConstellation',true);
```



Repeat the modulation process with binary symbol mapping. The symbol mapping follows a natural binary order and is sequential.

```
z = qammod(d,M,'bin','PlotConstellation',true);
```

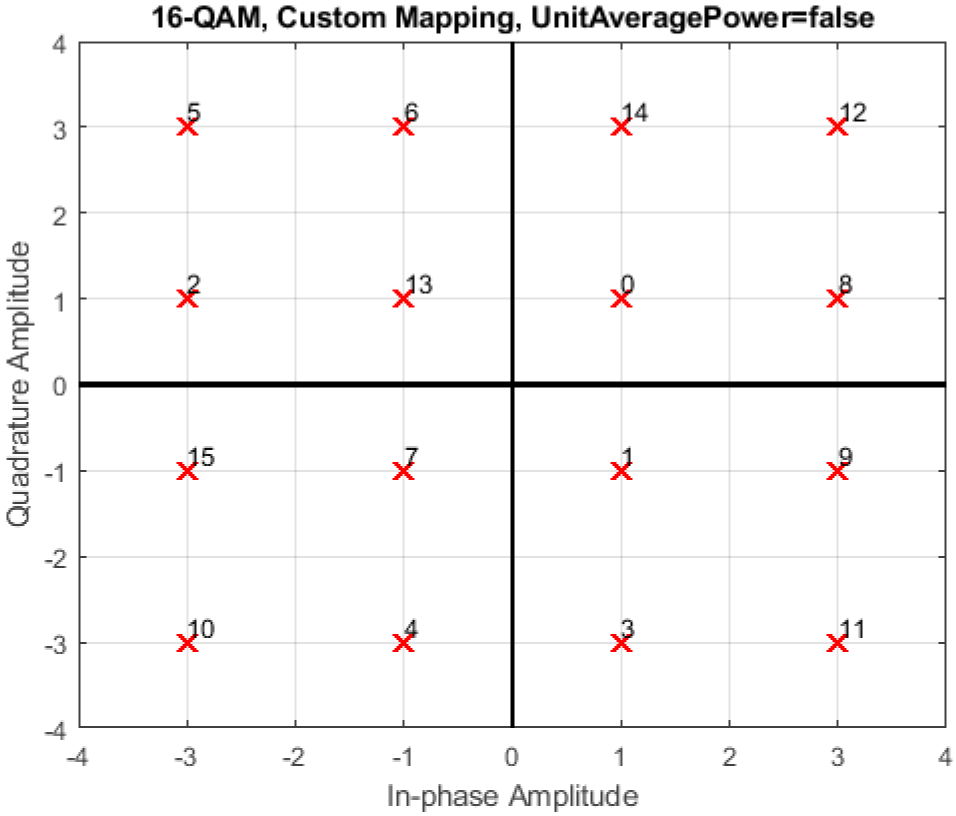


Create a custom symbol mapping.

```
smap = randperm(M)-1;
```

Modulate and plot the constellation.

```
w = qammod(d,M,smap,'PlotConstellation',true);
```



Quadrature Amplitude Modulation with Bit Inputs

Modulate a sequence of bits using 64-QAM. Pass the signal through a noisy channel. Display the resultant constellation diagram.

Set the modulation order, and determine the number of bits per symbol.

```
M = 64;
k = log2(M);
```

Create a binary data sequence. When using binary inputs, the number of rows in the input must be an integer multiple of the number of bits per symbol.

```
data = randi([0 1],1000*k,1);
```

Modulate the signal using bit inputs, and set it to have unit average power.

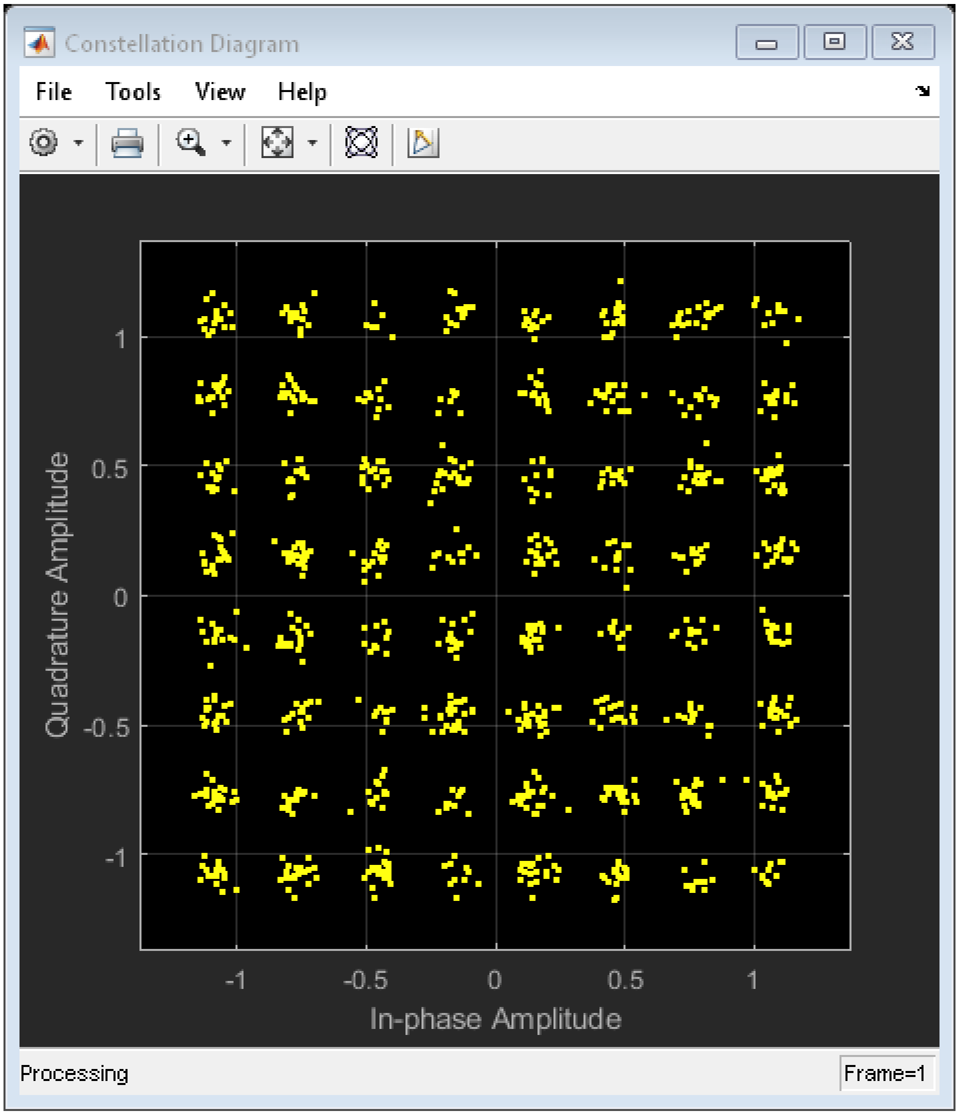
```
txSig = qammod(data,M,'InputType','bit','UnitAveragePower',true);
```

Pass the signal through a noisy channel.

```
rxSig = awgn(txSig,25);
```

Plot the constellation diagram.

```
cd = comm.ConstellationDiagram('ShowReferenceConstellation',false);
step(cd,rxSig)
```



Demodulate QAM Fixed-Point Signal

Demodulate a fixed-point QAM signal and verify that the data is recovered correctly.

Set the modulation order as 64, and determine the number of bits per symbol.

```
M = 64;
bitsPerSym = log2(M);
```

Generate random bits. When operating in bit mode, the length of the input data must be an integer multiple of the number of bits per symbol.

```
x = randi([0 1],10*bitsPerSym,1);
```

Modulate the input data using a binary symbol mapping. Set the modulator to output fixed-point data. The numeric data type is signed with a 16-bit word length and a 10-bit fraction length.

```
y = qammod(x,M,'bin','InputType','bit','OutputDataType', ...
    numerictype(1,16,10));
```

Demodulate the 64-QAM signal. Verify that the demodulated data matches the input data.

```
z = qamdemod(y,M,'bin','OutputType','bit');
s = isequal(x,double(z))
```

```
s = logical
   1
```

Input Arguments

x — Input signal

scalar | vector | matrix | 3-D array

Input signal, specified as a scalar, vector, matrix, or 3-D array. The elements of x must be binary values or integers that range from 0 to (M – 1), where M is the modulation order.

Note To process input signal as binary elements, set the 'InputType' name-value pair to 'bit'. For binary inputs, the number of rows must be an integer multiple of $\log_2$(M). Groups of $\log_2$(M) bits are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: double | single | fi | int8 | int16 | uint8 | uint16

M — Modulation order

scalar integer

Modulation order, specified as a power-of-two scalar integer. The modulation order specifies the number of points in the signal constellation.

Example: 16

Data Types: double

symOrder — Symbol order

'gray' (default) | 'bin' | vector

Symbol order, specified as 'gray', 'bin', or a vector.

- 'gray' — Use “Gray Code” on page 2-706 ordering
- 'bin' — Use natural binary-coded ordering
- Vector — Use custom symbol ordering

Vectors must use unique elements whose values range from 0 to M – 1. The first element corresponds to the upper-left point of the constellation, with subsequent elements running down column-wise from left to right.

Example: [0 3 1 2]

Data Types: char | double

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `y = qammod(x,M,symOrder,'InputType','bit')`

InputType — Input type

'integer' (default) | 'bit'

Input type, specified as the comma-separated pair consisting of `'InputType'` and either `'integer'` or `'bit'`. If you specify `'integer'`, the input signal must consist of integers from 0 to `M` – 1. If you specify `'bit'`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2(M)$.

Data Types: char

UnitAveragePower — Unit average power flag

false or 0 (default) | true or 1

Unit average power flag, specified as the comma-separated pair consisting of `'UnitAveragePower'` and a numeric or logical `0` (`false`) or `1` (`true`). When this flag is `1` (`true`), the function scales the constellation to the average power of one watt referenced to 1 ohm. When this flag is `0` (`false`), the function scales the constellation so that the QAM constellation points are separated by a minimum distance of two.

OutputDataType — Output data type

numerictype object

Output data type, specified as the comma-separated pair consisting of `'OutputDataType'` and a `numerictype` object.

For more information on constructing these objects, see `numerictype`. If you do not specify `'OutputDataType'`, data type is `double` if the input is of data type `double` or built-in integer and `single` if the input is of data type `single`.

PlotConstellation — Option to plot constellation

false or 0 (default) | true or 1

Option to plot constellation, specified as the comma-separated pair consisting of `'PlotConstellation'` and a numeric or logical `0` (`false`) or `1` (`true`) To plot the QAM constellation, set `'PlotConstellation'` to `true`.

Output Arguments

y — Modulated signal

scalar | vector | matrix | 3-D array

Modulated signal, returned as a complex scalar, vector, matrix, or 3-D array of numeric values. For integer inputs, output `y` has the same dimensions as input signal `x`. For bit inputs, the number of rows in `y` is the number of rows in `x` divided by $\log_2(M)$.

Data Types: double | single

More About

Gray Code

A Gray code, also known as a reflected binary code, is a system where the bit patterns in adjacent constellation points differ by only one bit.

Compatibility Considerations

Initial Phase Input Removed

Errors starting in R2018b

Starting in R2018b, you can no longer offset the initial phase for the QAM constellation using the `qammod` function.

Instead use `genqammod` to offset the initial phase of the data being modulated, or you can multiply the `qammod` output by the desired initial phase:

```
y = qammod(x,M) .* exp(1i*initPhase)
```

to adjust the initial phase of the QAM data.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`genqamdemod` | `genqammod` | `modnorm` | `pamdemod` | `pammod` | `qamdemod`

Topics

“Digital Modulation”

Introduced before R2006a

qfunc

Q function

Syntax

```
y = qfunc(x)
```

Description

`y = qfunc(x)` returns the output of the Q function for each element of the real-valued input. The Q function is (1 – *f*), where *f* is the result of the cumulative distribution function of the standardized normal random variable. For more information, see “Algorithms” on page 2-709.

Examples

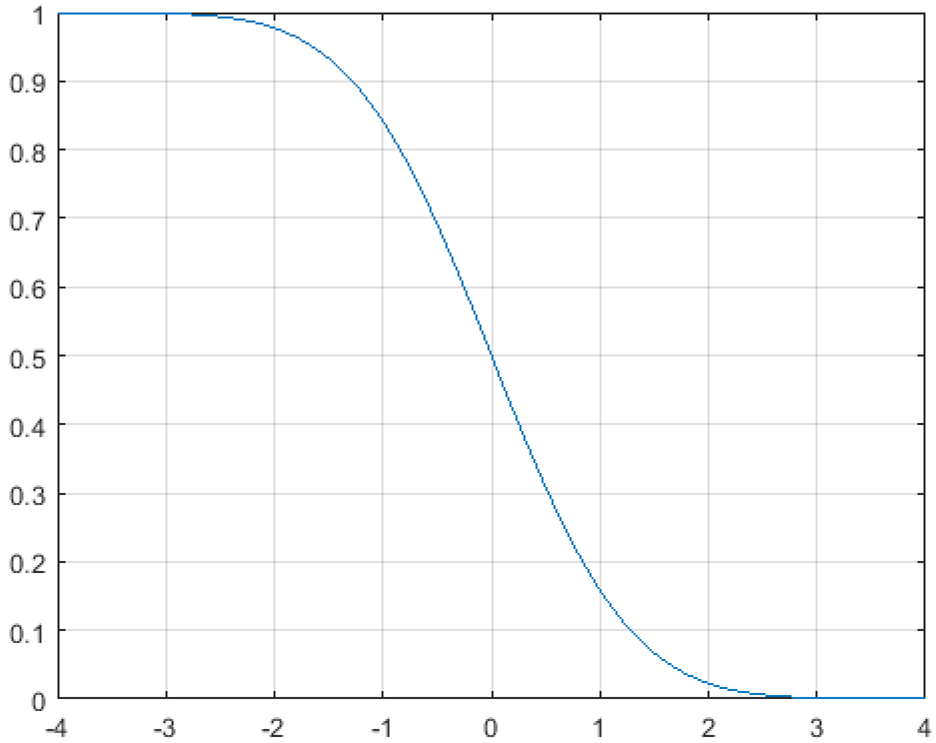
Calculate Q Function Value and Plot Results

Calculate the Q function values for a real-valued input vector.

```
x = -4:0.1:4;
y = qfunc(x);
```

Plot the results.

```
plot(x,y)
grid
```



Calculate QPSK Error Probability Using Q Function

Calculate the QPSK error probability at an E_b/N_0 setting of 7 dB by using the Q function.

Convert the E_b/N_0 in dB to its linear equivalent.

```
ebnodB = 7;
ebno = 10^(ebnodB/10);
```

Using the Q function, calculate the QPSK error probability, $P_b = Q\left(\sqrt{2\frac{E_\mathrm{b}}{N_0}}\right)$.

```
Pb = qfunc(sqrt(2*ebno))
```

```
Pb = 7.7267e-04
```

Input Arguments

x — Input
scalar | vector | matrix | array

Input, specified as a real-valued scalar, matrix, or array.

Data Types: `double`

Output Arguments

y — Q function output

scalar | vector | matrix | array

Q function output, returned as a scalar, matrix, or array. `y` has the same dimensions as input `x`. Output values are in the range [0, 1].

Algorithms

For a scalar *x*, the Q function is (1 – *f*), where *f* is the result of the cumulative distribution function of the standardized normal random variable. The Q function is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}}\int_{x}^{\infty} \exp(-t^2/2)dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2}\mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

See Also

Functions
`erf` | `erfc` | `erfcinv` | `erfcx` | `erfinv` | `qfuncinv`

Introduced before R2006a

qfuncinv

Inverse Q function

Syntax

```
z = qfuncinv(y)
```

Description

`z = qfuncinv(y)` returns the input argument of the Q function for which the output value of the Q function is `y`. For more information, see “Algorithms” on page 2-711.

Examples

Recover Argument of Q Function

Recover the Q function input argument by using the inverse Q function. Show the inverse relationship between Q function and its inverse.

Calculate the Q function values for a real-valued input.

```
x1 = [0 1 2; 3 4 5];
y1 = qfunc(x1)
```

```
y1 = 2×3

    0.5000    0.1587    0.0228
    0.0013    0.0000    0.0000
```

Recover the Q function input argument by calculating the inverse Q function values for `y1`.

```
x1_recovered = qfuncinv(y1)
```

```
x1_recovered = 2×3

     0     1     2
     3     4     5
```

Confirm the original and recovered Q functions arguments are the same.

```
isequal (x1,x1_recovered)
```

```
ans = logical
   1
```

Calculate the inverse of values representing Q function output values.

```
y2 = 0:0.2:1;
x2 = qfuncinv(y2)
```

```
x2 = 1×6

       Inf    0.8416    0.2533   -0.2533   -0.8416      -Inf
```

Recover the Q function output argument by calculating the Q function values for `x2`.

```
y2_recovered = qfunc(x2)
```

```
y2_recovered = 1×6

         0    0.2000    0.4000    0.6000    0.8000    1.0000
```

Confirm the original values and recovered inverse Q functions arguments are the same.

```
isequal (y2,y2_recovered)
```

```
ans = logical
   1
```

Input Arguments

y — Q function output
scalar | vector | matrix | array

Q function output, specified as a scalar, matrix, or array. Input values must be in the range [0, 1].

Data Types: `double`

Output Arguments

z — Q function input argument
scalar | vector | matrix | N-D array

Q function input argument, returned as a real-valued scalar, matrix, or array. `z` has the same dimensions as input `y`.

Algorithms

For a scalar *x*, the Q function is (1 – *f*), where *f* is the result of the cumulative distribution function of the standardized normal random variable. The Q function is defined as

$$Q(x) = \frac{1}{\sqrt{2\pi}}\int_{x}^{\infty}\exp(-t^2/2)dt$$

The Q function is related to the complementary error function, erfc, according to

$$Q(x) = \frac{1}{2}\text{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

See Also

Functions
`erf` | `erfc` | `erfcinv` | `erfcx` | `erfinv` | `qfunc`

Introduced before R2006a

quantiz

Produce quantization index and 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 of input signal `sig` by using the scalar quantization partition specified in input `partition`.

`[index,quants] = quantiz(sig,partition,codebook)` specifies `codebook`, which prescribes a value for each partition in the scalar quantization. `codebook` is a vector whose length must exceed the length of `partition` by one. The function also returns `quants`, which contains the scalar quantization of `sig` and depends on the quantization levels and prescribed values in the codebook.

`[index,quants,distor] = quantiz(sig,partition,codebook)` returns an estimate of the mean square distortion of the quantization data.

Examples

Produce Quantization Index and Quantized Output Value

Generate sample data.

```
samp = [-2.4, -1, -0.2, 0, 0.2, 1, 1.2, 2, 2.9, 3, 3.5, 5]
```

```
samp = 1×12

   -2.4000   -1.0000   -0.2000         0    0.2000    1.0000    1.2000    2.0000    2.9000    3.0
```

Create the quantization partition. To specify a partition, list the distinct endpoints of the different ranges of values.

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

Specify the codebook values.

```
codebook = [-1, 0.5, 2, 3]; % Codebook length must be equal to the number of partition intervals
```

Perform quantization on the sampled data. Display the quantization index and the corresponding quantized output value of the input data.

```
[index,quantized] = quantiz(samp,partition,codebook)
```

```
index = 1×12
```

```
     0     0     0     0     1     1     2     2     2     2     3     3

quantized = 1×12

   -1.0000   -1.0000   -1.0000   -1.0000    0.5000    0.5000    2.0000    2.0000    2.0000    2.0
```

Quantize Sampled Sine Wave

Generate a sampled sine wave.

```
t = [0:.1:2*pi];
sig = sin(t);
```

Create the quantization partition. To specify a partition, list the distinct endpoints of the different ranges of values.

```
partition = [-1:.2:1];
```

Specify the codebook values.

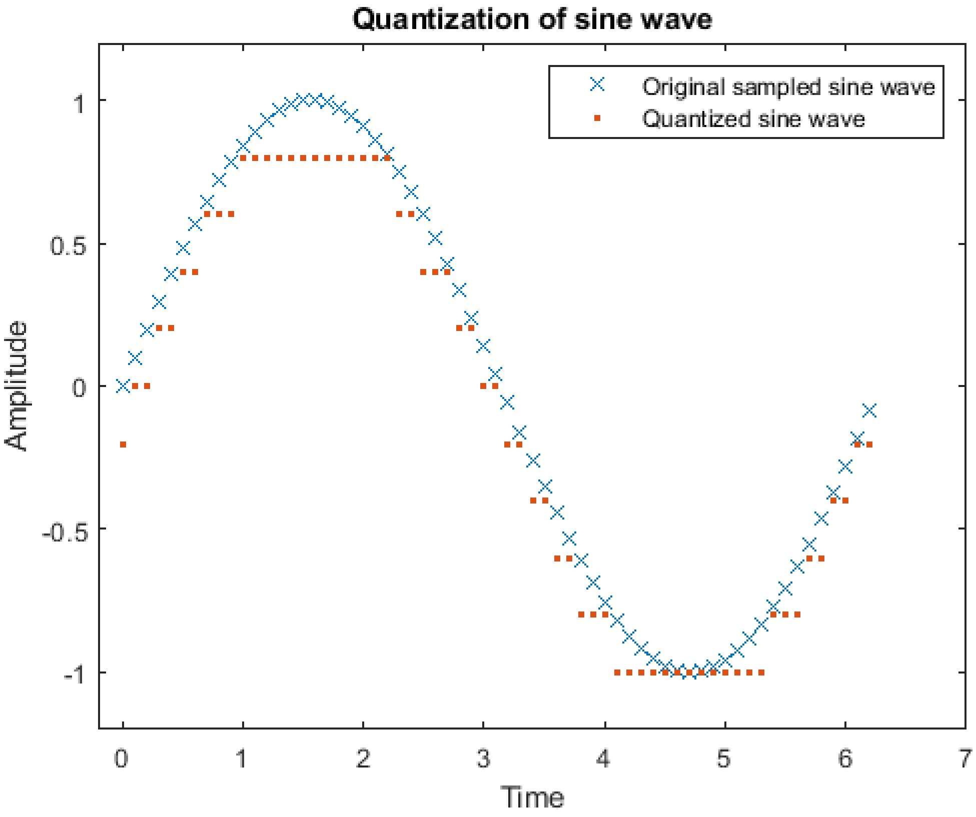
```
codebook = [-1.2:.2:1]; % Codebook length must be equal to the number of partition intervals
```

Perform quantization on the sampled sine wave.

```
[index,quants] = quantiz(sig,partition,codebook);
```

Plot the quantized sine wave and the sampled sine wave.

```
plot(t,sig,'x',t,quants,'.')
title('Quantization of sine wave')
xlabel('Time')
ylabel('Amplitude')
legend('Original sampled sine wave','Quantized sine wave');
axis([-.2 7 -1.2 1.2])
```



Input Arguments

`sig` — Input signal

vector

Input signal, specified as a vector. This input specifies the sampled signal for this function to perform quantization.

Data Types: `double`

`partition` — Distinct endpoints of different ranges

row vector

Distinct endpoints of different ranges, specified as a row vector. This input defines several contiguous, nonoverlapping ranges of values within the set of real numbers. The values present in this input must be strictly in ascending order. The length of this vector must be one less than the number of partition intervals.

Example: [`0`, `1`, `3`] partitions the input row vector into the four sets {*X*: *X* ≤ `0`}, {*X*: `0` < *X* ≤ `1`}, {*X*: `1` < *X* ≤ `3`}, and {*X*: *3* < *X*}.

Data Types: `double`

codebook — Quantization value for each partition

row vector

Quantization value for each partition, specified as a row vector. This input prescribes a common value for each partition in the scalar quantization. The length of this vector must equal the number of partition intervals, that is, the length of this vector must exceed the length of the `partition` input by one.

Data Types: `double`

Output Arguments

`index` — Quantization index

nonnegative row vector

Quantization index of the input signal, returned as a nonnegative row vector. This output determines on which partition interval, each input value is mapped. Each element in `index` is one of the *N* integers in the range [0, *N*–1].

If the `partition` input has length *N*, `index` is a vector whose *K*th entry is:

- `0` if `sig`(*K*) ≤ `partition`(`1`)
- *M* if `partition`(*M*) < `sig`(*K*) ≤ `partition`(*M+1*)
- *N* if `partition`(*N*) ≤ `sig`(K)

`quants` — Output of quantizer

row vector

Output of the quantizer, which contains the quantization values of the input signal, returned as a row vector. The size of `quants` matches that of input argument `sig`. When `codebook` is not specified as an input argument, you can define the codebook values as a vector whose length must exceed the length of the `partition` by one.

`quants` is calculated based on the `codebook` and `index` inputs and is given by `quants`(*i*) = `codebook`(`index`(*i*) + 1), where *i* is an integer between 1 and the length of `sig`.

`distor` — Mean square distortion

positive scalar

Mean square distortion of the quantized signal, returned as a positive scalar. You can reduce this distortion by choosing appropriate partition and codebook values. For more information on optimizing partition and codebook values, see the `lloyds` function.

See Also

Functions

`dpcmdeco` | `dpcmenco` | `huffmandeco` | `huffmanenco` | `lloyds`

Introduced before R2006a

randdeintrlv

Restore ordering of symbols using random permutation

Syntax

```
deintrlvd = randdeintrlv(data,state)
```

Description

`deintrlvd = randdeintrlv(data,state)` restores the original ordering of the elements in `data` by inverting a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

To use this function as an inverse of the `randintrlv` function, use the same `state` input in both functions. In that case, the two functions are inverses in the sense that applying `randintrlv` followed by `randdeintrlv` leaves data unchanged.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the `rand` function.

See `rand` for details on the generator algorithm.

Examples

For an example using random interleaving and deinterleaving, see "Improve Error Rate Using Block Interleaving in MATLAB".

See Also

`rand` | `randintrlv`

Topics

"Interleaving"

Introduced before R2006a

randerr

Generate bit error patterns

Syntax

```
out = randerr(m)
out = randerr(m,n)
out = randerr(m,n,errors)
out = randerr(m,n,errors,seed)
out = randerr(m,n,errors,randstream)
```

Description

Use the `randerr` function to generate bit error patterns. For all syntaxes, `randerr` treats each row of the output independently.

`out = randerr(m)` generates an `m`-by-`m` binary matrix, where each row has exactly one nonzero entry in a random position. Each permutation has an equal probability.

`out = randerr(m,n)` generates an `m`-by-`n` binary matrix, where each row has exactly one nonzero entry in a random position. Each permutation has an equal probability.

`out = randerr(m,n,errors)` uses the `errors` input to determine the number of nonzero entries in each row of the output `m`-by-`n` binary matrix.

`out = randerr(m,n,errors,seed)` specifies a seed value for initializing the uniform random number generator of the `rand` function.

`out = randerr(m,n,errors,randstream)` specifies a random stream object to generate uniform random noise samples by using the `rand` function. Providing a random stream object or using the `reset (RandStream)` function on the default random stream object enables you to generate repeatable noise samples.

Note To generate repeatable noise samples, use the same seed input value for each call of `randerr` or reset the random stream input before calling `randerr`. For more information on resetting the random stream, see the `RandStream` object.

Examples

Generate Random Error Matrix

Generate an 8-by-7 binary matrix in which each row is equally likely to have either zero or two nonzero elements.

```
out = randerr(8,7,[0 2])
```

```
out = 8×7
```

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

Generate a matrix in which each row is three times more likely to have two nonzero elements rather than zero nonzero elements.

```
out = randerr(8,7,[0 2; 0.25 0.75])
```

```
out = 8×7

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

Generate Repeatable Random Error Matrix

Demonstrate generation of a random error matrix without a seed input value for a nonrepeatable output and with the seed input value for a repeatable output.

Specify input parameters for the output matrix dimensions, the number of errors, and a seed value.

```
m = 2;
n = 8;
errors = 2;
seed = 1234;
```

Use the `randerr` function to generate a random error binary matrix twice with the same command. The output binary matrix values are the same for each execution of the `randerr` function.

```
out = randerr(m,n,errors,seed)
```

```
out = 2×8

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

```
out = randerr(m,n,errors,seed)
```

```
out = 2×8

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

Change the seed value and call the `randerr` function twice. The binary matrix output values are the same for each execution of the `randerr` function, but they differ from the binary matrix values output using the previous seed value.

```
seed = 345;
out = randerr(m,n,errors,seed)
```

```
out = 2×8

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

```
out = randerr(m,n,errors,seed)
```

```
out = 2×8

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

Use the `randerr` function to generate a random error binary matrix twice with the same command, not specifying a seed input value. The output matrix values change for each execution of the `randerr` function.

```
out = randerr(m,n,errors)
```

```
out = 2×8

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

```
out = randerr(m,n,errors)
```

```
out = 2×8

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

Input Arguments

`m` — Size of random binary matrix

positive integer

Size of the random binary matrix, specified as a positive integer.

- When you specify only the input `m`, the random binary matrix output is of size `m`-by-`m`.
- When you specify the inputs `m` and `n`, the random binary matrix output is of size `m`-by-`n`.

Data Types: `double`

n — Column size of random binary matrix

1 (default) | positive integer

Column size of the random binary matrix, specified as a positive integer.

Data Types: double

errors — Number of nonzero entries

1 (default) | nonnegative integer | nonnegative integer row vector | nonnegative two-row matrix

Number of nonzero entries, specified as one of these forms.

- If specified as a integer, errors defines the number of 1s in each row.
- If specified as a integer row vector, errors defines the number of 1s possible in each row. Every number of 1s included in this vector occurs with equal probability.
- If specified as a two-row matrix, the first row of errors defines the integer number of 1s possible in any given row of the output matrix. The second row specifies the probabilities of each corresponding number of ones. The elements in the second row of errors must sum to 1.

Data Types: double

seed — Seed value

nonnegative integer value less than 2^{32}

Seed value for initializing the uniform random number generator used in the rand function, specified as nonnegative integer value less than 2^{32}.

Data Types: double

randstream — Random stream object

RandStream object

Random stream object to generate uniform random noise samples by using the rand function, specified as a RandStream object. Providing a random stream object or using the reset (RandStream) function on the default random stream object enables you to generate repeatable noise samples.

Output Arguments

out — Random binary matrix output

matrix

Random binary matrix output, returned as a matrix of binary values.

- When you specify only the input m, this output is of size m-by-m.
- When you specify the inputs m and n, this output is of size m-by-n.

Data Types: double

See Also

Functions

rand | randi | randsrc

Objects
`RandStream`

Topics
"Sources and Sinks"

Introduced before R2006a

randintrlv

Reorder symbols using random permutation

Syntax

```
intrlvd = randintrlv(data,state)
```

Description

`intrlvd = randintrlv(data,state)` rearranges the elements in `data` using a random permutation. The `state` parameter initializes the random number generator that the function uses to determine the permutation. `state` is either a scalar or a 35x1 vector, and is described in the `rand` function, which is used in `randintrlv`. The function is predictable and invertible for a given state, but different states produce different permutations. If `data` is a matrix with multiple rows and columns, the function processes the columns independently.

This function uses, by default, the Mersenne Twister algorithm by Nishimura and Matsumoto.

Note Using the `state` parameter causes this function to switch random generators to use the `'state'` algorithm of the`rand` function.

See `rand` for details on the generator algorithm.

Examples

For an example using random interleaving and deinterleaving, see “Improve Error Rate Using Block Interleaving in MATLAB”.

See Also

`rand` | `randdeintrlv`

Topics

“Interleaving”

Introduced before R2006a

randseed

(To be removed) Generate prime numbers for use as random number seeds

Compatibility

randseed will be removed in a future release. Use `rng(N)` or `rng('shuffle')` instead. For more information, see “Compatibility Considerations” on page 2-725.

Syntax

```
out = randseed
out = randseed(state)
out = randseed(state,m)
out = randseed(state,m,n)
out = randseed(state,m,n,rmin)
out = randseed(state,m,n,rmin,rmax)
```

Description

The `randseed` function produces random prime numbers that work well as seeds for random source blocks or noisy channel blocks in Communications Toolbox software.

Note The `randseed` function uses a local stream of numbers that is independent from the global stream of numbers in the MATLAB software. Use of this function does not affect the state of the global random number stream.

`out = randseed` generates a random prime number between 31 and 2^{17}-1, using the MATLAB function `rand`.

`out = randseed(state)` generates a random prime number after setting the state of `rand` to the positive integer `state`. This syntax produces the same output for a particular value of `state`.

`out = randseed(state,m)` generates a column vector of `m` random primes.

`out = randseed(state,m,n)` generates an `m`-by-`n` matrix of random primes.

`out = randseed(state,m,n,rmin)` generates an `m`-by-`n` matrix of random primes between `rmin` and 2^{17}-1.

`out = randseed(state,m,n,rmin,rmax)` generates an `m`-by-`n` matrix of random primes between `rmin` and `rmax`.

Examples

To generate a two-element sample-based row vector of random bits using the Bernoulli Random Binary Generator block, you can set **Probability of a zero** to `[0.1 0.5]` and set **Initial seed** to `randseed(391,1,2)`.

To generate three streams of random data from three different blocks in a single model, you can define `out = randseed(93,3)` in the MATLAB workspace and then set the three blocks' **Initial seed** parameters to `out(1)`, `out(2)`, and `out(3)`, respectively.

Compatibility Considerations

`randseed` will be removed

Warns starting in R2019a

The functionality provided by `randseed` is no longer necessary for controlling random number generation. Instead, use `rng`(`seed`), where `seed` specifies a nonnegative integer seed for the random number generator, or use `rng`(`‘shuffle’`) to seed the random number generator based on the current time.

See Also

`primes` | `rand` | `rng`

Introduced before R2006a

randsrc

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,___,seed)
out = randsrc(m,n,___,streamhandle)
```

Description

`out = randsrc` generates a random scalar that is either `-1` or `1`, with equal probability.

`out = randsrc(m)` generates an `m`-by-`m` random bipolar matrix. Each entry independently takes the value `-1` or `1` with equal probability.

`out = randsrc(m,n)` generates an `m`-by-`n` random bipolar matrix. Each entry independently takes the value `-1` or `1` with equal probability.

`out = randsrc(m,n,alphabet)` generates an `m`-by-`n` matrix, with each entry 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, with each entry 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 1.

`out = randsrc(m,n,___,seed)` accepts input combinations from prior syntaxes and a seed value, for initializing the uniform random number generator, `rand`.

`out = randsrc(m,n,___,streamhandle)` accepts input combinations from prior syntaxes and a random stream handle to generate uniform random noise samples by using `rand`. Providing a random stream handle or using the `reset (RandStream)` function on the default random stream object enables you to generate repeatable noise samples. If you want to generate repeatable noise samples, then either reset the random stream input before calling `randsrc` or use the same seed input. For more information, see `RandStream`.

Examples

Generate Random Matrix from Prescribed Alphabet

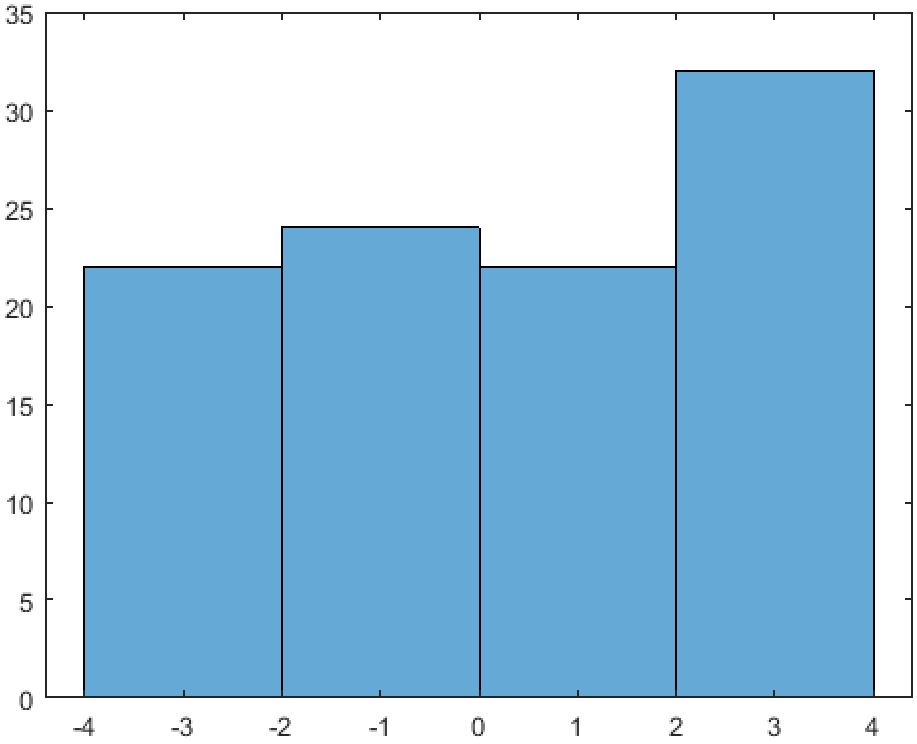
Generate a 10-by-10 matrix from the set of {-3,-1,1,3}.

```
out = randsrc(10,10,[-3 -1 1 3])
```

```
out = 10×10

     3    -3     1     1    -1    -1     3     3    -1    -3
     3     3    -3    -3    -1     1    -1    -1     3    -3
    -3     3     3    -1     3     1     1     3     1     1
     3    -1     3    -3     3    -3     1    -3     1     3
     1     3     1    -3    -3    -3     3     3     3     3
    -3    -3     3     3    -1    -1     3    -1    -1    -3
    -1    -1     1     1    -1     3     1    -3     3     1
     1     3    -1    -1     1    -1    -3    -1     3    -1
     3     3     1     3     1     1    -3     1    -1    -3
     3     3    -3    -3     3    -3    -1    -1     1    -1
```

Plot the histogram. Each of the four possible element values occur with equal probability. Your values might differ.

```
histogram(out,[-4 -2 0 2 4])
```



Generate a matrix in which the likelihood of a -1 or 1 is four times higher than the likelihood of a -3 or 3.

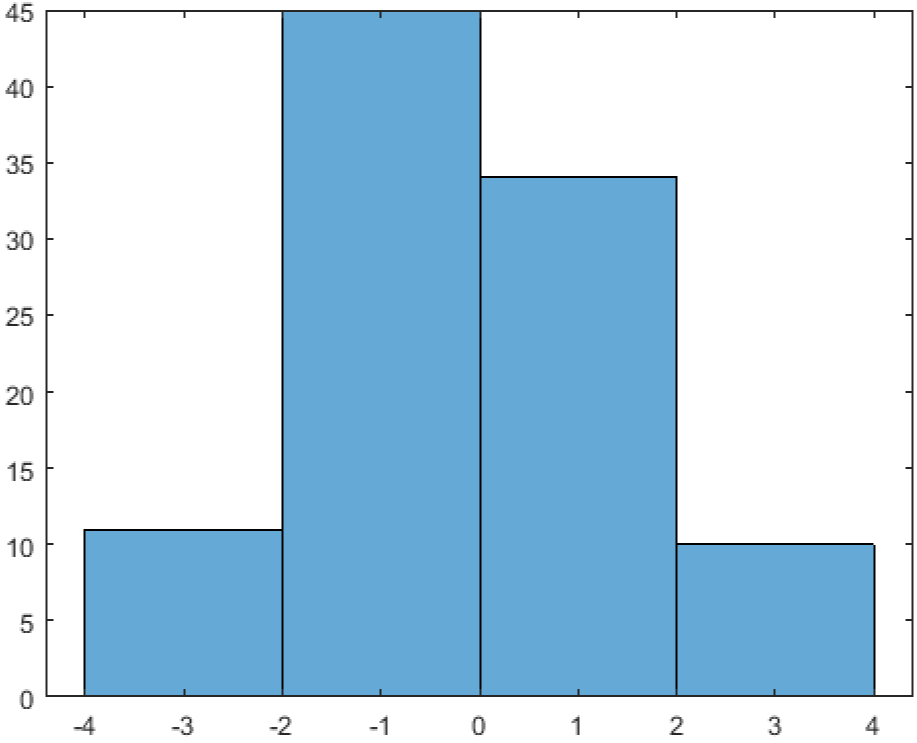
```
out = randsrc(10,10,[-3 -1 1 3; 0.1 0.4 0.4 0.1])
```

```
out = 10×10
```

```
    -1    -1    -1    -1     1    -1     1    -1     1     3
     1    -3     3     3     1    -3    -1    -1    -1     1
    -1    -1    -3    -1    -1     3    -1     1     1    -1
     1     3     1    -1     1     3    -1    -3    -1    -1
    -1    -1     1    -1    -1    -1    -3    -3     1    -1
     1     1     1    -1    -3    -1    -1    -1    -1    -1
    -1     1    -3     1    -1    -1     3     1    -1     1
     1     3    -1     1    -1     3     3     1     1     1
     1    -3    -1     1    -1    -1     1     1     1     1
     1    -1     1    -1    -1    -1    -3    -1    -3     1
```

Plot the histogram. Values of -1 and 1 are more likely.

```
histogram(out,[-4 -2 0 2 4])
```



Input Arguments

`m` — Bipolar matrix size

1 (default) | scalar

Size of random bipolar matrix, specified as a scalar. If `n` is specified, then `m` is the row size of the random bipolar matrix.

Data Types: `double`

n — Bipolar matrix column size

1 (default) | scalar

Column size of random binary matrix, specified as a scalar.

Data Types: double

alphabet — Possible element values

[-1 1] (default) | vector | matrix

Possible elements of output vector or matrix. If alphabet is a row vector, the contents of alphabet define which possible elements randsrc output. If alphabet is a two-row matrix, then the first row is defines the possible elements, and the second row defines the probabilities for each corresponding element in the first row. The elements of the second row must sum to one. If all entries of alphabet are distinct, then the probability distribution is uniform.

Data Types: double

prob — Element probabilities

[0.5 0.5] (default) | vector

Row vector of probabilities that correspond to elements of the corresponding alphabet vector.

Data Types: double

seed — Seed value

scalar

Seed value for initializing the uniform random number generator, rand.

Data Types: double

streamhandle — Random stream handle

RandStream Object

Random stream handle to generate uniform random noise samples by using rand. Providing a random stream handle or using the reset (RandStream) function on the default random stream object enables you to generate repeatable noise samples. If you want to generate repeatable noise samples, then either reset the random stream input before calling randsrc or use the same seed input. For more information, see RandStream.

Data Types: double

Output Arguments

out — Random matrix output

scalar | vector | matrix

Random output, returned as a scalar, vector, or matrix. The dimensions of the output are specified by arguments m and n, otherwise it is a 1-by-1 scalar. The possible elements of the output and their probabilities are specified by alphabet, prob respectively, otherwise the elements of the output are -1 and 1, with equal distribution.

Data Types: double

See Also

Functions

`RandStream` | `rand` | `randerr` | `randi`

Introduced before R2006a

rayleighchan

(Removed) Construct Rayleigh fading channel object

Note `rayleighchan` has been removed. Use `comm.RayleighChannel` instead.

Syntax

```
chan = rayleighchan(ts,fd)
chan = rayleighchan(ts,fd,tau,pdb)
chan = rayleighchan
```

Description

`chan = rayleighchan(ts,fd)` constructs a frequency-flat ("single path") Rayleigh fading channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. You can model the effect of the channel on a signal `x` by using the syntax `y = filter(chan,x)`.

`chan = rayleighchan(ts,fd,tau,pdb)` constructs a frequency-selective ("multiple path") fading channel object that models each discrete path as an independent Rayleigh fading process. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

With the above two syntaxes, a smaller `fd` (a few hertz to a fraction of a hertz) leads to slower variations, and a larger `fd` (a couple hundred hertz) to faster variations.

`chan = rayleighchan` constructs a frequency-flat Rayleigh channel object with no Doppler shift. This is a static channel. The sample time of the input signal is irrelevant for frequency-flat static channels.

Properties

The tables below describe the properties of the channel object, `chan`, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see “Displaying and Changing Object Properties”.

Writeable Properties

Property	Description
InputSamplePeriod	Sample period of the signal on which the channel acts, measured in seconds.
DopplerSpectrum	Doppler spectrum object(s). The default is a Jakes Doppler object.
MaxDopplerShift	Maximum Doppler shift of the channel, in hertz (applies to all paths of a channel).
PathDelays	Vector listing the delays of the discrete paths, in seconds.
AvgPathGaindB	Vector listing the average gain of the discrete paths, in decibels.
NormalizePathGains	If 1, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
StoreHistory	If this value is 1, channel state information needed by the channel visualization tool is stored as the channel filter function processes the signal. The default value is 0.
StorePathGains	If set to 1, the complex path gain vector is stored as the channel filter function processes the signal. The default value is 0.
ResetBeforeFiltering	If 1, each call to filter resets the state of chan before filtering. If 0, the fading process maintains continuity from one call to the next.

Read-Only Properties

Property	Description	When MATLAB Sets or Updates Value
ChannelType	Fixed value, 'Rayleigh'	When you create object
PathGains	Complex vector listing the current gains of the discrete paths. When you create or reset chan, PathGains is a random vector influenced by AvgPathGaindB and NormalizePathGains.	When you create object, reset object, or use it to filter a signal
ChannelFilterDelay	Delay of the channel filter, measured in samples. The ChannelFilterDelay property returns a delay value that is valid only if the first value of the PathGain is the biggest path gain. In other words, main channel energy is in the first path.	When you create object or change ratio of InputSamplePeriod to PathDelays
NumSamplesProcessed	Number of samples the channel processed since the last reset. When you create or reset chan, this property value is 0.	When you create object, reset object, or use it to filter a signal

Relationships Among Properties

The PathDelays and AvgPathGaindB properties of the channel object must always have the same vector length, because this length equals the number of discrete paths of the channel. The DopplerSpectrum property must either be a single Doppler object or a vector of Doppler objects with the same length as PathDelays.

If you change the length of PathDelays, MATLAB truncates or zero-pads the value of AvgPathGaindB if necessary to adjust its vector length (MATLAB may also change the values of read-only properties such as PathGains and ChannelFilterDelay). If DopplerSpectrum is a vector of Doppler objects, and you increase or decrease the length of PathDelays, MATLAB will add Jakes Doppler objects or remove elements from DopplerSpectrum, respectively, to make it the same length as PathDelays.

If StoreHistory is set to 1 (the default is 0), the object stores channel state information as the channel filter function processes the signal. You can then visualize this state information through a GUI using the plot (channel) method.

Note Setting StoreHistory to 1 will result in a slower simulation. If you do not want to visualize channel state information using plot (channel), but want to access the complex path gains, then set StorePathGains to 1, while keeping StoreHistory as 0.

Visualization of Channel

The characteristics of a channel can be plotted using the channel visualization tool, plot (channel). You can use the channel visualization tool in Normal mode and Accelerator mode.

Examples

The example below illustrates that when you change the value of `PathDelays`, MATLAB automatically changes the values of other properties to make their vector lengths consistent with that of the new value of `PathDelays`.

```
c1 = rayleighchan(1e-5,130) % Create object.
c1.PathDelays = [0 1e-6] % Change the number of delays.
```

MATLAB automatically changes the size of `c1.AvgPathGaindB`, `c1.PathGains`, and `c1.ChannelFilterDelay`. The output below displays all the properties of the channel object before and after the change in the value of the `PathDelays` property. In the second listing of properties, the `AvgPathGaindB`, `PathGains`, and `ChannelFilterDelay` properties all have different values compared to the first listing of properties.

```
c1 =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-005
         DopplerSpectrum: [1x1 doppler.jakes]
         MaxDopplerShift: 130
              PathDelays: 0
           AvgPathGaindB: 0
      NormalizePathGains: 1
            StoreHistory: 0
               PathGains: 0.2035 + 0.1014i
      ChannelFilterDelay: 0
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0


c1 =

             ChannelType: 'Rayleigh'
       InputSamplePeriod: 1.0000e-005
         DopplerSpectrum: [1x1 doppler.jakes]
         MaxDopplerShift: 130
              PathDelays: [0 1.0000e-006]
           AvgPathGaindB: [0 0]
      NormalizePathGains: 1
            StoreHistory: 0
               PathGains: [0.6108 - 0.4688i 0.1639 - 0.0027i]
      ChannelFilterDelay: 4
    ResetBeforeFiltering: 1
     NumSamplesProcessed: 0
```

Algorithms

The methodology used to simulate fading channels is described in “Methodology for Simulating Multipath Fading Channels”. The properties of the channel object are related to the quantities of the latter section as follows:

- The `InputSamplePeriod` property contains the value of T_s.
- The `PathDelays` vector property contains the values of $\{\tau_k\}$, where $1 \le k \le K$.

- The `PathGains` read-only property contains the values of $\{a_k\}$, where $1 \le k \le K$.
- The `AvgPathGaindB` vector property contains the values of $10\log_{10}\left\{E\left[|a_k|^2\right]\right\}$, where $1 \le k \le K$, and $E[\,\cdot\,]$ denotes statistical expectation.
- The `ChannelFilterDelay` read-only property contains the value of N_1.

Compatibility Considerations

`rayleighchan` has been removed
Errors starting in R2020b

rayleighchan has been removed. Use `comm.RayleighChannel` instead.

References

[1] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan, *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`comm.RayleighChannel`

Topics
"Fading Channels"

Introduced before R2006a

rectpulse

Rectangular pulse shaping

Syntax

```
y = rectpulse(x,nsamp)
```

Description

`y = rectpulse(x,nsamp)` applies rectangular pulse shaping to `x` to produce an output signal having `nsamp` samples per symbol. Rectangular pulse shaping means that each symbol from `x` is repeated `nsamp` times to form the output `y`. If `x` is a matrix with multiple rows, the function treats each column as a channel and processes the columns independently.

Note To insert zeros between successive samples of `x` instead of repeating the samples of `x`, use the `upsample` function instead.

Examples

An example in “Combine Pulse Shaping and Filtering with Modulation” uses this function in conjunction with modulation.

The code below processes two independent channels, each containing three symbols of data. In the pulse-shaped matrix `y`, each symbol contains four samples.

```
nsamp = 4; % Number of samples per symbol
nsymb = 3; % Number of symbols
s = RandStream('mt19937ar', 'Seed', 0);
ch1 = randi(s, [0 1], nsymb, 1); % Random binary channel
ch2 = [1:nsymb]';
x = [ch1 ch2] % Two-channel signal
y = rectpulse(x,nsamp)
```

The output is below. In `y`, each column corresponds to one channel and each row corresponds to one sample. Also, the first four rows of `y` correspond to the first symbol, the next four rows of `y` correspond to the second symbol, and the last four rows of `y` correspond to the last symbol.

```
x =

     1     1
     1     2
     0     3


y =

     1     1
     1     1
     1     1
     1     1
     1     2
     1     2
     1     2
     1     2
     0     3
     0     3
     0     3
     0     3
```

See Also

`intdump` | `upsample`

Introduced before R2006a

reset (channel)

(To be removed) Reset channel object

Note This function will be removed in a future release. Use function associated with `comm.RicianChannel` or `comm.RayleighChannel` instead.

Syntax

```
reset(chan)
reset(chan,randstate)
```

Description

`reset(chan)` resets the channel object `chan`, initializing the `PathGains` and `NumSamplesProcessed` properties as well as internal filter states. This syntax is useful when you want the effect of creating a new channel.

`reset(chan,randstate)` resets the channel object `chan` and initializes the state of the random number generator that the channel uses. `randstate` is a two-element column vector. This syntax is useful when you want to repeat previous numerical results that started from a particular state.

Note `reset(chan,randstate)` will not support `randstate` in a future release. See the `legacychannelsim` function for more information.

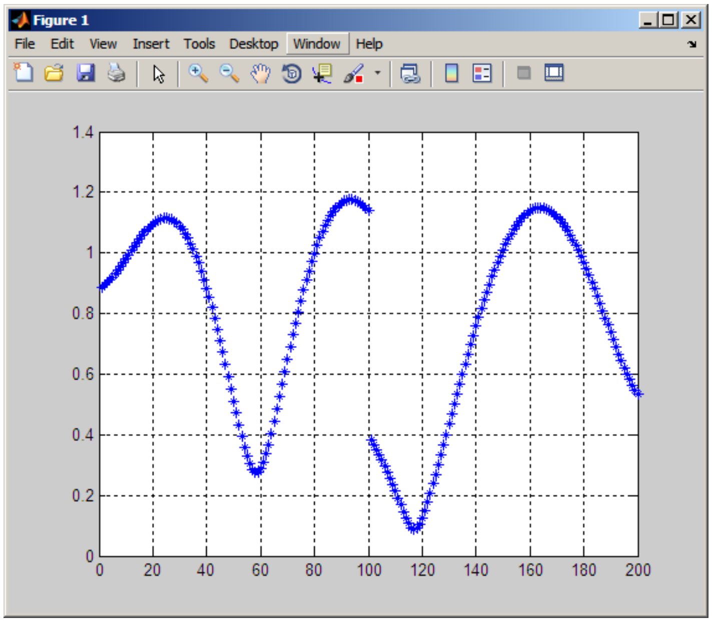
Examples

The example below shows how to obtain repeatable results. The example chooses a state for the random number generator immediately after defining the channel object and later resets the random number generator to that state.

```
%% Set up channel.
% Assume you want to maintain continuity
% from one filtering operation to the next, except
% when you explicitly reset the channel.
c = rayleighchan(1e-4,100);
c.ResetBeforeFiltering = 0;

% Filter all ones.
sig = ones(100,1);
y1 = [filter(c,sig(1:50)); filter(c,sig(51:end))];

% Reset the channel and filter all ones.
reset(c);
% Generate an independent channel
y2 = [filter(c,sig(1:50)); filter(c,sig(51:end))];

% Plot the magnitude of the channel output
```

```
plot(abs([y1; y2]),'*')
grid on
```

This example generates a plot similar to this figure.



Compatibility Considerations

reset has been removed

Errors starting in R2020b

reset (channel) has been removed. Use `comm.RicianChannel` instead.

See Also

`comm.RayleighChannel` | `comm.RicianChannel`

Topics

“Fading Channels”

Introduced in R2007a

reset (equalizer)

Reset equalizer object

Syntax

```
reset(eqobj)
```

Description

`reset(eqobj)` resets the equalizer object `eqobj`, initializing the `Weights`, `WeightInputs`, and `NumSamplesProcessed` properties and the adaptive algorithm states. If `eqobj` is a CMA equalizer, `reset` does not change the `Weights` property.

See Also

`dfe` | `equalize` | `lineareq`

Topics

"Equalization"

Introduced before R2006a

ricianchan

(Removed) Construct Rician fading channel object

Note `ricianchan` has been removed. Use `comm.RicianChannel` instead.

Syntax

```
chan = ricianchan(ts,fd,k)
chan = ricianchan(ts,fd,k,tau,pdb)
chan = ricianchan(ts,fd,k,tau,pdb,fdlos)
chan = ricianchan
```

Description

`chan = ricianchan(ts,fd,k)` constructs a frequency-flat (single path) Rician fading-channel object. `ts` is the sample time of the input signal, in seconds. `fd` is the maximum Doppler shift, in hertz. `k` is the Rician K-factor in linear scale. You can model the effect of the channel `chan` on a signal `x` by using the syntax `y = filter(chan,x)`.

`chan = ricianchan(ts,fd,k,tau,pdb)` constructs a frequency-selective (multiple paths) fading-channel object. If `k` is a scalar, then the first discrete path is a Rician fading process (it contains a line-of-sight component) with a K-factor of `k`, while the remaining discrete paths are independent Rayleigh fading processes (no line-of-sight component). If `k` is a vector of the same size as `tau`, then each discrete path is a Rician fading process with a K-factor given by the corresponding element of the vector `k`. `tau` is a vector of path delays, each specified in seconds. `pdb` is a vector of average path gains, each specified in dB.

`chan = ricianchan(ts,fd,k,tau,pdb,fdlos)` specifies `fdlos` as the Doppler shift(s) of the line-of-sight component(s) of the discrete path(s), in hertz. `fdlos` must be the same size as `k`. If `k` and `fdlos` are scalars, the line-of-sight component of the first discrete path has a Doppler shift of `fdlos`, while the remaining discrete paths are independent Rayleigh fading processes. If `fdlos` is a vector of the same size as `k`, the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector `fdlos`. By default, `fdlos` is `0`. The initial phase(s) of the line-of-sight component(s) can be set through the property `DirectPathInitPhase`.

`chan = ricianchan` sets the maximum Doppler shift to `0`, the Rician K-factor to `1`, and the Doppler shift and initial phase of the line-of-sight component to `0`. This syntax models a static frequency-flat channel, and, in this trivial case, the sample time of the signal is unimportant.

Properties

The following tables describe the properties of the channel object, `chan`, that you can set and that MATLAB technical computing software sets automatically. To learn how to view or change the values of a channel object, see “Displaying and Changing Object Properties”.

Writeable Properties

Property	Description
`InputSamplePeriod`	Sample period of the signal on which the channel acts, measured in seconds.
`DopplerSpectrum`	Doppler spectrum object(s). The default is a Jakes doppler object.
`MaxDopplerShift`	Maximum Doppler shift of the channel, in hertz (applies to all paths of a channel).
`KFactor`	Rician K-factor (scalar or vector). The default value is 1 (line-of-sight component on the first path only).
`PathDelays`	Vector listing the delays of the discrete paths, in seconds.
`AvgPathGaindB`	Vector listing the average gain of the discrete paths, in decibels.
`DirectPathDopplerShift`	Doppler shift(s) of the line-of-sight component(s) in hertz. The default value is 0.
`DirectPathInitPhase`	Initial phase(s) of line-of-sight component(s) in radians. The default value is 0.
`NormalizePathGains`	If this value is `1`, the Rayleigh fading process is normalized such that the expected value of the path gains' total power is 1.
`StoreHistory`	If this value is `1`, channel state information needed by the channel visualization tool is stored as the channel filter function processes the signal. The default value is `0`.
`StorePathGains`	If this value is `1`, the complex path gain vector is stored as the channel filter function processes the signal. The default value is `0`.
`ResetBeforeFiltering`	If this value is `1`, each call to `filter` resets the state of `chan` before filtering. If it is `0`, the fading process maintains continuity from one call to the next.

Read-Only Properties

Property	Description	When MATLAB Sets or Updates Value
ChannelType	Fixed value, 'Rician'.	When you create object.
PathGains	Complex vector listing the current gains of the discrete paths. When you create or reset chan, PathGains is a random vector influenced by AvgPathGaindB and NormalizePathGains.	When you create object, reset object, or use it to filter a signal.
ChannelFilterDelay	Delay of the channel filter, measured in samples. The ChannelFilterDelay property returns a delay value that is valid only if the first value of the PathGain is the biggest path gain. In other words, main channel energy is in the first path.	When you create object or change ratio of InputSamplePeriod to PathDelays.
NumSamplesProcessed	Number of samples the channel processed since the last reset. When you create or reset chan, this property value is 0.	When you create object, reset object, or use it to filter a signal.

Relationships Among Properties

Changing the length of PathDelays also changes the length of AvgPathGaindB, the length of KFactor if KFactor is a vector (no change if it is a scalar), and the length of DopplerSpectrum if DopplerSpectrum is a vector (no change if it is a single object).

DirectPathDopplerShift and DirectPathInitPhase both follow changes in KFactor.

The PathDelays and AvgPathGaindB properties of the channel object must always have the same vector length, because this length equals the number of discrete paths of the channel. The DopplerSpectrum property must either be a single Doppler object or a vector of Doppler objects with the same length as PathDelays.

If you change the length of PathDelays, MATLAB truncates or zero-pads the value of AvgPathGaindB if necessary to adjust its vector length (MATLAB may also change the values of read-only properties such as PathGains and ChannelFilterDelay). If DopplerSpectrum is a vector of Doppler objects, and you increase or decrease the length of PathDelays, MATLAB will add Jakes Doppler objects or remove elements from DopplerSpectrum, respectively, to make it the same length as PathDelays.

If StoreHistory is set to 1 (the default is 0), the object stores channel state information as the channel filter function processes the signal. You can then visualize this state information through a GUI using the plot (channel) method.

Note Setting StoreHistory to 1 will result in a slower simulation. If you do not want to visualize channel state information using plot (channel), but want to access the complex path gains, then set StorePathGains to 1, while keeping StoreHistory as 0.

Reset Method

If `MaxDopplerShift` is set to `0` (the default), the channel object, `chan`, models a static channel.

Use the syntax `reset(chan)` to generate a new channel realization.

Algorithm

The methodology used to simulate fading channels is described in “Methodology for Simulating Multipath Fading Channels”, where the properties specific to the Rician channel object are related to the quantities of this section as follows:

- The `Kfactor` property contains the value of K_r (if it’s a scalar) or $\{K_{r,k}\}$, $1 \le k \le K$ (if it’s a vector).
- The `DirectPathDopplerShift` property contains the value of $f_{d,LOS}$ (if it’s a scalar) or $\{f_{d,LOS,k}\}$, $1 \le k \le K$ (if it’s a vector).
- The `DirectPathInitPhase` property contains the value of θ_{LOS} (if it’s a scalar) or $\{\theta_{LOS,k}\}$, $1 \le k \le K$ (if it’s a vector).

The `rayleighchan` reference page includes descriptions for properties common to both Rayleigh and Rician channel objects.

Channel Visualization

The characteristics of a channel can be plotted using the channel visualization tool, `plot (channel)`. You can use the channel visualization tool in Normal mode and Accelerator mode.

Compatibility Considerations

`ricianchan` has been removed

Errors starting in R2020b

ricianchan has been removed. Use `comm.RicianChannel` instead.

References

[1] Jeruchim, M., Balaban, P., and Shanmugan, K., *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

See Also

`comm.RicianChannel`

Topics

“Fading Channels”

Introduced before R2006a

rls

(To be removed) Construct recursive least squares (RLS) adaptive algorithm object

Note will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead.

Syntax

```
alg = rls(forgetfactor)
alg = rls(forgetfactor,invcorr0)
```

Description

The `rls` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`alg = rls(forgetfactor)` constructs an adaptive algorithm object based on the recursive least squares (RLS) algorithm. The forgetting factor is `forgetfactor`, a real number between 0 and 1. The inverse correlation matrix is initialized to a scalar value.

`alg = rls(forgetfactor,invcorr0)` sets the initialization parameter for the inverse correlation matrix. This scalar value is used to initialize or reset the diagonal elements of the inverse correlation matrix.

Properties

The table below describes the properties of the RLS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Fixed value, `'RLS'`
`ForgetFactor`	Forgetting factor
`InvCorrInit`	Scalar value used to initialize or reset the diagonal elements of the inverse correlation matrix

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` function or `dfe` function), the equalizer object has an `InvCorrMatrix` property that represents the inverse correlation matrix for the RLS algorithm. The initial value of `InvCorrMatrix` is `InvCorrInit*eye(N)`, where `N` is the total number of equalizer weights.

Examples

Configuring Linear Equalizers

This example configures the recommended comm.LinearEqualizer System object™ and the legacy lineareq feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use RLS Algorithm with Linear Equalizer

Configure lineareq and comm.LinearEqualizer objects with comparable settings.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: 0.1000
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When ResetBeforeFiltering is set to true, each call of the equalize object resets the equalizer. To get the equivalent behavior call reset after each call of the comm.LinearEqualizer object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));
```

```
yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. For the `comm.LinearEqualizer` object, set the initial inverse correlation matrix to `eye(5)*0.2`.

```
eqOld = lineareq(5,rls(0.95),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'RLS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ForgetFactor: 0.9500
  InvCorrInit: 0.1000
  InvCorrMatrix: [5×5 double]
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','RLS', ...
    'ForgettingFactor',0.95,'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1, ...
    'InitialInverseCorrelationMatrix',eye(5)*0.2)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'RLS'
  NumTaps: 5
  ForgettingFactor: 0.9500
  InitialInverseCorrelationMatrix: [5×5 double]
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

Algorithms

Referring to the schematics presented in “Equalization”, define w as the vector of all weights w_i and define u as the vector of all inputs u_i. Based on the current set of inputs, u, and the current inverse correlation matrix, P, this adaptive algorithm first computes the Kalman gain vector, K

$$K = \frac{Pu}{(ForgettingFactor) + u^H Pu}\,.$$

where H denotes the Hermitian transpose.

Then the new inverse correlation matrix is given by

$(\texttt{ForgetFactor})^{-1}(P - Ku^{H}P)$

and the new set of weights is given by

$w + K^{*}e$

where the * operator denotes the complex conjugate.

Compatibility Considerations

`rls` will be removed

Warns starting in R2020a

rls will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to RLS. For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-745.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Haykin, S., *Adaptive Filter Theory*, Third Ed., Upper Saddle River, NJ, Prentice-Hall, 1996.

[3] Kurzweil, J., *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

rsdec

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, `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 `rsdec` detects more than `(n-k)/2` errors in a row of `code`. 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 the parity symbols in `code` were appended or prepended to the message in the coding operation. *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`. If *`paritypos`* is `'beginning'`, 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`, the corresponding row in `ccode` contains that row unchanged.

Examples

Reed-Solomon Decoding

Set the RS code parameters.

```
m = 3;                     % Number of bits per symbol
n = 2^m-1;                 % Codeword length
k = 3;                     % Message length
```

Generate three codewords composed of 3-bit symbols. Encode the message with a (7,3) RS code.

```
msg = gf([2 7 3; 4 0 6; 5 1 1],m);
code = rsenc(msg,n,k);
```

Introduce one error on the first codeword, two errors on the second codeword, and three errors on the third codeword.

```
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;
```

Decode the corrupted codeword.

```
[rxcode,cnumerr] = rsdec(noisycode,n,k);
```

Observe that the number of corrected errors matches the introduced errors for the first two rows. In row three, the number of corrected errors is -1 because a (7,3) RS code cannot correct more than two errors.

```
cnumerr
```

```
cnumerr = 3×1

     1
     2
    -1
```

Limitations

n and k must differ by an even integer. n must be between 3 and 65535.

Algorithms

rsdec uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see the works listed in “References” on page 2-750 below.

References

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

[2] Berlekamp, E. R., *Algebraic Coding Theory*, New York, McGraw-Hill, 1968.

See Also

gf | rsenc | rsgenpoly

Topics

“Block Codes”

Introduced before R2006a

rsenc

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, `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`. *`paritypos`* can be either `'end'` or `'beginning'`. The default is `'end'`.

Examples

Reed-Solomon Code Generation

Set the code parameters.

```
m = 3;           % Number of bits per symbol
n = 2^m - 1;     % Codeword length
k = 3;           % Message length
```

Create two messages based on GF(8).

```
msg = gf([2 7 3; 4 0 6],m)
```

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

Array elements =

   2   7   3
   4   0   6
```

Generate RS (7,3) codewords.

```
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
```

The codes are systematic so the first three symbols of each row match the rows of `msg`.

Limitations

`n` and `k` must differ by an integer. `n` between 7 and 65535.

See Also

`gf` | `rsdec` | `rsgenpoly`

Topics

“Block Codes”
“Represent Words for Reed-Solomon Codes”
“Create and Decode Reed-Solomon Codes”

Introduced before R2006a

rsgenpoly

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 = rsgenpoly(N,K,prim_poly,B,outputFormat)
[genpoly,T] = rsgenpoly( ___ )
```

Description

`genpoly = rsgenpoly(N,K)` returns the narrow-sense generator polynomial of an [N,K] Reed-Solomon code. The output `genpoly` is a Galois field array that represents the coefficients of the generator polynomial in order of descending powers. A narrow-sense BCH code is a BCH code with `B` = 1. Here, the narrow-sense generator polynomial is $(X - \alpha^1)(X - \alpha^2)\ldots(X - \alpha^{N-K})$, where α is a root of the default primitive polynomial for the field GF(`N+1`). For additional information, see Narrow-Sense BCH Codes and “Reed-Solomon Codes”.

`genpoly = rsgenpoly(N,K,prim_poly)` also specifies the primitive polynomial, `prim_poly`, for GF(`N+1`) that has α as a root.

`genpoly = rsgenpoly(N,K,prim_poly,B)` returns the generator polynomial, $(X - \alpha^{B})(X - \alpha^{B+1})\ldots(X - \alpha^{B+N-K-1})$, where `B` is an integer.

`genpoly = rsgenpoly(N,K,prim_poly,B,outputFormat)` specifies the output format of `genpoly` as a Galois field array or double-precision array.

`[genpoly,T] = rsgenpoly( ___ )` also returns the error-correction capability of the [N,K] Reed-Solomon code, `T`, using any of the preceding input argument syntaxes.

Examples

Create Narrow-Sense Generator Polynomial

Specify the codeword length, `n`, and message length, `k`.

```
n = 7;
k = 3;
```

Create the narrow-sense generator polynomial for the [n,k] Reed-Solomon code. `genpoly` is a Galois field array, by default, that represents the coefficients of this generator polynomial in order of descending powers.

```
genpoly = rsgenpoly(n,k)
```

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

```
Array elements = 

   1   3   1   2   3
```

Create Narrow-Sense Generator Polynomial Specifying Primitive Polynomial

Create the narrow-sense generator polynomial of a Reed-Solomon code with respect to the primitive polynomial $D^3 + D^2 + 1$.

Specify the codeword length, n, message length, k, and primitive polynomial $D^3 + D^2 + 1$ represented in decimal form.

```
n = 7;
k = 3;
prim_poly = 13;
```

Create the narrow-sense generator polynomial for the [n,k] Reed-Solomon code with respect to primitive polynomial $D^3 + D^2 + 1$ for GF(8). genpoly is a Galois field array, by default, that represent the coefficients of this generator polynomial in order of descending powers.

```
genpoly = rsgenpoly(n,k,prim_poly)
```

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

Array elements = 

   1   4   5   1   5
```

Create Generator Polynomial for Specified B

Create the generator polynomial of a Reed-Solomon code with respect to the default primitive polynomial.

Specify the codeword length, n, message length, k, and exponent of α, b.

```
n = 7;
k = 3;
b = 4;
```

Create the generator polynomial $\left(X - \alpha^4\right)\left(X - \alpha^5\right)\left(X - \alpha^6\right)\left(X - \alpha^7\right)$, with respect to the default primitive polynomial. genpoly is a Galois field array that represents the coefficients of this generator polynomial in order of descending powers. Display the error-correcting capability of the code.

```
[genpoly,t] = rsgenpoly(n,k,[],b)
```

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

Array elements = 
     1     5     5     3     2

t = 2
```

Create Generator Polynomial for DVB-S and WiMAX

Create the generator polynomial of a Reed-Solomon code with respect to the primitive polynomial $D^8 + D^4 + D^3 + D^2 + 1$.

Specify the codeword length, n, message length, k, the primitive polynomial represented in decimal form, and the exponent of α, b.

```
n = 255;
k = 239;
prim_poly = 285;
b = 0;
```

Create the generator polynomial for the [n,k] Reed-Solomon code. genpoly is a Galois field array that represents a generator polynomial and is compliant with DVB-S and WiMAX.

```
genpoly = rsgenpoly(n,k,prim_poly,b)
```

```
genpoly = GF(2^8) array. Primitive polynomial = D^8+D^4+D^3+D^2+1 (285 decimal)

Array elements = 

  Columns 1 through 13

     1    59    13   104   189    68   209    30     8   163    65    41   229

  Columns 14 through 17

    98    50    36    59
```

Create Narrow-Sense Generator Polynomial with Output Format Double

Create the narrow-sense generator polynomial of a Reed-Solomon code. Specify the output data type as a double-precision array.

Specify the codeword length, n, and message length, k.

```
n = 7;
k = 3;
```

Create the narrow-sense generator polynomial for the [n,k] Reed-Solomon code. genpoly is a double-precision array, that represents the coefficients of this generator polynomial in order of descending powers. Specify defaults values for the primitive polynomial and exponent of α inputs by assigning [ ] for them.

```
genpoly = rsgenpoly(n,k,[],[],'double')
genpoly = 1×5

     1     3     1     2     3
```

Input Arguments

N — Codeword length

positive odd integer

Codeword length, specified as an integer of the form `N` = 2^M – 1, where M is in the range [3,16]. For more information, see “Limitations” on page 2-758.

Example: Set `N` to `15` for M=4.

K — Message length

positive integer

Message length, specified as a positive integer. For more information, see “Limitations” on page 2-758.

prim_poly — Primitive polynomial

GF(`N`+1) (default) | positive integer

Primitive polynomial, specified as a positive integer. `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 `[]`. For more information, see “Default Primitive Polynomials” on page 2-758.

Example: `19` specifies the primitive polynomial D^4+D+1 because its binary representation is 10011.

B — Exponent of *α*

`1` (default) | positive integer

Exponent of α, specified as a positive integer. α is a root of `prim_poly`.

outputFormat — Output format

`'gf'` (default) | `'double'`

Output format of `genpoly`, specified as:

- `'gf'` — to output a Galois field array.
- `'double'` — to output a double-precision array of the Galois field values.

For more information, see “Working with Galois Fields”.

Output Arguments

genpoly — Generator polynomial coefficients

Galois field array | double-precision array

Generator polynomial coefficients in descending order, returned as a Galois field array or double-precision array. `genpoly` is a row vector that represents the coefficients of the narrow-sense generator polynomial of an [`N`,`K`] Reed-Solomon code in order of descending powers.

T — Error-correction capability
positive integer

Error-correction capability of the code, returned as a positive integer equal to $\lfloor(N – K)/2\rfloor$.

Limitations

- Valid values for N = 2^M – 1, where M is an integer in the range [3,16]. The maximum allowable value of N = 2^{16} – 1 = 65,535.
- Valid values for K = [1,N – 1].

More About

Default Primitive Polynomials

This table lists the default primitive polynomial used for each Galois field array GF(2^m). To use a different primitive polynomial, specify `prim_poly` as an input argument. `prim_poly` must be in the range [(2^m + 1), (2^{m+1} – 1)] and must indicate an irreducible polynomial. For more information, see “Primitive Polynomials and Element Representations”.

Value of 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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

Functions

`gf` | `gfprimfd` | `rsdec` | `rsenc`

Topics

“Block Codes”
“Parameters for Reed-Solomon Codes”
“Representing Elements of Galois Fields”
“Working with Galois Fields”

Introduced before R2006a

rsgenpolycoeffs

Generator polynomial coefficients of Reed-Solomon code

Syntax

```
x = rsgenpolycoeffs(...)
[x,t] = rsgenpolycoeffs(...)
```

Description

`x = rsgenpolycoeffs(...)` returns the coefficients for the generator polynomial of the Reed-Solomon code. The output is identical to genpoly = rsgenpoly(...); x = genpoly.x.

`[x,t] = rsgenpolycoeffs(...)` returns *t*, the error-correction capability of the code.

Examples

Generate Polynomial Coefficients for a Reed-Solomon Code

This example shows how to generate polynomial coefficients for a (15,11) Reed-Solomon code.

Generate the coefficients using `rsgenpolycoeffs`.

```
genpoly = rsgenpolycoeffs(15,11)
```

```
genpoly = 1x5 uint32 row vector

    1    13    12     8     7
```

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation, these usage notes and limitations apply:

All inputs must be constants. Expressions or variables are allowed if their values do not change.

See Also

`gf` | `rsdec` | `rsenc` | `rsgenpoly`

Introduced in R2010b

scatterplot

Generate scatter plot

Syntax

```
scatterplot(x)
scatterplot(x,n)
scatterplot(x,n,offset)
scatterplot(x,n,offset,plotstring)
scatterplot(x,n,offset,plotstring,scatfig)
scatfig = scatterplot( ___ )
```

Description

`scatterplot(x)` creates a scatter plot for signal `x`.

`scatterplot(x,n)` specifies decimation factor `n`. The function plot every `n`th value of `x`, starting from its first value.

`scatterplot(x,n,offset)` specifies the offset value. The function plots every `n`th value of `x`, starting from its (`offset` + 1)th value.

`scatterplot(x,n,offset,plotstring)` specifies plot attributes for the scatter plot.

`scatterplot(x,n,offset,plotstring,scatfig)` generates the scatter plot in the existing `Figure` object, `scatfig`. To plot multiple signals in the same figure, use `hold on`.

`scatfig = scatterplot( ___ )` returns the `Figure` object of the scatter plot. Use `scatfig` to query or modify properties of the figure after it is created. You can specify any of the input argument combinations from the previous syntaxes.

Examples

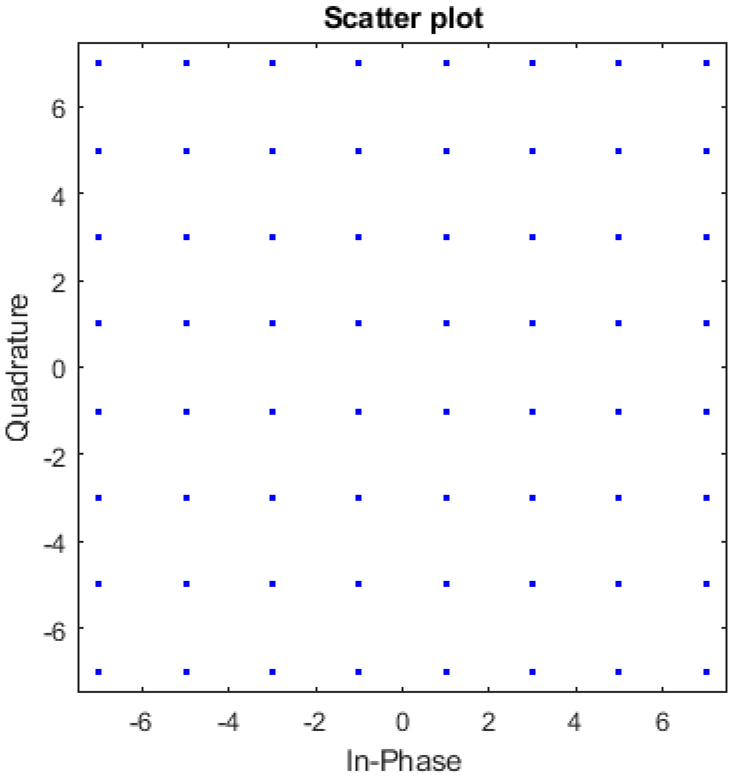
Generate Scatter Plot of 64-QAM Signal

Create a 64-QAM signal in which each constellation point is used.

```
d = (0:63)';
s = qammod(d,64);
```

Display the scatter plot of the constellation.

```
scatterplot(s)
```



Input Arguments

x — Input signal
vector | matrix

Input signal, specified as a vector or matrix.

The interpretation of x depends on its shape and complexity.

- If x is a real-valued two-column matrix, the function interprets the first column as in-phase components and the second column as quadrature components.
- If x is a complex-valued vector, the function interprets the real part as in-phase components and the imaginary part as quadrature components.
- If x is a real-valued vector, the function interprets it as a real signal.

Data Types: double | single
Complex Number Support: Yes

n — Decimation factor
1 (default) | positive integer

Decimation factor, specified as a positive integer. The function plots every nth value of input signal x, starting from its first value.

Data Types: double

`offset` — Offset value

`0` (default) | nonnegative integer

Offset value, specified as a nonnegative integer. This offset value specifies the number of samples at the beginning of input `x` that the function skips before generating the scatter plot.

Data Types: `double`

`plotstring` — Plot attributes

`'b.'` (default) | character vector | string scalar

Plot attributes, specified as a character vector or string scalar containing symbols.

This argument sets the plotting symbol, line type, and color for the scatter plot. The format and meaning of the symbols are the same as in the `plot` function. For example, the default value `'b.'` produces blue dots.

Data Types: `char` | `string`

`scatfig` — Target scatterplot

`Figure` object

Target scatterplot, specified as a `Figure` object for a previously generated `scatterplot`.

Output Arguments

`scatfig` — Target scatterplot

`Figure` object

Target scatterplot, returned as a `Figure` object. To modify properties of this object, see `Figure Properties`.

See Also

Functions
`plot` | `scatter`

Objects
`comm.ConstellationDiagram`

Topics
"Scatter Plots and Constellation Diagrams"

Introduced before R2006a

semianalytic

Calculate bit error rate (BER) using semianalytic technique

Syntax

```
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,EbNo)
ber = semianalytic(txsig,rxsig,modtype,M,Nsamp,num,den,EbNo)
[ber,avgampl,avgpower] = semianalytic(...)
```

Description

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp)` returns the bit error rate (BER) of a system that transmits the complex baseband vector signal `txsig` and receives the noiseless complex baseband vector signal `rxsig`. Each of these signals has `Nsamp` samples per symbol. `Nsamp` is also the sampling rate of `txsig` and `rxsig`, in Hz. The function assumes that `rxsig` is the input to the receiver filter, and the function filters `rxsig` with an ideal integrator. `modtype` is the modulation type of the signal and *`M`* is the alphabet size. The table below lists the valid values for *`modtype`* and *`M`*.

Modulation Scheme	Value of *modtype*	Valid Values of *M*
Differential phase shift keying (DPSK)	`'dpsk'`	2, 4
Minimum shift keying (MSK) with differential encoding	`'msk/diff'`	2
Minimum shift keying (MSK) with nondifferential encoding	`'msk/nondiff'`	2
Phase shift keying (PSK) with differential encoding, where the phase offset of the constellation is 0	`'psk/diff'`	2, 4
Phase shift keying (PSK) with nondifferential encoding, where the phase offset of the constellation is 0	`'psk/nondiff'`	2, 4, 8, 16, 32, or 64
Offset quadrature phase shift keying (OQPSK)	`'oqpsk'`	4
Quadrature amplitude modulation (QAM)	`'qam'`	4, 8, 16, 32, 64, 128, 256, 512, 1024

`'msk/diff'` is equivalent to conventional MSK (setting the `'Precoding'` property of the MSK object to `'off'`), while `'msk/nondiff'` is equivalent to precoded MSK (setting the `'Precoding'` property of the MSK object to `'on'`).

Note The output `ber` is an *upper bound* on the BER in these cases:

- DQPSK (*modtype* = `'dpsk'`, `M` = 4)
- Cross QAM (*modtype* = `'qam'`, `M` not a perfect square). In this case, note that the upper bound used here is slightly tighter than the upper bound used for cross QAM in the `berawgn` function.

When the function computes the BER, it assumes that symbols are Gray-coded. The function calculates the BER for values of E_b/N_0 in the range of [0:20] dB and returns a vector of length 21 whose elements correspond to the different E_b/N_0 levels.

Note You must use a sufficiently long vector `txsig`, or else the calculated BER will be inaccurate. If the system's impulse response is L symbols long, the length of `txsig` should be at least $\mathtt{M}^L$. A common approach is to start with an augmented binary pseudonoise (PN) sequence of total length $(\mathtt{log_2M})\mathtt{M}^L$. An *augmented* PN sequence is a PN sequence with an extra zero appended, which makes the distribution of ones and zeros equal.

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,num,den)` is the same as the previous syntax, except that the function filters `rxsig` with a receiver filter instead of an ideal integrator. The transfer function of the receiver filter is given in descending powers of z by the vectors `num` and `den`.

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,EbNo)` is the same as the first syntax, except that `EbNo` represents E_b/N_0, the ratio of bit energy to noise power spectral density, in dB. If `EbNo` is a vector, then the output `ber` is a vector of the same size, whose elements correspond to the different E_b/N_0 levels.

`ber = semianalytic(txsig,rxsig,`*`modtype`*`,`*`M`*`,Nsamp,num,den,EbNo)` combines the functionality of the previous two syntaxes.

`[ber,avgampl,avgpower] = semianalytic(...)` returns the mean complex signal amplitude and the mean power of `rxsig` after filtering it by the receiver filter and sampling it at the symbol rate.

Examples

A typical procedure for implementing the semianalytic technique is in “Procedure for the Semianalytic Technique”. Sample code is in “Using Semianalytic Technique”.

Limitations

The function makes several important assumptions about the communication system. See “When to Use the Semianalytic Technique” to find out whether your communication system is suitable for the semianalytic technique and the `semianalytic` function.

Alternatives

As an alternative to the `semianalytic` function, invoke the BERTool GUI (`bertool`) and use the **Semianalytic** tab.

References

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

[2] Pasupathy, S., “Minimum Shift Keying: A Spectrally Efficient Modulation,” *IEEE Communications Magazine*, July, 1979, pp. 14–22.

See Also

`noisebw` | `qfunc`

Topics

“Performance Results via the Semianalytic Technique”

Introduced before R2006a

shift2mask

Convert shift to mask vector for 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 polynomial character vector
- 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.

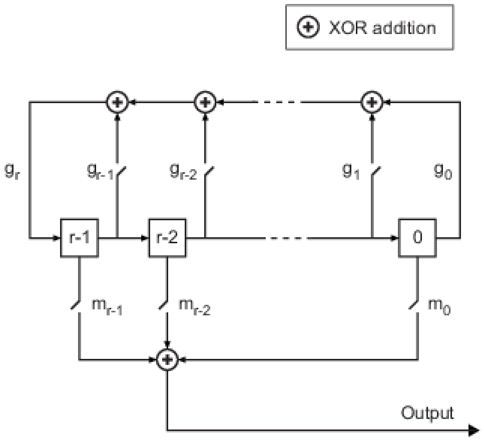
Note To save time, `shift2mask` does not check that `prpoly` is primitive. If it is not primitive, the output is not meaningful. To find primitive polynomials, use `primpoly` or see [2].

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, and 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, 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 Communications Toolbox software and want to generate a pseudonoise sequence in a Simulink model, see the PN Sequence Generator block reference page.

Examples

Convert Shift to Mask

Convert a shift in a linear feedback shift register into an equivalent mask.

Convert 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 $x^4 + x^3 + 1$. The length of the mask is equal to the degree of the primitive polynomial, 4.

```
mk = shift2mask([1 1 0 0 1],5)
```

```
mk = 1×4

     1     0     1     1
```

Convert a shift of 7 to a mask of $x^4 + x^2$ for the primitive polynomial $x^5 + x^2 + 1$.

```
mk2 = shift2mask('x5+x2+1',7)
```

```
mk2 = 1×5

     1     0     1     0     0
```

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.

See Also

`deconv` | `isprimitive` | `mask2shift` | `primpoly`

Introduced before R2006a

showcommblockdatatypetable

Communications Toolbox block characteristics

Syntax

```
showcommblockdatatypetable
```

Description

`showcommblockdatatypetable` shows a table of characteristics for the Communications Toolbox blocks. The table lists capabilities and limitations about code generation, variable size, and supported data types for each block. If a cell includes an "X", the corresponding block supports the capability indicated by the column heading. Descriptions for numbered footnotes, "(#)", follow the table.

Examples

Show Communications Toolbox Block Characteristics

Show a table of Communications Toolbox block characteristics. The table opens in a separate window.

```
showcommblockdatatypetable
```

```
Loading Communications Toolbox Library.
```

See Also

Topics
"Block Characteristics"

Introduced in R2008b

signlms

(To be removed) Construct signed least mean square (LMS) adaptive algorithm object

Note will be removed in a future release. Consider using `comm.LinearEqualizer` or `comm.DecisionFeedback` instead.

Syntax

```
alg = signlms(stepsize)
alg = signlms(stepsize,algtype)
```

Description

The `signlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`alg = signlms(stepsize)` constructs an adaptive algorithm object based on the signed least mean square (LMS) algorithm with a step size of `stepsize`.

`alg = signlms(stepsize,`*`algtype`*`)` constructs an adaptive algorithm object of type *`algtype`* from the family of signed LMS algorithms. The table below lists the possible values of *`algtype`*.

Value of *algtype*	Type of Signed LMS Algorithm
`'Sign LMS'`	Sign LMS (default)
`'Signed Regressor LMS'`	Signed regressor LMS
`'Sign Sign LMS'`	Sign-sign LMS

Properties

The table below describes the properties of the signed LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Type of signed LMS algorithm, corresponding to the *`algtype`* input argument. You cannot change the value of this property after creating the object.
`StepSize`	LMS step size parameter, a nonnegative real number
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Algorithms

Referring to the schematics presented in “Equalization”, define *w* as the vector of all weights w_i and define *u* as the vector of all inputs u_i. Based on the current set of weights, *w*, this adaptive algorithm creates the new set of weights given by

- `(LeakageFactor) w + (StepSize) u`*`sgn(Re(e))`, for sign LMS
- `(LeakageFactor) w + (StepSize) sgn(Re(u)) Re(e)`, for signed regressor LMS
- `(LeakageFactor) w + (StepSize) sgn(Re(u)) sgn(Re(e))`, for sign-sign LMS

where the * operator denotes the complex conjugate and `sgn` denotes the signum function (`sign` in MATLAB technical computing software).

Compatibility Considerations

`signlms` will be removed

Warns starting in R2020a

- signlms will be removed in a future release. Consider using `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to LMS.
- The `comm.LinearEqualizer` or `comm.DecisionFeedback` System objects do not have a leakage factor property. This is equivalent to setting `LeakageFactor` to `1` in the signlms function.
- For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-772.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, John Wiley & Sons, 1998.

[2] Kurzweil, J., *An Introduction to Digital Communications*, New York, John Wiley & Sons, 2000.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

ssbdemod

Single sideband amplitude demodulation

Syntax

```
z = ssbdemod(y,Fc,Fs)
z = ssbdemod(y,Fc,Fs,ini_phase)
z = ssbdemod(y,Fc,Fs,ini_phase,num,den)
```

Description

For All Syntaxes

`z = ssbdemod(y,Fc,Fs)` demodulates the single sideband amplitude modulated signal `y` from the carrier signal having frequency `Fc` (Hz). The carrier signal and `y` have sampling rate `Fs` (Hz). The modulated signal has zero initial phase, and can be an upper- or lower-sideband signal. The demodulation process uses the lowpass filter specified by `[num,den] = butter(5,Fc*2/Fs)`.

Note The `Fc` and `Fs` arguments must satisfy `Fs` > 2(`Fc` + `BW`), where `BW` is the bandwidth of the original signal that was modulated.

`z = ssbdemod(y,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

`z = ssbdemod(y,Fc,Fs,ini_phase,num,den)` specifies the numerator and denominator of the lowpass filter used in the demodulation.

Examples

Demodulate Sideband Signal

Define the sampling frequency and original signal.

```
fs = 270000;
t = (0:1/fs:0.01)';
signal = sin(2*pi*300.*t)+2*sin(2*pi*600.*t);
```

Convert the original signal to upper-sideband and lower-sideband modulated signals using `ssbmod`. Use a cutoff frequency of `12000` and an initial phase of `0`.

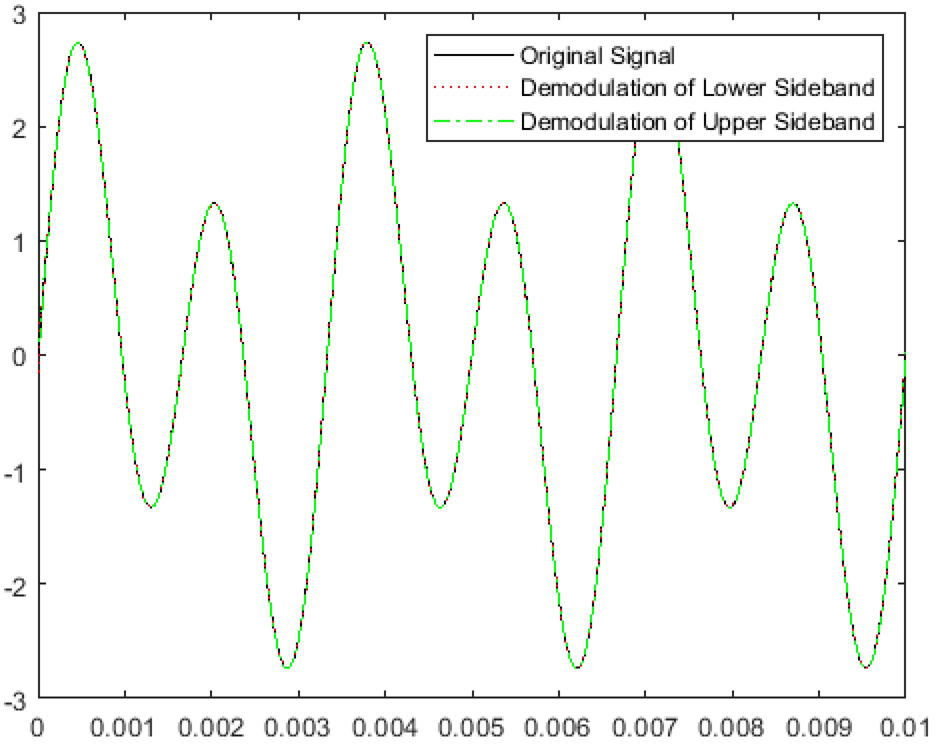
```
fc = 12000;
initialPhase = 0;
lowerSidebandSignal = ssbmod(signal,fc,fs,initialPhase);
upperSidebandSignal = ssbmod(signal,fc,fs,initialPhase,'upper');
```

Demodulate the lower and upper sideband signals.

```
s1 = ssbdemod(lowerSidebandSignal,fc,fs);
s2 = ssbdemod(upperSidebandSignal,fc,fs);
```

Compare processed signals with original and verify reconstruction.

```
plot(t,signal,'k',t,s1,'r:',t,s2,'g-.');
legend('Original Signal','Demodulation of Lower Sideband','Demodulation of Upper Sideband');
```



See Also

`amdemod` | `ssbmod`

Topics

“Digital Modulation”

Introduced before R2006a

ssbmod

Single sideband amplitude modulation

Syntax

```
y = ssbmod(x,Fc,Fs)
y = ssbmod(x,Fc,Fs,ini_phase)
y = ssbmod(x,fc,fs,ini_phase,'upper')
```

Description

`y = ssbmod(x,Fc,Fs)` uses the message signal `x` to modulate a carrier signal with frequency `Fc` (Hz) using single sideband amplitude modulation in which the lower sideband is the desired sideband. The generated output `y` is a single side band signal with a suppressed carrier. The carrier signal and `x` have sample frequency `Fs` (Hz). The modulated signal has zero initial phase.

`y = ssbmod(x,Fc,Fs,ini_phase)` specifies the initial phase of the modulated signal in radians.

`y = ssbmod(x,fc,fs,ini_phase,'upper')` uses the upper sideband as the desired sideband.

Examples

Compare Double-Sideband and Single-Sideband Amplitude Modulation

Set the sample rate to 100 Hz. Create a time vector 100 seconds long.

```
fs = 100;
t = (0:1/fs:100)';
```

Set the carrier frequency to 10 Hz. Generate a sinusoidal signal.

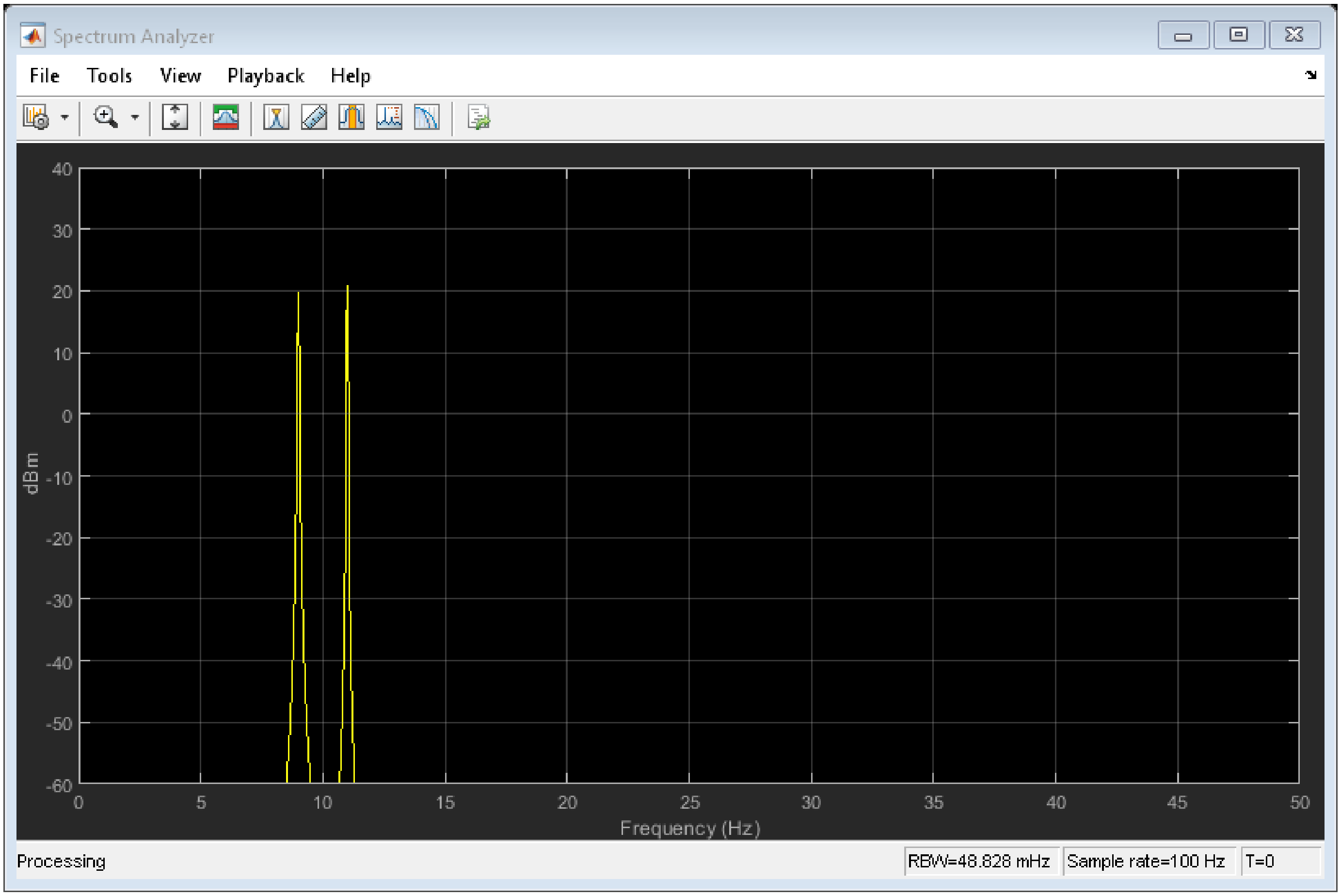
```
fc = 10;
x = sin(2*pi*t);
```

Modulate `x` using single- and double-sideband AM.

```
ydouble = ammod(x,fc,fs);
ysingle = ssbmod(x,fc,fs);
```

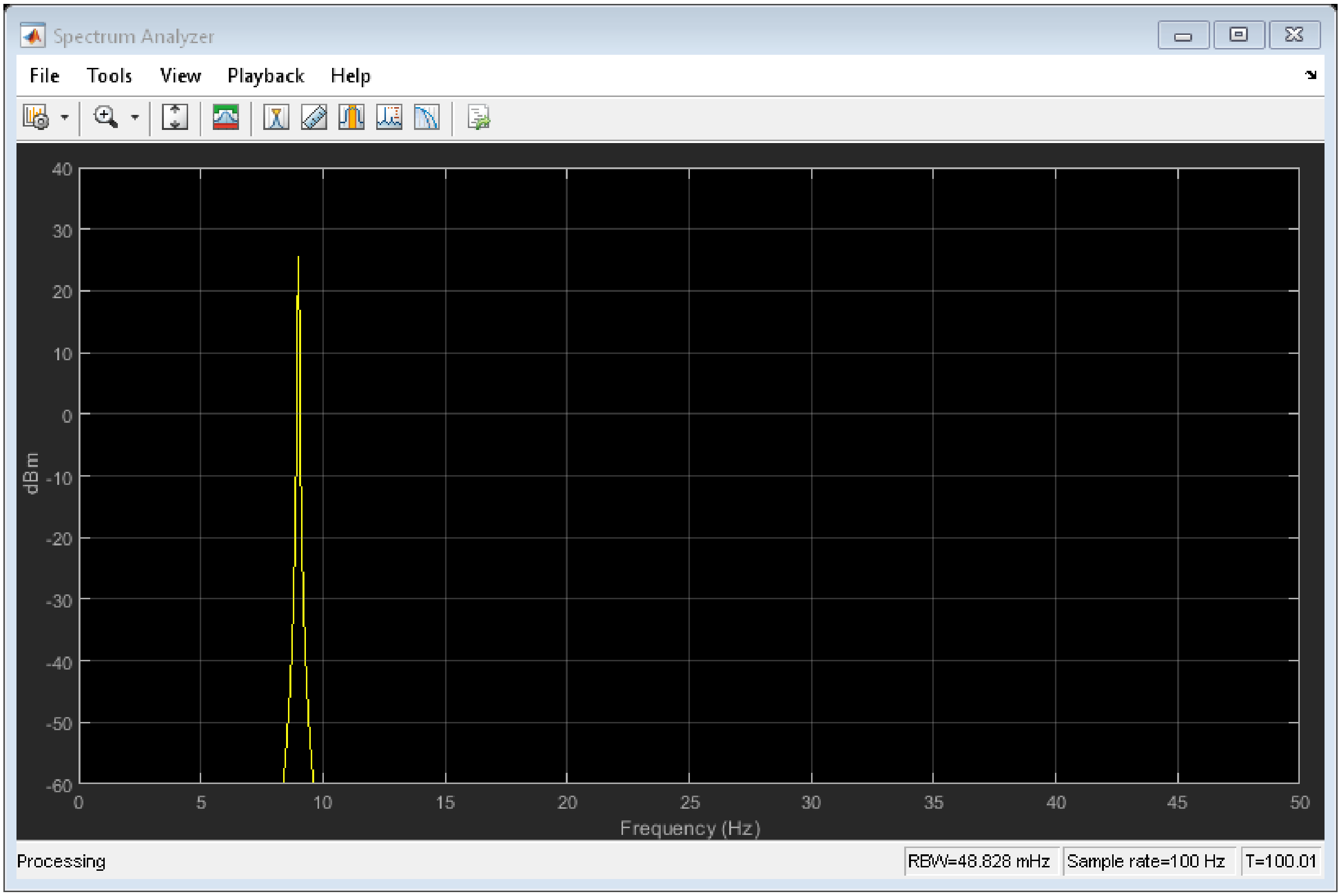
Create a spectrum analyzer object to plot the spectra of the two signals. Plot the spectrum of the double-sideband signal.

```
sa = dsp.SpectrumAnalyzer('SampleRate',fs, ...
    'PlotAsTwoSidedSpectrum',false, ...
    'YLimits',[-60 40]);
step(sa,ydouble)
```



Plot the single-sideband spectrum.

```
step(sa,ysingle)
```



See Also

`ammod` | `ssbdemod`

Topics

"Digital Modulation"

Introduced before R2006a

stdchan

Construct channel System object from set of standardized channel models

Syntax

```
chan = stdchan(chantype,rs,fd)
```

Description

`chan = stdchan(chantype,rs,fd)` constructs a fading channel object `chan` according to the specified `chantype`. `chantype` is chosen from the channel models listed in “Supported Standards” on page 2-789. `rs` is the sampling rate of the input signal and `fd` is the maximum Doppler shift.

Examples

Filter Signal Through CDMA Channel

Set the sample rate and the maximum Doppler shift.

```
rs = 20e6;
fd = 3;
```

Create a CDMA Typical Urban channel model (TUx) channel object and turn on frequency response visualization.

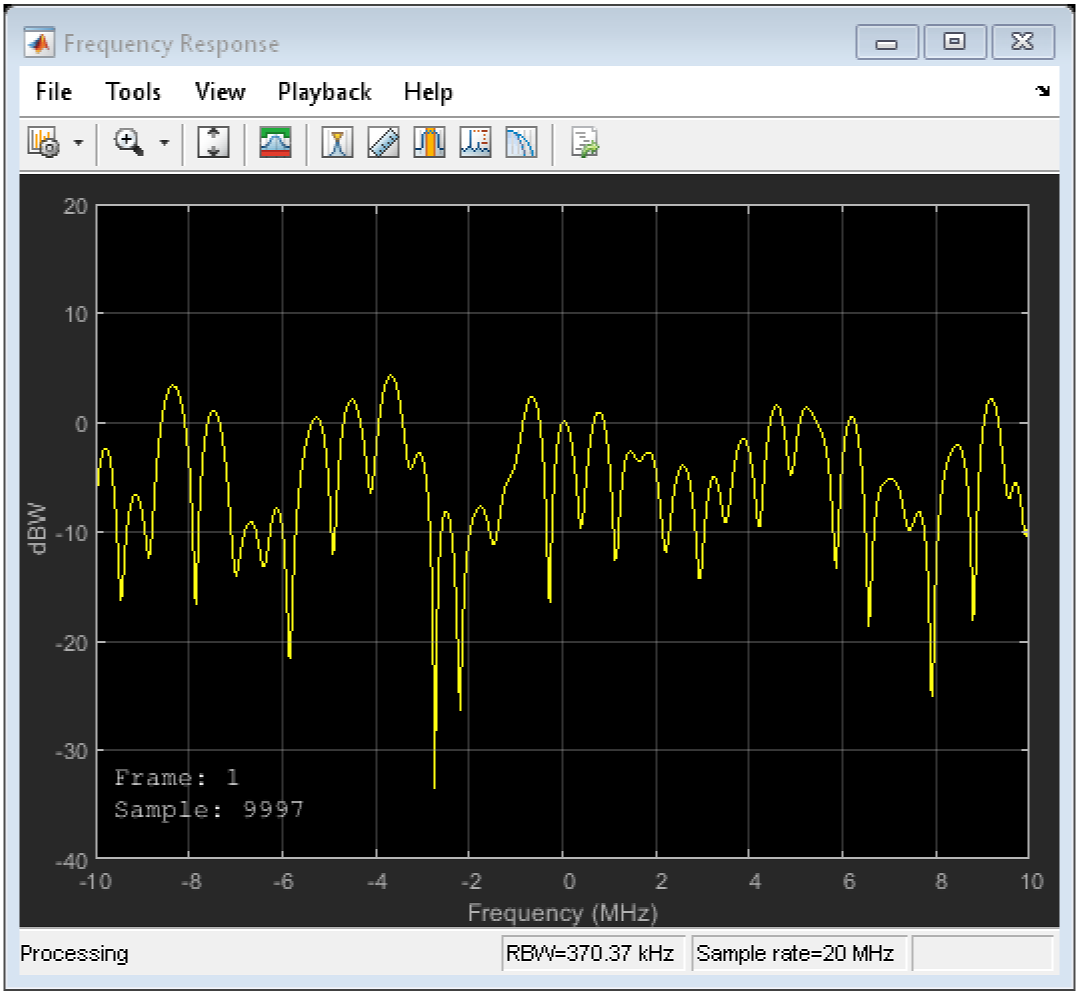
```
chan = stdchan('cdmaTUx',rs,fd);
chan.Visualization = 'Frequency response';
```

Generate random data and apply QPSK modulation.

```
data = randi([0 3],10000,1);
txSig = pskmod(data,4,pi/4);
```

Filter the QPSK signal through the CDMA channel.

```
y = chan(txSig);
```



GSM and EDGE Channel Model

Create a channel model useful for GSM and EDGE simulations. Experiment with low speed and high speed conditions.

Configure parameters and System objects

Frame configuration.

```
M = 8; % Modulation order, 8-PSK
Rbit = 9600; % Input bit rate
Rs = Rbit / log2(M); % Symbol rate
Nsamples = 5e2; % Number of samples per frame
Nframes = 10; % Number of frames
```

Speed and channel configuration.

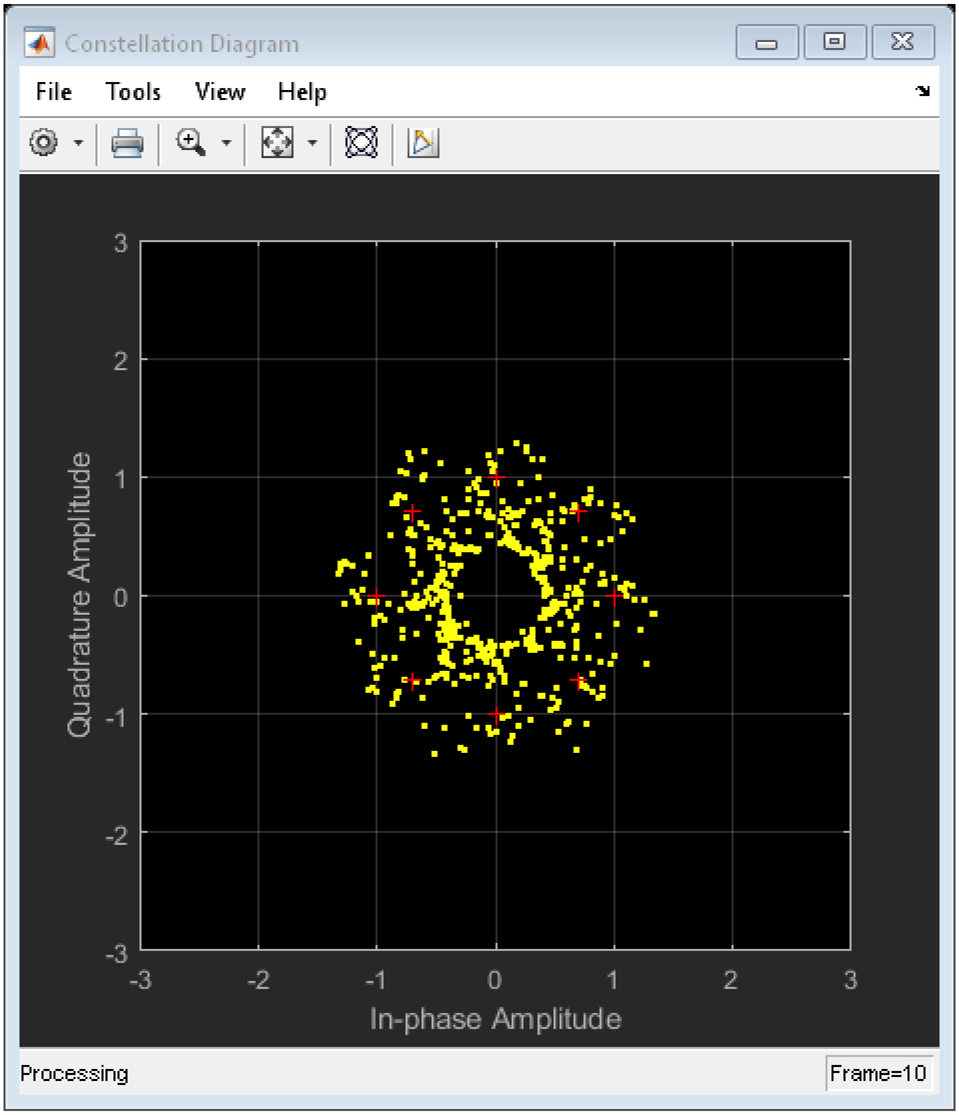
```
v = 10 * 1e3/3600; % Mobile speed (m/s)
fc = 1800e6; % Carrier frequency
c = physconst('LightSpeed'); % Speed of light in free space
fd = v*fc/c; % Maximum Doppler shift of diffuse component
```

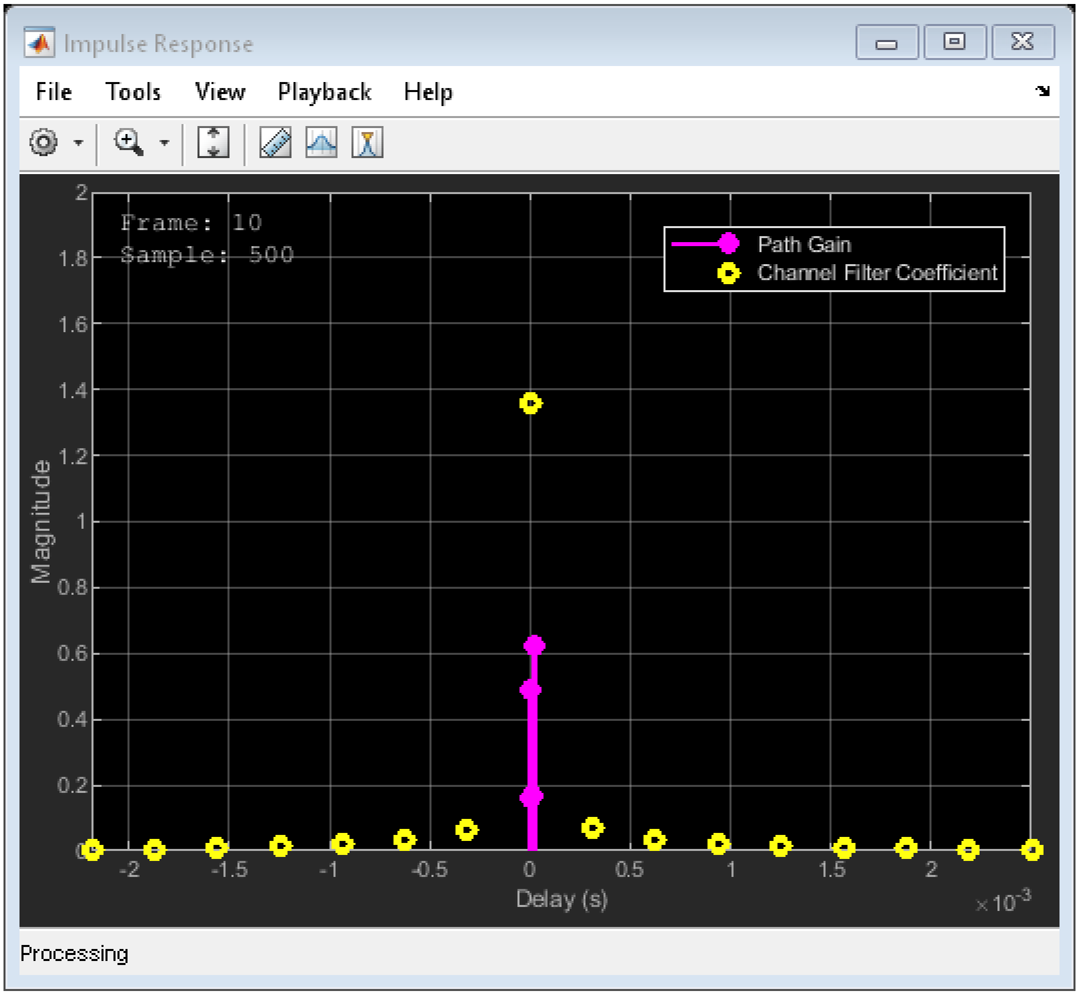
Create System objects for modulator and channel.

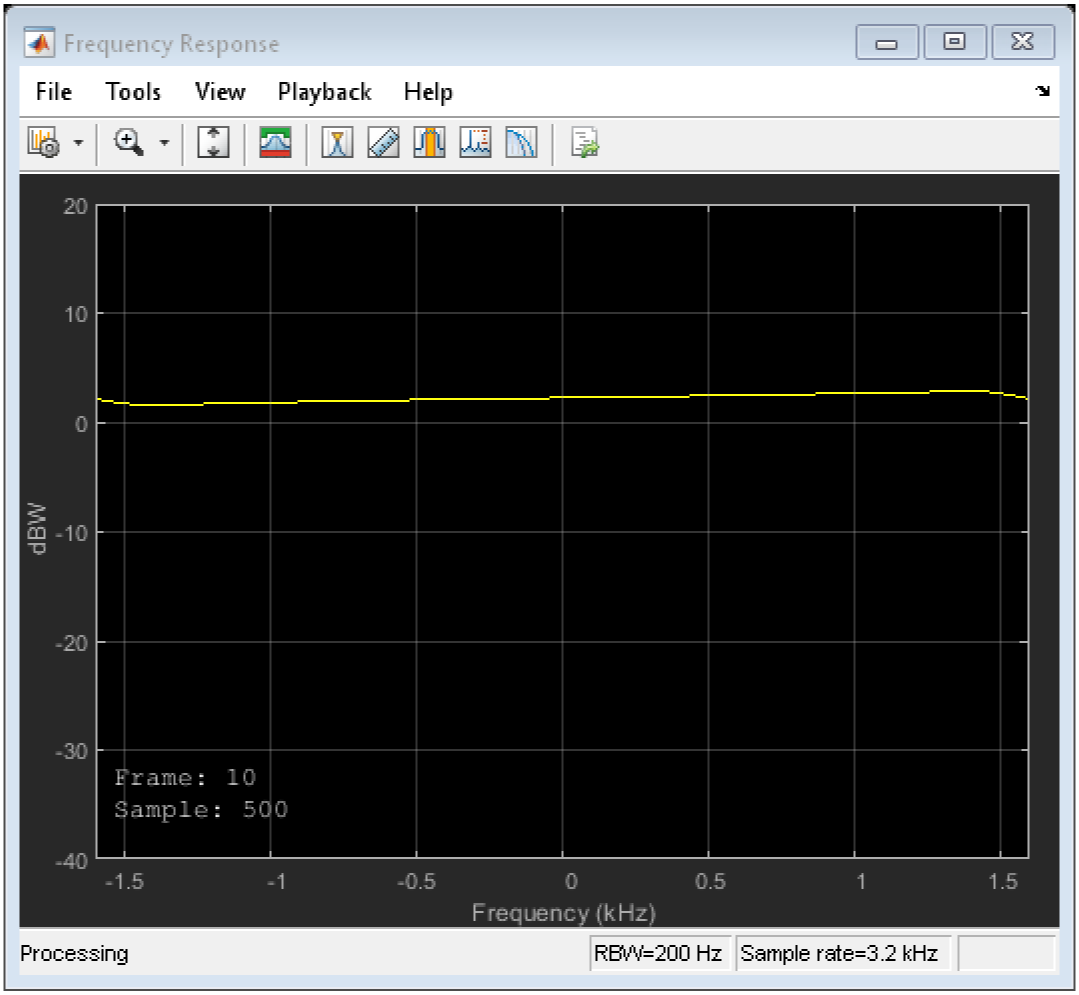
```
modulator  = comm.PSKModulator(M,'PhaseOffset',0);

channel = stdchan('gsmeqx6',Rs,fd);
channel.RandomStream = 'mt19937ar with seed'; % set for reproducibility
channel.Visualization = 'Impulse and frequency responses';
channel.SamplesToDisplay = '100%';

refC = constellation(modulator);
constDiagram = comm.ConstellationDiagram( ...
    'ReferenceConstellation',refC, ...
    'XLimits',[-3 3],'YLimits',[-3 3]);
```

Simulate at low speed

```
for iFrames = 1:Nframes
    msg = randi([0 M-1], Nsamples, 1);
    modSignal = modulator(msg);
    chanOut = channel(modSignal);
    constDiagram(chanOut);
end
```


Constellation Diagram
File
Tools
View
Help
3
2
1
0
-1
-2
-3
-3
-2
-1
0
1
2
3
Quadrature Amplitude
In-phase Amplitude
Processing
Frame=10

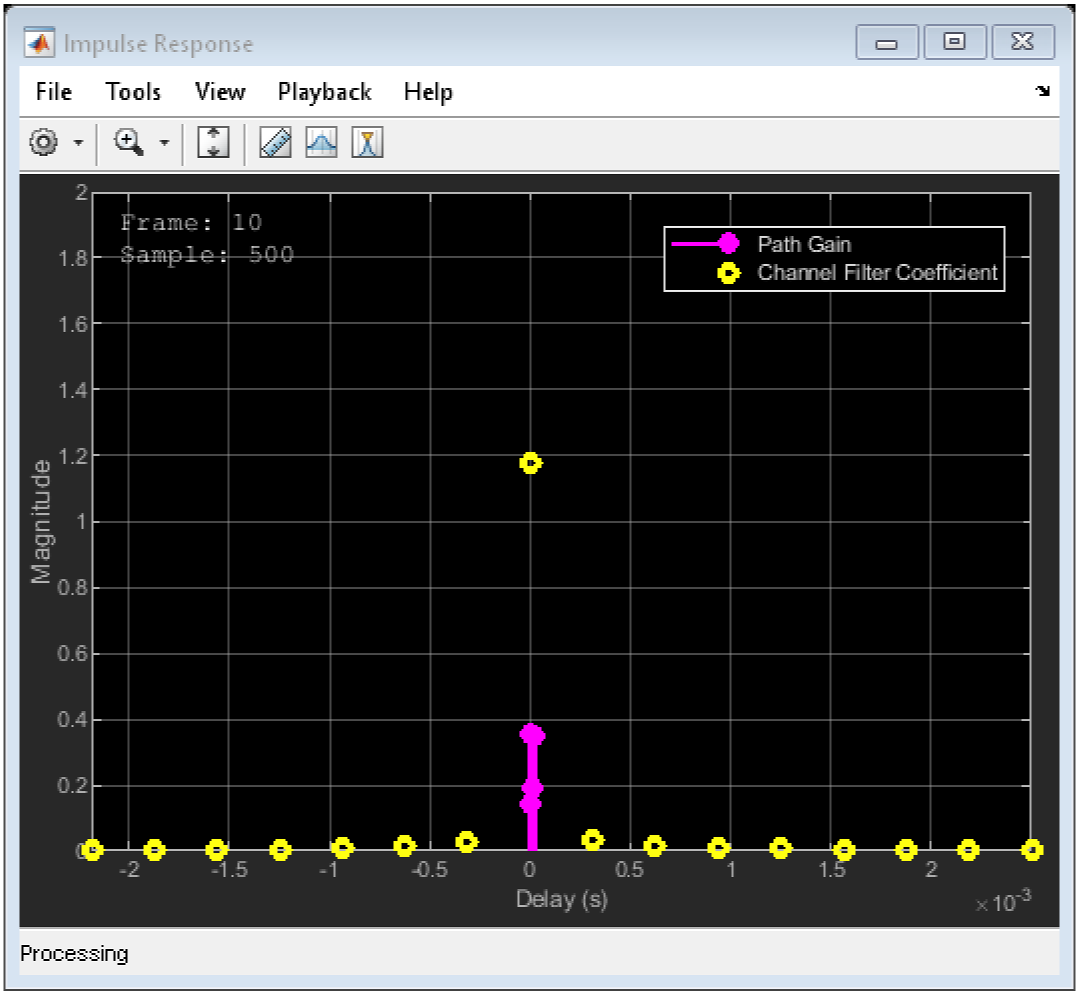

Impulse Response
File
Tools
View
Playback
Help
Frame: 10
Sample: 500
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Delay (s)
× 10^-3
Processing

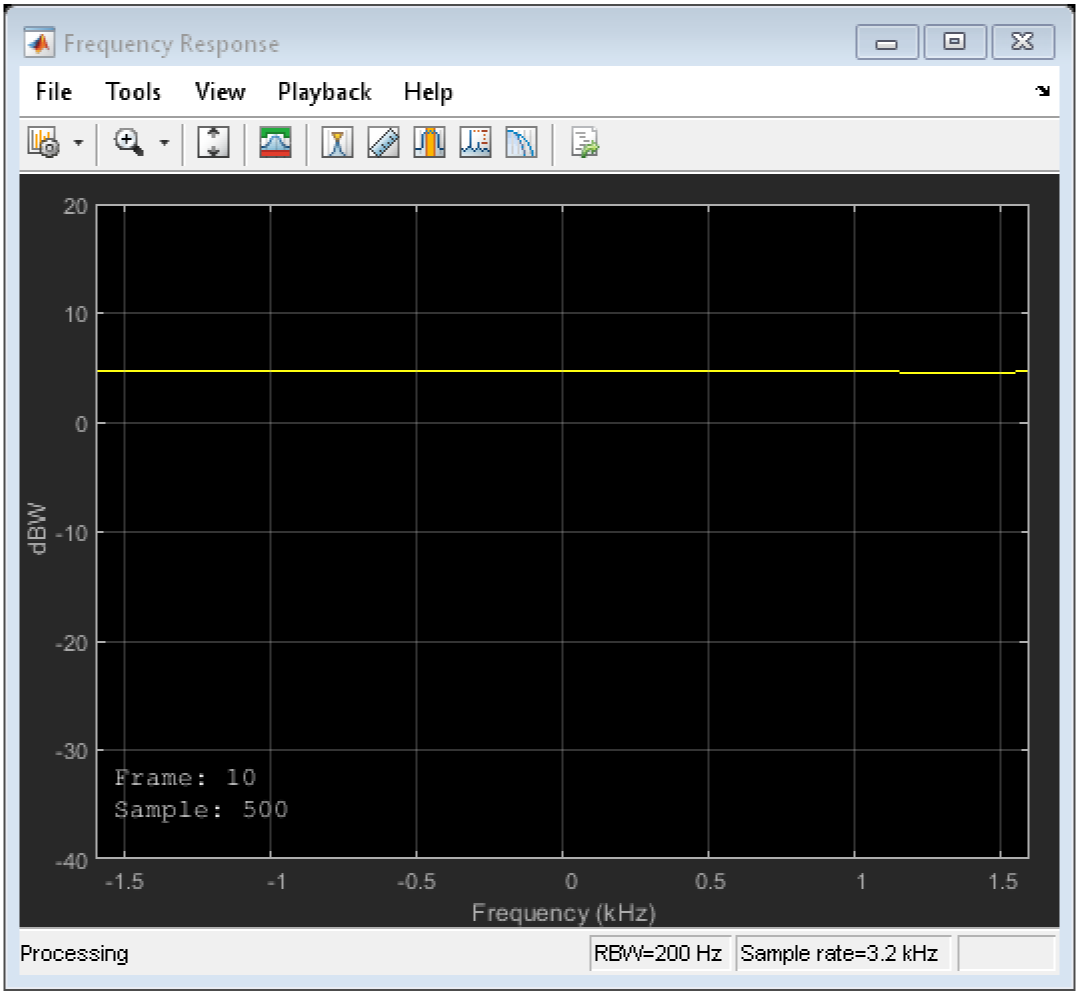


Simulate at high speed

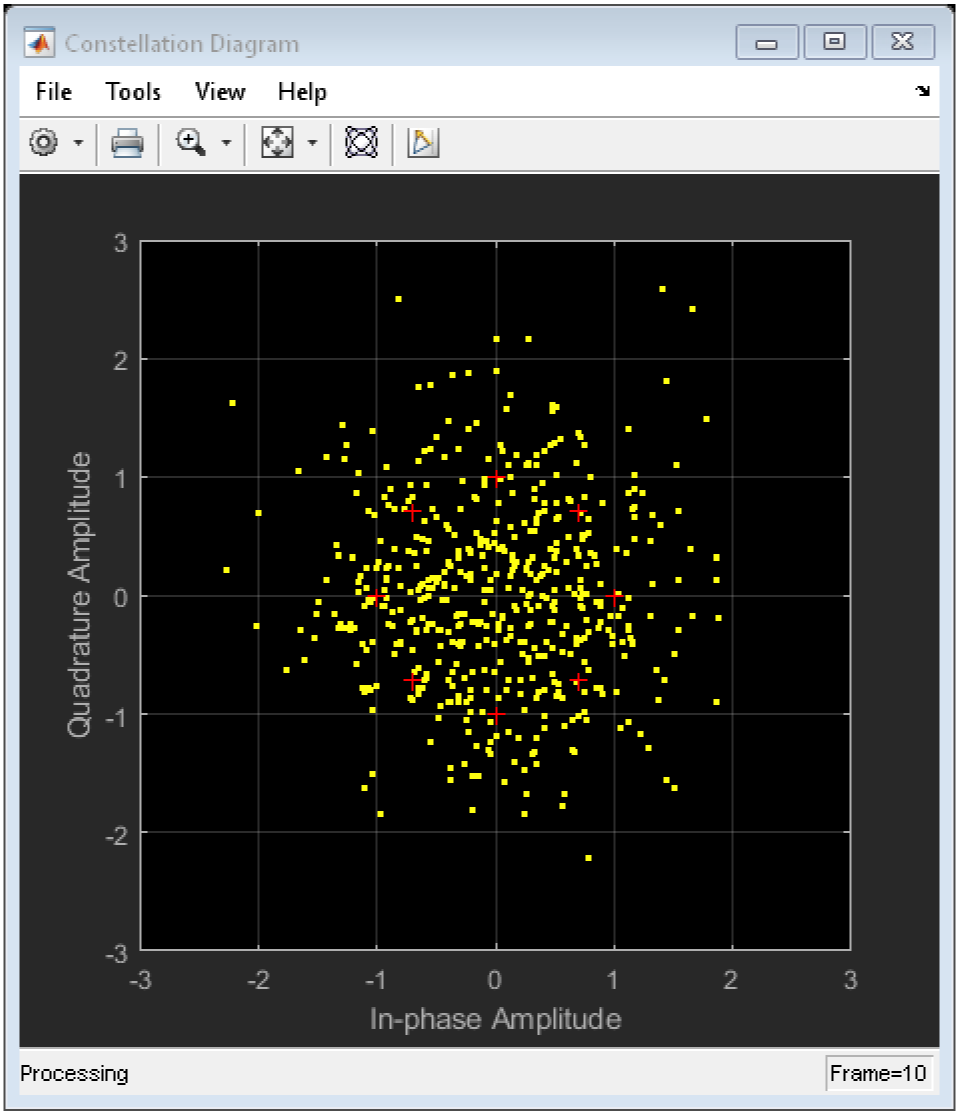
Release and reconfigure objects.

```
release(constDiagram);
release(channel);

v = 120 * 1e3 / 3600; % Mobile speed (m/s)
fd = v*fc/c; % Maximum Doppler shift of diffuse component

channel.MaximumDopplerShift = fd; % Adjust maximum doppler shift

for iFrames = 1:Nframes
    msg = randi([0 M-1], Nsamples, 1);
    modSignal = modulator(msg);
    chanOut = channel(modSignal);
    constDiagram(chanOut);
end
```


Impulse Response
File
Tools
View
Playback
Help
Frame: 10
Sample: 500
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
Delay (s)
×10^-3
Processing



Frame: 10
Sample: 500



Input Arguments

chantype — Channel type

string | character vector

Channel type, specified as a string or character vector. Valid options are listed in “Supported Standards” on page 2-789.

Example: `stdchan('gsmRAx6c1',rs,fd)`, configures a channel model for the GSM typical case for rural area (RAx), 6 taps, case 1, with a sample rate `rs`, and maximum Doppler shift `fd`

Data Types: `char` | `string`

rs — Sample rate

scalar

Sample rate in Hertz, specified as a scalar.

Data Types: `double`

fd — Maximum Doppler shift
scalar

Maximum Doppler shift in Hertz, specified as a scalar.

Data Types: `double`

Output Arguments

chan — Channel object
System object

Channel object, returned as a `comm.RayleighChannel` or `comm.RicianChannel` System object.

More About

Supported Standards

For GSM, CDMA, and ITU-R HF standards, call `stdchan` to return a `comm.RayleighChannel` or `comm.RicianChannel` System object modeling one of these profiles.

GSM/EDGE channel models (3GPP TS 45.005 V7.9.0 (2007-2), 3GPP TS 05.05 V8.20.0 (2005-11)):

Channel model	Profile
`gsmRAx6c1`	Typical case for rural area (RAx), 6 taps, case 1
`gsmRAx4c2`	Typical case for rural area (RAx), 4 taps, case 2
`gsmHTx12c1`	Typical case for hilly terrain (HTx), 12 taps, case 1
`gsmHTx12c2`	Typical case for hilly terrain (HTx), 12 taps, case 2
`gsmHTx6c1`	Typical case for hilly terrain (HTx), 6 taps, case 1
`gsmHTx6c2`	Typical case for hilly terrain (HTx), 6 taps, case 2
`gsmTUx12c1`	Typical case for urban area (TUx), 12 taps, case 1
`gsmTUx12c1`	Typical case for urban area (TUx), 12 taps, case 2
`gsmTUx6c1`	Typical case for urban area (TUx), 6 taps, case 1
`gsmTUx6c2`	Typical case for urban area (TUx), 6 taps, case 2
`gsmEQx6`	Profile for equalization test (EQx), 6 taps
`gsmTIx2`	Typical case for very small cells (TIx), 2 taps

CDMA channel models for deployment evaluation (3GPP TR 25.943 V6.0.0 (2004-12)):

Channel model	Profile
`cdmaTUx`	Typical Urban channel model (TUx)
`cdmaRAx`	Rural Area channel model (RAx)

Channel model	Profile
`cdmaHTx`	Hilly Terrain channel model (HTx)

ITU-R HF channel models (ITU-R F.1487 (2000)) (FD must be 1 to obtain the correct frequency spreads for these models.):

Channel model	Profile
`iturHFLQ`	Low latitudes, Quiet conditions
`iturHFLM`	Low latitudes, Moderate conditions
`iturHFLD`	Low latitudes, Disturbed conditions
`iturHFMQ`	Medium latitudes, Quiet conditions
`iturHFMM`	Medium latitudes, Moderate conditions
`iturHFMD`	Medium latitudes, Disturbed conditions
`iturHFMDV`	Medium latitudes, Disturbed conditions near vertical incidence
`iturHFHQ`	High latitudes, Quiet conditions
`iturHFHM`	High latitudes, Moderate conditions
`iturHFHD`	High latitudes, Disturbed conditions

Compatibility Considerations

`stdchan(ts,fd,channType)` syntax has been removed

stdchan`(ts,fd,channType)` syntax has been removed.

Compatibility considerations for the `stdchan` function includes addition of a function syntax, removal of a function syntax, and removal of configuration support for several channel models.

- The syntax `chan = stdchan(ts,fd,chantype)` has been removed. A System object is returned using the new syntax.
- `stdchan` has removed support for configuration of several channel models by supported standards and associated syntax, compatibility considerations are indicated here:

Standard	Previous Syntax	New Syntax to Return System Object	Notes
3GPP, CDMA	`stdchan(ts,fd,'3gppXXX')`	`stdchan('cdmaXXX',rs,fd)`	Prefix changed from '`3gpp`' to '`cdma`'. `ts` and `rs` are reciprocal values.
GSM	`stdchan(ts,fd,'gsmXXX')`	`stdchan('gsmXXX',rs,fd)`	`ts` and `rs` are reciprocal values.
ITU-R HF	`stdchan(ts,fd,'iturHFXXX')`	`stdchan('iturHFXXX',rs,fd)`	`ts` and `rs` are reciprocal values.

COST207	`stdchan(ts,fd,'cost207XXX')`	N/A	In the future `stdchan` will not configure these channels. Use `comm.RayleighChannel` or `comm.RicianChannel` to configure the channel models for COST207, ITU-R 3G, JTC, HIPERLAN/2, and 802.11a/b/g standards. For guidance mapping parameters, see “Rayleigh Channel Compatibility Considerations” and “Rician Channel Compatibility Considerations”.
ITU-R 3G	`stdchan(ts,fd,'itur3GXXX')`	N/A	
JTC	`stdchan(ts,fd,'jtcXXX')`	N/A	
HIPERLAN/2	`stdchan(ts,fd,'hiperlan2XXX')`	N/A	
802.11a/b/g	`stdchan(ts,fd,'802.11X')`	N/A	

See Also

Functions
`doppler`

Objects
`comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2007b

symerr

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

`[number,ratio] = symerr(x,y)` compares the elements in `x` and `y`. The sizes of `x` and `y` determine which elements are compared. The output `number` is a scalar or vector that indicates the number of elements that differ. The output `ratio` equals `number` divided by the total number of elements in the *smaller* input.

`[number,ratio] = symerr(x,y,flg)` compares the elements in `x` and `y`. Optional input *`flg`* and the size of x, and y, determine the size of `number`.

`[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

Compare Elements of Matrix

Compare Elements of Matrix with Another Matrix

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4

     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aMatrix = [1,1,1,1;2,2,2,2;3,3,3,3]
```

```
aMatrix = 3×4

     1     1     1     1
     2     2     2     2
     3     3     3     3
```

```
[number1,ratio1] = symerr(x,aMatrix)
```

```
number1 = 3
```

```
ratio1 = 0.2500
```

Compare Elements of Matrix with Row Vector

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4

     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aRowVector = [1,2,3,1]
```

```
aRowVector = 1×4

     1     2     3     1
```

```
[number2,ratio2] = symerr(x,aRowVector)
```

```
number2 = 3×1

     1
     3
     4
```

```
ratio2 = 3×1

    0.2500
    0.7500
    1.0000
```

Compare Elements of Matrix with Column Vector

```
x = [1,1,3,1;3,2,2,2;3,3,8,3]
```

```
x = 3×4

     1     1     3     1
     3     2     2     2
     3     3     8     3
```

```
aColumnVector = [1;2;3]
```

```
aColumnVector = 3×1

     1
     2
     3
```

```
[number3,ratio3] = symerr(x,aColumnVector)
```

```
number3 = 1×4

     1     0     2     0


ratio3 = 1×4

    0.3333         0    0.6667         0
```

Use Alternative Type of Comparison

You can specify alternative comparison methods used by `symerr`. In this example, you use a flag 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
```

Input Arguments

x — First input to compare

scalar | vector | matrix

First input to compare, specified as a vector, or a matrix.

Data Types: `double`

y — Second input to compare

scalar | vector | matrix

Second input to compare, specified as a vector, or a matrix.

Data Types: `double`

`flg` — Element comparison type

`'overall'` | `'column-wise'` | `'row-wise'`

Optional argument to override the defaults that govern which elements `symerr` compares and how `symerr` computes the outputs.

- `'overall'` -- `x` and `y` are compared element by element.

- `'column-wise'` -- m^{th} row of `x` vs. m^{th} row of `y`.
- `'row-wise'` -- m^{th} column of `x` vs. m^{th} column of `y`.

See “Specifying Element Comparison” on page 2-796 for more information

Output Arguments

`number` — Number of differing elements

scalar | vector

Number of elements that differ between `x` and `y`, returned as a scalar or vector. The size of `number` is determined by the optional input *`flg`* and by the dimensions of `x` and `y`. See “Default Element Comparison” on page 2-795 and “Specifying Element Comparison” on page 2-796 for more information.

`ratio` — Ratio of differing elements

scalar

The ratio of the number of differing elements, `number`, and the total number of elements of the *smaller* input, returned as a scalar.

`loc` — Location of errors

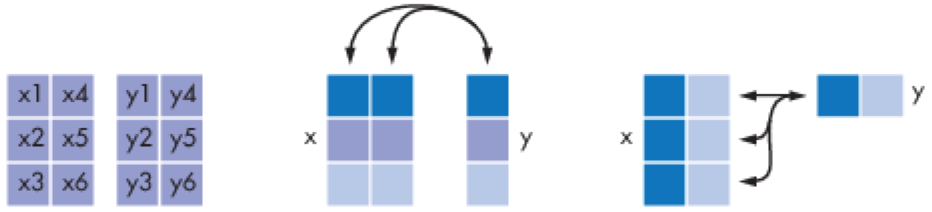
scalar | vector | matrix

Logical array of same size and dimensions as `x`, and `y`, with corresponding locations of errors between inputs `x` and `y`. An element of `loc` is zero if the corresponding comparison yields no discrepancy, and one otherwise.

More About

Default Element Comparison

The `symerr` function compares binary representations of elements in `x` with those in `y`. When optional argument *`flg`* is not specified, `symerr` uses the shape of the inputs `x` and `y` to determine the element comparison method. 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

- 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.

Specifying Element Comparison

Use *flg* to 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-dim. matrix	`'overall'` (default)	Element by element	Total number of symbol errors
	`'column-wise'`	m^{th} column of `x` vs. m^{th} column of `y`	Row vector whose entries count symbol errors in each column
	`'row-wise'`	m^{th} row of `x` vs. m^{th} 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`

See Also

`alignsignals` | `biterr` | `finddelay`

Introduced before R2006a

syndtable

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”.

References

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

See Also

`decode` | `gfcosets` | `hammgen`

Topics

“Block Codes”

Introduced before R2006a

testconsole.Results

(To be removed) Gets results from test console simulations

Compatibility

testconsole.Results will be removed in a future release. Use `comm.ErrorRate` or `bertool` instead. For more information, see “Compatibility Considerations” on page 2-812.

Description

The getResults method of the Error Rate Test Console returns an instance of a testconsole.Results object containing simulation results data. You use methods of the results object to retrieve and plot simulations results data.

Properties

A testconsole.Results object has the properties shown in the following table. All properties are writable except for the ones explicitly noted otherwise.

Property	Description
TestConsoleName	Error Rate Test Console. This property is not writable.
System Under Test Name	Name of the system under test for which the Error Rate Test Console obtained results. This property is not writable.
IterationMode	Iteration mode the Error Rate Test Console used for obtaining results. This property is not writable.
TestPoint	Specify the name of the registered test point for which the results object parses results. The getData, plot, and semilogy methods of the Results object return data or create a plot for the test point that the TestPoint property specifies.
Metric	Specify the name of the test metric for which the results object parses results. The getData, plot, and semilogy methods of the Results object returns data or creates a plot for the metric that the Metric property specifies.
TestParameter1	Specifies the name of the first independent variable for which the results object parses results.
TestParameter2	Specifies the name of the second independent variable for which the results object parses results.

Methods

A testconsole.Results object has the following methods.

getData

`d = getData(r)` returns results data matrix, *d*, available in the results object *r*. The returned results correspond to the test point currently specified in the `TestPoint` property of *r*, and to the test metric currently specified in the `Metric` property of *r*.

If `IterationMode` is 'Combinatorial' then *d* is a matrix containing results for all the sweep values available in the test parameters specified in the `TestParameter1` and `TestParameter2` properties. The rows of the matrix correspond to results for all the sweep values available in `TestParameter1`. The columns of the matrix correspond to results for all sweep values available in `TestParameter2`. If more than two test parameters are registered to the Error Rate Test Console, *d* contains results corresponding to the first value in the sweep vector of all parameters that are not `TestParameter1` or `TestParameter2`.

If `IterationMode` is 'Indexed', then *d* is a vector of results corresponding to each indexed combination of all the test parameter values registered to the Error Rate Test Console.

plot

`plot(r)` creates a plot for the results available in the results object r. The plot corresponds to the test point and test metric, specified by the `TestPoint` and `Metric` properties of *r*

If `IterationMode` is 'Combinatorial' then the plot contains a set of curves. The sweep values in TestParameter1 control the x-axis and the number of sweep values for TestParameter2 specifies how many curves the plot contains. If more than two test parameters are registered to the Error Rate Test Console, the curves correspond to results obtained with the first value in the sweep vector of all parameters that are not TestParameter1, or TestParameter2.

No plots are available when 'IterationMode' is 'Indexed'.

semilogy

`semilogy(...)` is the same as `plot(...)`, except that the Y-Axis uses a logarithmic (base 10) scale.

surf

surf(r) creates a 3-D, color, surface plot for the results available in the `results` object, r. The surface plot corresponds to following items:

- The test point you specify using the `TestPoint` property of the `results` object
- The test metric currently you specify in the `Metric` property of the `results` object

You can specify parameter/value pairs for the `results` object, which establishes additional properties of the surface plot.

When you select 'Combinatorial' for the IterationMode, the sweep values available in the test parameter you specify for the TestParameter1 property control the x-axis of the surface plot. The sweep values available in the test parameter you specify for the TestParameter2 property control the y-axis.

If more than two test parameters are registered to the test console, the surface plot corresponds to the results obtained with the parameter sweep values previously specified with the setParsingValues method of the results object.

You display the current parsing values by calling the getParsingValues method of the results object. The parsing values default to the first value in the sweep vector of each test parameter. By default, the surf method ignores the parsing values for any parameters currently set as TestParameter1 or TestParameter2.

No surface plots are available if the IterationMode is 'Indexed', when less than two registered test parameters exist, or TestParameter2 is set to 'None'.

setParsingValues

`setParsingValues(R,'ParameterName1', 'Value1', ... 'ParameterName2', 'Value2', ...)` sets the parsing values to the values you specify using the parameter-value pairs. Parameter name inputs must correspond to names of registered test parameters, and value inputs must correspond to a valid test parameter sweep value.

You use this method for specifying single sweep values for test parameters that differ from the values for TestParameter1 and TestParameter2. When you define this method, the `results` object returns the data values or plots corresponding to the sweep values you set for the setParsingValues method. The parsing values default to the first value in the sweep vector of each test parameter.

You display the current parsing values by calling the `getParsingValues` method of the results object. You may set parsing values for parameters in TestParameter1 and TestParameter2, but the results object ignores the values when getting data or returning plots.

Parsing values are irrelevant when `IterationMode` is 'Indexed'.

getParsingValues

`getParsingValues` displays the current parsing values for the Error Rate Test Console.

`s = getParsingValues(r)` returns a structure, s, with field names equal to the registered test parameter names and with values corresponding to the current parsing values.

Parsing values are irrelevant when IterationMode is 'Indexed'.

Examples

Error Rate Simulation Sweeps

The `commtest.ErrorRate` and `testconsole.Results` object packages will be removed in a future release. They can be used to perform parameter sweeps to analyze communication system performance. This example demonstrates a workflow that uses them and along with recommended alternate workflows.

Multiple Parameter Sweep and Parallel Run using commtest.ErrorRate

Obtain bit error rate and symbol error rate of an M-PSK system for different modulation orders and EbNo values. System under test is `commtest.MPSKSystem`.

```
% Create an M-ary PSK system
systemUnderTest = commtest.MPSKSystem;

% Instantiate an Error Rate Test Console and attach the system
errorRateTester = commtest.ErrorRate(systemUnderTest);
```

```
Warning: commtest.ErrorRate will be removed in the future. Use comm.ErrorRate or bertool instead
```

```
errorRateTester.SimulationLimitOption = 'Number of errors or transmissions';
errorRateTester.MaxNumTransmissions = 1e5;

% Set sweep values for simulation test parameters
setTestParameterSweepValues(errorRateTester,'M',2.^[1 2 3 4],'EbNo',(-5:10))

% Register a test point
registerTestPoint(errorRateTester,'MPSK_BER','TxInputBits','RxOutputBits')

% Get information about the simulation settings
info(errorRateTester)
```

```
Test console name:           commtest.ErrorRate
System under test name:      commtest.MPSKSystem
Available test inputs:       NumTransmissions, RandomIntegerSource
Registered test inputs:      NumTransmissions
Registered test parameters:  EbNo, M
Registered test probes:      RxOutputBits, RxOutputSymbols, TxInputBits, TxInputSymbols
Registered test points:      MPSK_BER
Metric calculator functions: @commtest.ErrorRate.defaultErrorCalculator
Test metrics:                ErrorCount, TransmissionCount, ErrorRate
```

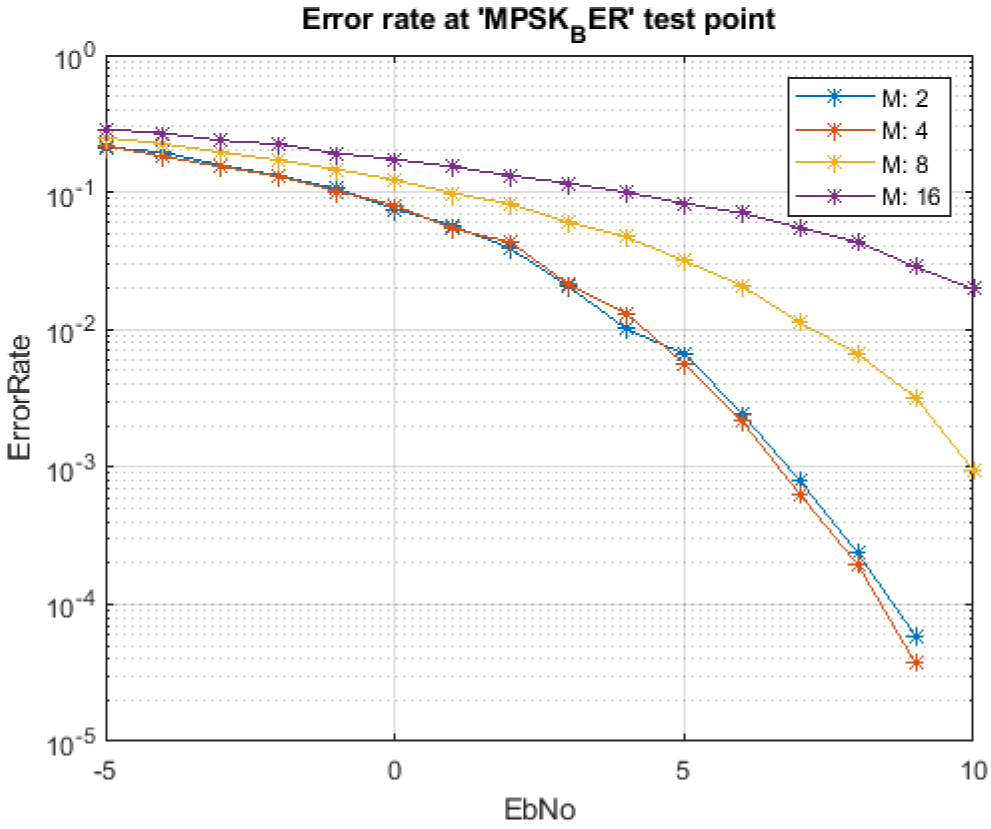
```
% Run the M-PSK simulations
run(errorRateTester)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 12).
12 workers available for parallel computing. Simulations will be distributed among these workers
Running simulations...
```

```
% Get the results
mpskResults = getResults(errorRateTester);
```

```
Warning: testconsole.Results will be removed in the future. See <a href="matlab:helpview(fullfile
```

```
% Get a semi-log scale plot of EbNo versus bit error rate for
% different values of modulation order M
mpskResults.TestParameter2 = 'M';
semilogy(mpskResults,'*-')
```



Multiple Parameter Sweep and Parallel Run using nested for loops and comm.ErrorRate

Run an error rate simulation over M=2.^(1:4) and EbNo=-5:10. Use `comm.ErrorRate` to collect both bit error rate (BER) and symbol error rate (SER) data. Run the simulations to collect a minimum of 100 symbol errors or for a maximum of 1e5 symbols.

```
% Set the M sweep values same as the commtest.ErrorRate object
getTestParameterSweepValues(errorRateTester,'M')
```

```
ans = 1×4

     2     4     8    16
```

```
MSweep = 2.^[1 2 3 4];

% Set EbNo sweep values same as the commtest.ErrorRate object
getTestParameterSweepValues(errorRateTester,'EbNo')
```

```
ans = 1×16

    -5    -4    -3    -2    -1     0     1     2     3     4     5     6     7     8     9    10
```

```
EbNoSweep = -5:10;

% Set minumum number of errors same as the commtest.ErrorRate object
errorRateTester.MinNumErrors
```

```
ans = 100
```

```
minNumErrors = 100;

% Set maximum number of transmissions same as the commtest.ErrorRate
% object. In this example a transmission is a symbol.
errorRateTester.MaxNumTransmissions
```

```
ans = 100000
```

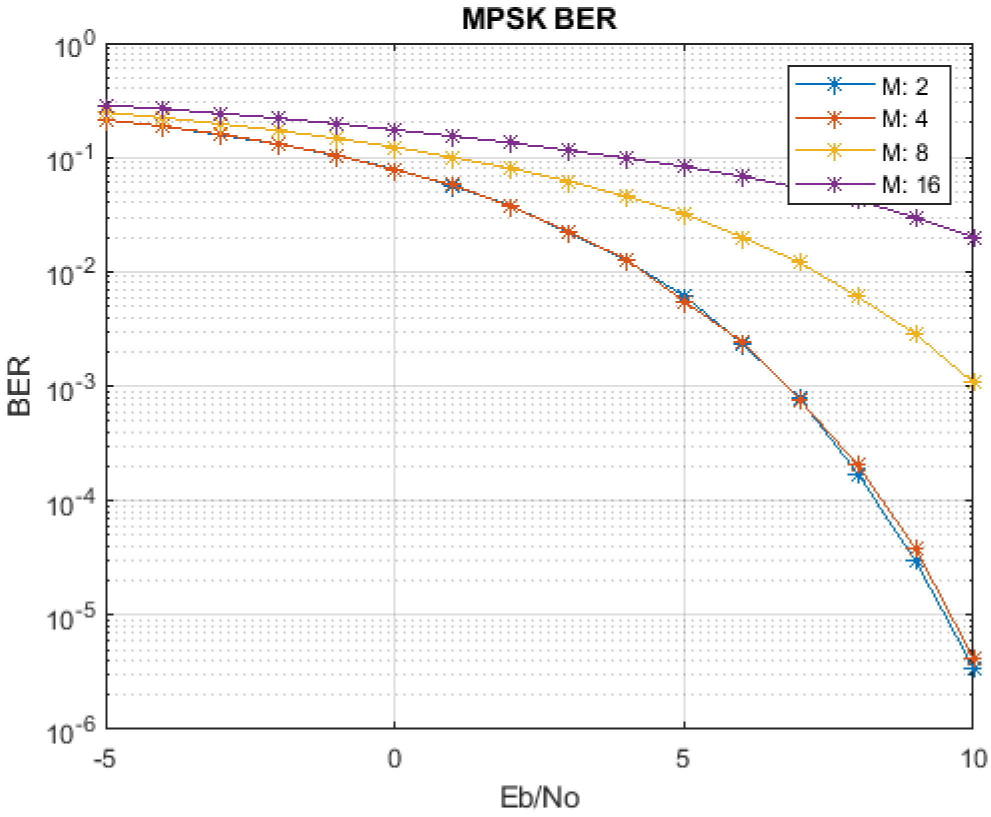
```
MaxNumTransmissions = 1e5;

% Set frame length same as the commtest.ErrorRate object
errorRateTester.FrameLength
```

```
ans = 500
```

```
frameLength = 500;

% Find out if there is a parallel pool and how many workers are available
[licensePCT,~] = license('checkout','distrib_computing_toolbox');
if (licensePCT && ~isempty(ver('parallel')))
    p = gcp;
    if isempty(p)
        numWorkers = 1;
    else
        numWorkers = p.NumWorkers
    end
else
    numWorkers = 1;
end
```

```
numWorkers = 12
```

```
minNumErrorsPerWorker = minNumErrors/numWorkers;
maxNumSymbolsPerWorker = MaxNumTransmissions/numWorkers;

% Store results in an array, where first dimension is M and second
% dimension is EbNo. Initialize the vector with NaN values.
ser = nan(length(MSweep),length(EbNoSweep));
ber = nan(length(MSweep),length(EbNoSweep));

% First sweep is over M (modulation order)
for MIdx = 1:length(MSweep)
    M = MSweep(MIdx);
    bitsPerSymbol = log2(M);

    % Second sweep is over EbNo
    for EbNoIdx = 1:length(EbNoSweep)
        EbNo = EbNoSweep(EbNoIdx);

        SNR = EbNo+10*log10(bitsPerSymbol);

        numSymbolErrors = zeros(numWorkers,1);
        numBitErrors = zeros(numWorkers,1);
        numSymbols = zeros(numWorkers,1);

        parfor worker = 1:numWorkers
            symErrRate = comm.ErrorRate;
            bitErrRate = comm.ErrorRate;
            while (numSymbolErrors(worker) < minNumErrorsPerWorker) ...
                    || (numSymbols(worker) < maxNumSymbolsPerWorker)
                % Generate frameLength source outputs
                txMsg = randi([0 M-1],frameLength,1);

                % Modulate the data
                txOutput = pskmod(txMsg,M,0,'gray');
                % Pass data through an AWGN channel with current SNR value
                chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
                % Demodulate the data
                rxOutput = pskdemod(chnlOutput,M,0,'gray');

                % Calculate number of symbol errors
                symErrVal = symErrRate(txMsg,rxOutput);
                numSymbolErrors(worker) = symErrVal(2);
                numSymbols(worker) = symErrVal(3);

                % Convert symbol streams to bit streams
                bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
                bTx = bTx(:);
                bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
                bRx = bRx(:);

                % Calculate number of bit errors
                bitErrVal = bitErrRate(bTx,bRx);
                numBitErrors(worker) = bitErrVal(2);
            end
        end

        ber(MIdx,EbNoIdx) = sum(numBitErrors)/(sum(numSymbols)*bitsPerSymbol);
        ser(MIdx,EbNoIdx) = sum(numSymbolErrors)/sum(numSymbols);
    end
end

% Plot results
semilogy(EbNoSweep,ber,'*-')
grid on
title('MPSK BER')
xlabel('Eb/No')
ylabel('BER')
legendText = cell(length(MSweep),1);
for p=1:length(MSweep)
    legendText{p} = sprintf('M: %d',MSweep(p));
end
legend(legendText)
```



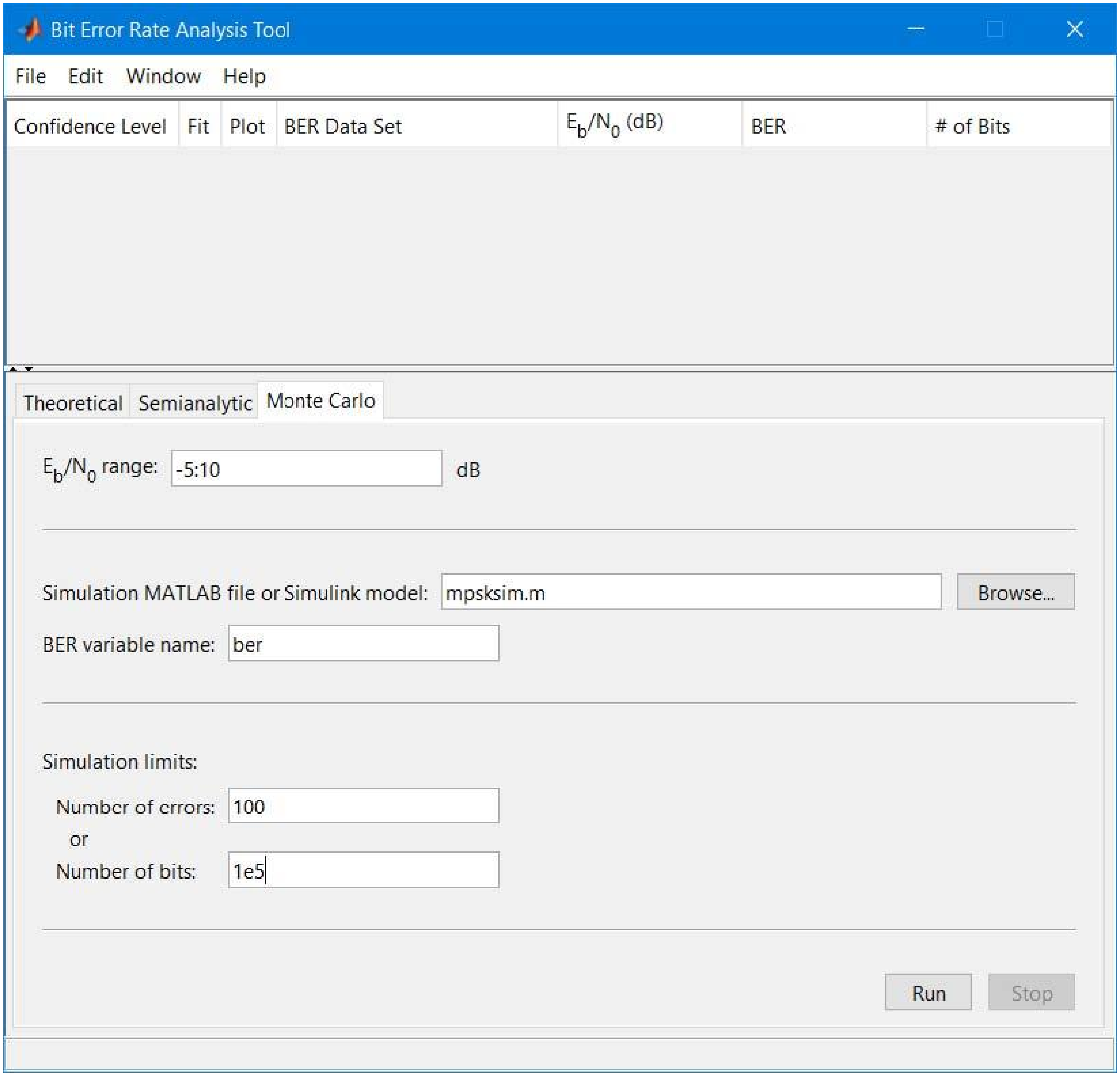
Multiple Variable Sweeps using BERTool

`BERTool` computes the BER as a function of signal-to-noise ratio. It analyzes performance either with Monte-Carlo simulations of MATLAB® functions and Simulink® models or with theoretical closed-form expressions for selected types of communication systems. The `bertool` function opens the `BERTool`. Here `BERTool` is configured to call the simulation defined in the function `mpsksim` included below.

```
function [ber,numBits] = mpsksim(EbNo,minNumErrs,maxNumBits)
% Import the Java class for BERTool, so that you will be able to stop the simulation using the "S
import com.mathworks.toolbox.comm.BERTool;

frameLength = 500;

M = 16; % Can be 2, 4, 8, 16
bitsPerSymbol = log2(M);

maxNumSymbols = maxNumBits/bitsPerSymbol;

SNR = EbNo + 10*log10(bitsPerSymbol);

% Initialize variables related to exit criteria.
numBitErrors = 0;
numSymbols = 0;

while (numBitErrors < minNumErrs) || (numSymbols < maxNumSymbols)
```

```
  % Check if the user clicked the Stop button of BERTool.
  if (BERTool.getSimulationStop)
    break;
  end

  % Generate frameLength source outputs
  txMsg = randi([0 M-1],frameLength,1);
  numSymbols = numSymbols+frameLength;

  % Modulate the data
  txOutput = pskmod(txMsg,M,0,'gray');
  % Pass data through an AWGN channel with current SNR value
  chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
  % Demodulate the data
  rxOutput = pskdemod(chnlOutput,M,0,'gray');

  % Convert symbol streams to bit streams
  bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
  bTx = bTx(:);
  bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
  bRx = bRx(:);

  % Calculate number of bit errors
  numBitErrors = numBitErrors+sum(bTx~=bRx);
end

% Assign values to the output variables.
numBits = numSymbols*bitsPerSymbol;
ber = numBitErrors/numBits;
```

Configure `BERTool` as follows.



Bit Error Rate Analysis Tool

E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

BER variable name: ber

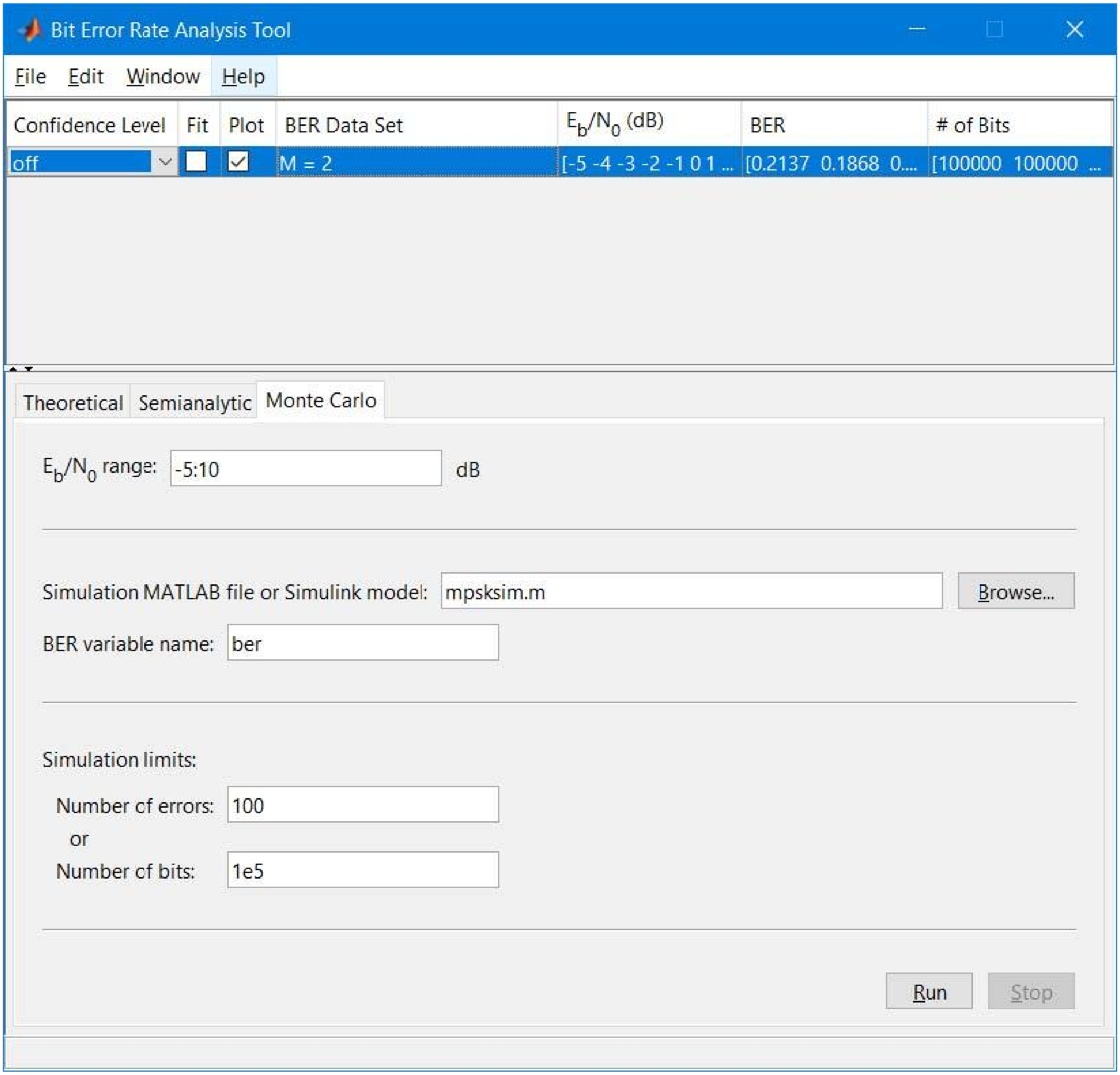
Simulation limits:

Number of errors: 100

or

Number of bits: 1e5

Set M=2 in the `mpsksim` function and click Run. Set the **BER Data Set** name to `'M=2'`.



Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	□	☑	M = 2	[-5 -4 -3 -2 -1 0 1 ...	[0.2137 0.1868 0....	[100000 100000 ...

E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

BER variable name: ber

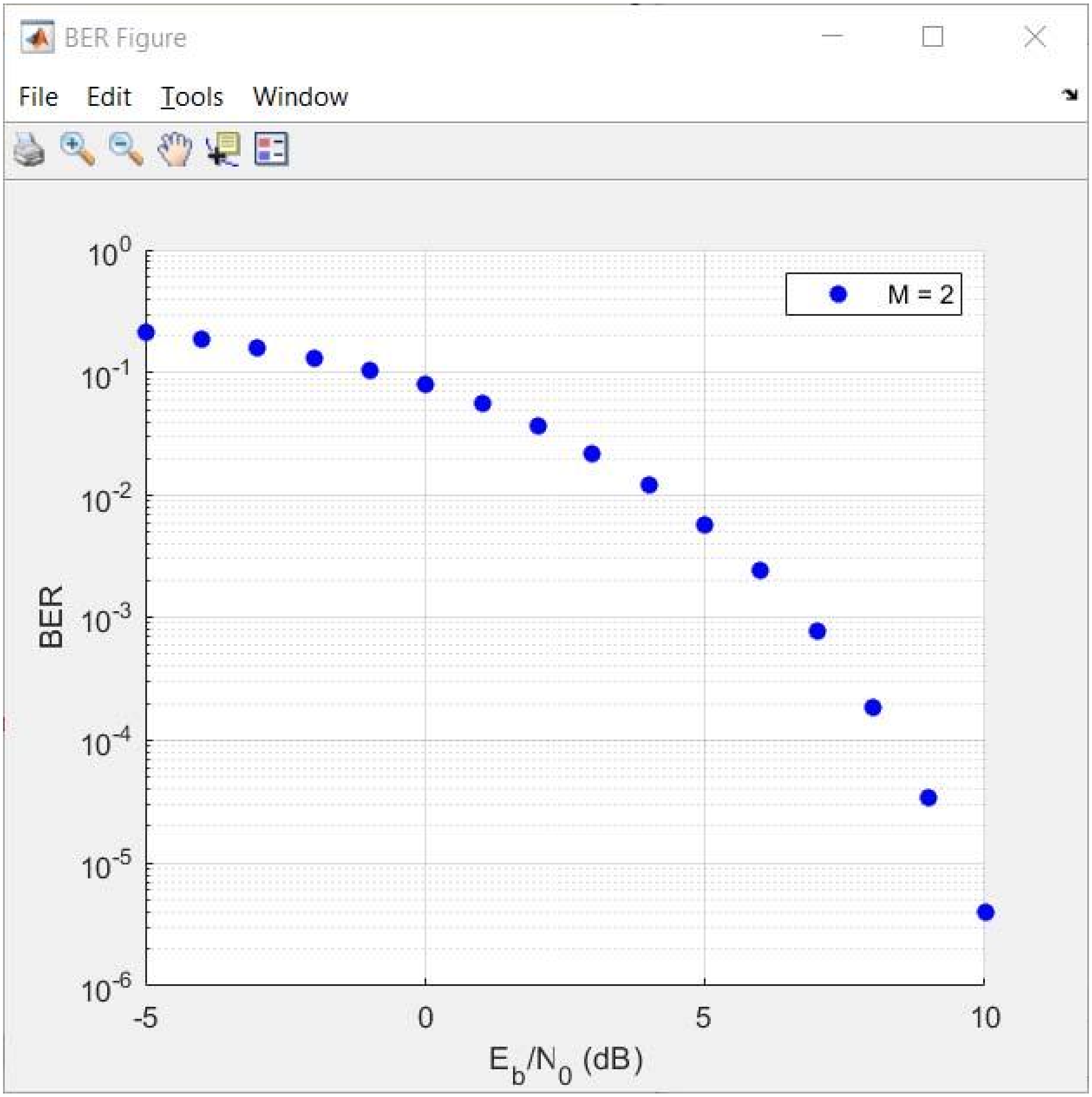
Simulation limits:

Number of errors: 100

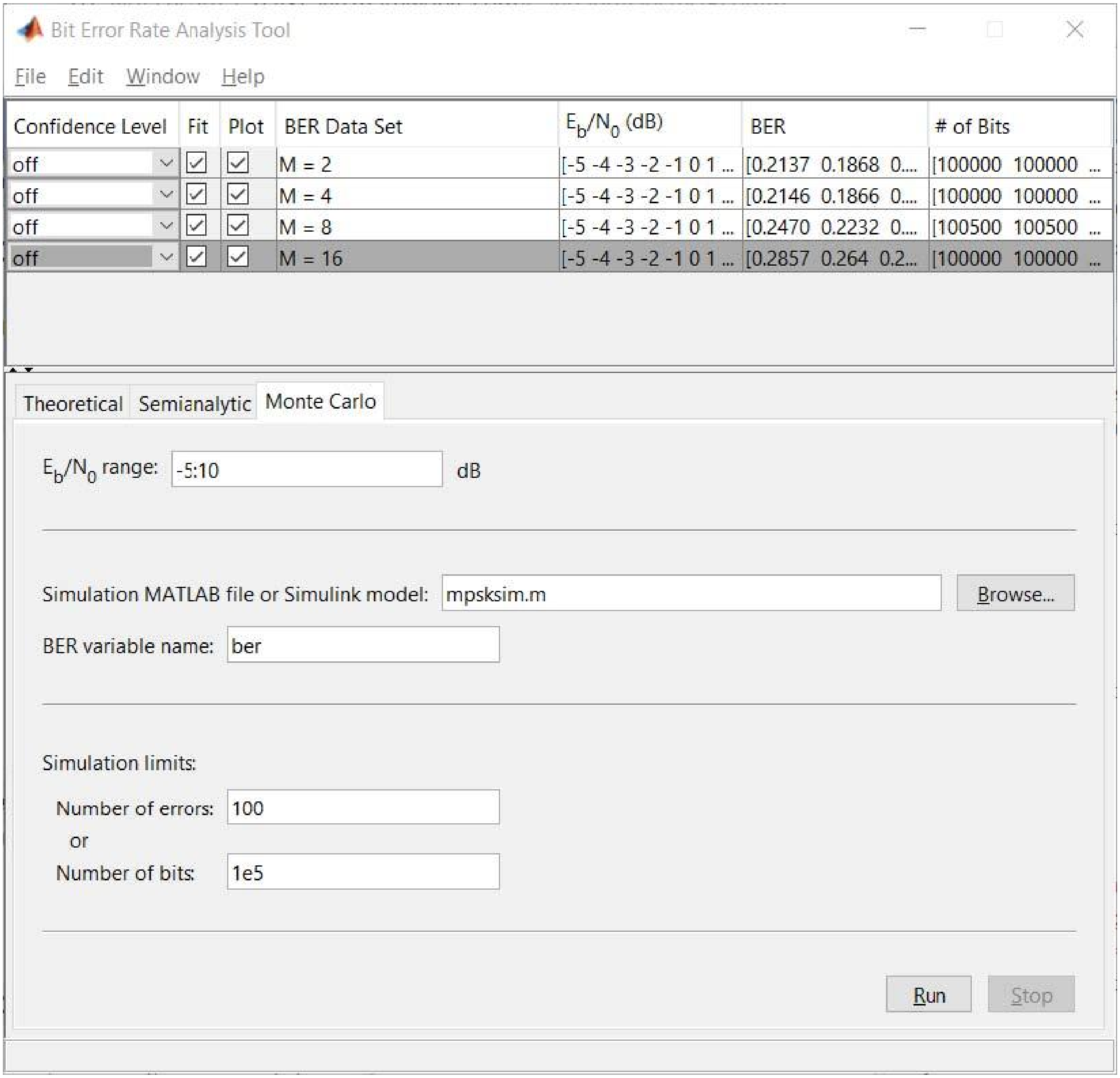
or

Number of bits: 1e5

Display the BER curve for M=2.



Update the value for `M` in the `mpsksim` function, repeating this process for `M = 4`, `8`, `16`. You will see results similar to those below in the **Bit Error Rate Analysis Tool** window and the **BER** figure.



Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
off	☑	☑	M = 2	[-5 -4 -3 -2 -1 0 1 ...	[0.2137 0.1868 0....	[100000 100000 ...
off	☑	☑	M = 4	[-5 -4 -3 -2 -1 0 1 ...	[0.2146 0.1866 0....	[100000 100000 ...
off	☑	☑	M = 8	[-5 -4 -3 -2 -1 0 1 ...	[0.2470 0.2232 0....	[100500 100500 ...
off	☑	☑	M = 16	[-5 -4 -3 -2 -1 0 1 ...	[0.2857 0.264 0.2...	[100000 100000 ...

E_b/N_0 range: -5:10 dB

Simulation MATLAB file or Simulink model: mpsksim.m

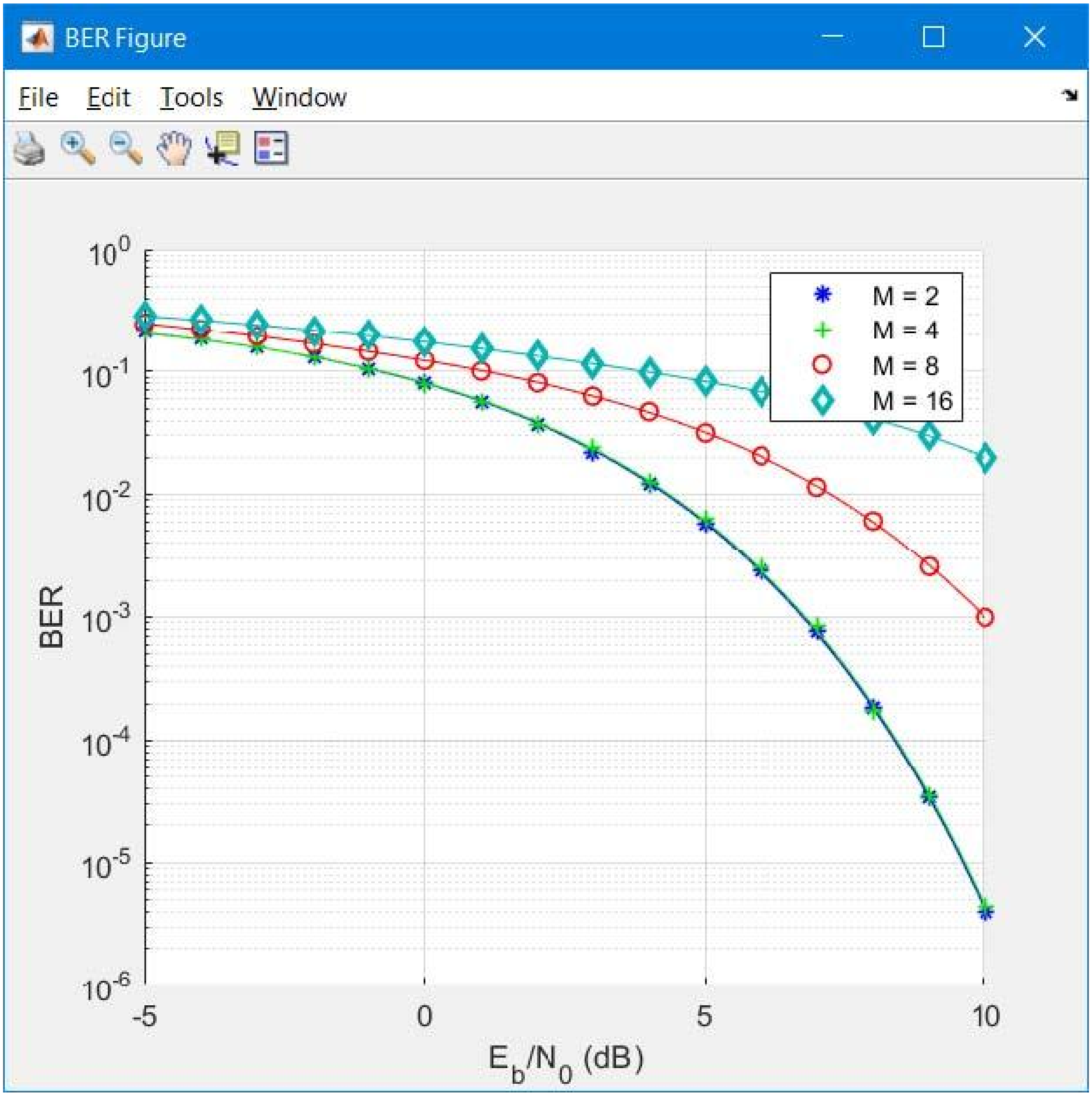
BER variable name: ber

Simulation limits:

Number of errors: 100

or

Number of bits: 1e5



Parallel SNR Sweep using BERTool

Using `parfor`, run each simulation point in parallel by configuring your simulation function similar to the `mpsksim_parfor` function included below. Since `parfor` cannot work with the Java class for `BERTool`, you will not be able to stop the simulation using the **Stop** button.

```
function [ber,numBits] = mpsksim_parfor(EbNo,minNumErrs,maxNumBits)

% Find out if there is a parallel pool and how many workers are available
if license('test','Distrib_Computing_Toolbox')
  p = gcp;
  if isempty(p)
    numWorkers = 1;
  else
    numWorkers = p.NumWorkers;
  end
else
  numWorkers = 1;
end

M = 2;
bitsPerSymbol = log2(M);

maxNumSymbols = maxNumBits/bitsPerSymbol;

minNumErrorsPerWorker = minNumErrs/numWorkers;
maxNumSymbolsPerWorker = maxNumSymbols/numWorkers;
frameLength = 500;

SNR = EbNo + 10*log10(bitsPerSymbol);

% Initialize variables related to exit criteria.
numBitErrors = zeros(numWorkers,1);
numSymbols = zeros(numWorkers,1);

parfor worker = 1:numWorkers
  while (numBitErrors(worker) < minNumErrorsPerWorker) ...
      || (numSymbols(worker) < maxNumSymbolsPerWorker)

    % Generate frameLength source outputs
    txMsg = randi([0 M-1],frameLength,1);
    numSymbols(worker) = numSymbols(worker)+frameLength;

    % Modulate the data
    txOutput = pskmod(txMsg, M, 0, 'gray');
    % Pass data through an AWGN channel with current SNR value
    chnlOutput  = awgn(txOutput,SNR,'measured',[],'dB');
    % Demodulate the data
    rxOutput = pskdemod(chnlOutput,M,0,'gray');

    % Convert symbol streams to bit streams
    bTx = de2bi(txMsg,bitsPerSymbol,'left-msb')';
    bTx = bTx(:);
    bRx = de2bi(rxOutput,bitsPerSymbol,'left-msb')';
    bRx = bRx(:);

    % Calculate number of bit errors
    numBitErrors(worker) = numBitErrors(worker)+sum(bTx~=bRx);
  end
end

% Assign values to the output variables.
ber = sum(numBitErrors)/sum(numSymbols);
numBits = sum(numSymbols)*bitsPerSymbol;
```

Compatibility Considerations

`testconsole.Results` will be removed

Warns starting in R2019b

testconsole.Results will be removed in a future release. Use `comm.ErrorRate` or `bertool` instead. The “Error Rate Simulation Sweeps” on page 2-800 example demonstrates alternate workflows using `comm.ErrorRate` and `bertool`.

See Also

Objects
`comm.ErrorRate`

Functions
`bertool`

Topics
“Bit Error Rate (BER)”

Introduced in R2009b

tpcdec

Turbo product code (TPC) decoder

Syntax

```
decoded = tpcdec(llr,N,K)
decoded = tpcdec(llr,N,K,S)
decoded = tpcdec(llr,N,K,S,maxnumiter)
decoded = tpcdec(llr,N,K,S,maxnumiter,earlyterm)
[decoded,actualnumiter] = tpcdec(___)
```

Description

`decoded = tpcdec(llr,N,K)` performs 2-D TPC decoding on input log likelihood ratios, `llr`, using two linear block codes specified by codeword length `N` and message length `K`. For a description of 2-D TPC decoding, see “Turbo Product Code Decoding” on page 2-819.

`decoded = tpcdec(llr,N,K,S)` performs 2-D TPC decoding on the shortened `llr` using a 2-D TPC decoder specified by codeword length (`N`–`K`+`S`) and message length `S`.

`decoded = tpcdec(llr,N,K,S,maxnumiter)` performs 2-D TPC decoding for `maxnumiter` iterations. To use `maxnumiter` with full-length messages, specify `S` as empty, [ ].

`decoded = tpcdec(llr,N,K,S,maxnumiter,earlyterm)` performs 2-D TPC decoding and terminates early if the calculated syndrome or parity-check of the component code evaluates to zero before `maxnumiter` decoding iterations. To use `maxnumiter` and `earlyterm` with full-length messages, specify `S` as empty, [ ].

`[decoded,actualnumiter] = tpcdec(___)`also returns the actual number of decoding iterations after performing 2-D TPC decoding using any of the prior syntaxes.

Examples

Decode Using Full-Length TPC Codes

Decode an approximate log-likelihood ratio output signal from 16-QAM demodulation.

Begin by encoding a random bit vector using 2-D turbo product coding (TPC) with extended Hamming codes and extended BCH codes.

Specify the (N,K) code pairs to use for TPC encoding.

```
N = [32;16];
K = [21;11];
```

Generate a column vector of random message bits and TPC-encode the message. Specify the message bits as a vector with length equal to the product of the elements in `K`.

```
msg = randi([0 1],prod(K),1);
code = tpcenc(msg,N,K);
```

Apply 16-QAM modulation. Add AWGN to the signal. Demodulate the signal, outputting approximate LLRs.

```
M = 16;
snr = 10;

txsig = qammod(code,M,'InputType','bit', ...
    'UnitAveragePower',true);

rxsig = awgn(txsig,snr,'measured');

llr = qamdemod(rxsig,M,'OutputType','approxllr', ...
    'UnitAveragePower',true,'NoiseVariance',10.^(-snr/10));
```

Perform TPC decoding using three iterations. Because the demodulator output is negative bipolar mapped and TPC decoder expects positive bipolar mapped input, the demodulated signal output must be negated at the decoder input. Check the number of bit errors in the decoded signal.

```
iterations = 3;
decoded = tpcdec(-llr,N,K,[],iterations);

numerr = biterr(msg,decoded)
```

```
numerr = 0
```

Decode Using Shortened TPC Codes

Decode a shortened TPC code. Apply QPSK modulation and output the approximate log-likelihood ratio signal obtained from QPSK demodulation.

Begin by encoding a random bit vector using 2-D turbo product coding (TPC) with extended Hamming codes and extended BCH codes.

Specify (N,K) code pairs and S for TPC encoding.

```
N = [32;32];
K = [21;26];
S = [19;24];
```

Generate a column vector of random message bits and TPC-encode the message. Specify the shortened message bits as a vector with length equal to the product of the elements in S.

```
msg = randi([0 1],prod(S),1);
code = tpcenc(msg,N,K,S);
```

Apply QPSK modulation. Add AWGN to the signal. Demodulate the signal and output approximate LLRs.

```
M = 4;
snr = 3;

txsig = qammod(code,M,'InputType','bit', ...
    'UnitAveragePower',true);

rxsig = awgn(txsig,snr,'measured');
```

```
llr = qamdemod(rxsig,M,'OutputType','approxllr', ...
    'UnitAveragePower',true,'NoiseVariance',10.^(-snr/10));
```

Perform TPC decoding using two iterations. Because the demodulator output is negative bipolar mapped and TPC decoder expects positive bipolar mapped input, the demodulated signal output must be negated at the decoder input. Check the bit error rate of the decoded signal.

```
iterations = 2;
decoded = tpcdec(-llr,N,K,S,iterations);

[~,ber] = biterr(msg,decoded)
```

```
ber = 0.0066
```

TPC Decoding with Shortening and Early Termination

Decode a shortened TPC code and specify early termination of decoding. Apply QPSK modulation and output the approximate log-likelihood ratio signal obtained from QPSK demodulation.

Begin by encoding a random bit vector using 2-D turbo product coding (TPC) with extended Hamming codes and extended BCH codes. Specify (N,K) code pairs and S for TPC encoding, and a maximum of 10 decoding iterations. Perform QPSK modulation on the signal.

```
n = [64; 32];
k = [51; 26];
s = [49; 24];
maxnumiter = 10;
M = 4;

msg = randi([0 1],prod(s),1);  % Random bits
code = tpcenc(msg,n,k,s);

txsig = qammod(code,M,'InputType','bit', ...
    'UnitAveragePower',true);
```

Add noise to the transmitted signal.

```
snr = 5;
rxsig = awgn(txsig,snr,'measured');
```

Demodulate the received signal using approximate LLR demapping.

```
llr = qamdemod(rxsig,M,'OutputType', ...
    'approxllr','UnitAveragePower',true, ...
    'NoiseVariance',10.^(-snr/10));
```

Specify the maximum number of TPC decoding iterations and return the actual number of iterations performed. Early termination of the TPC decoding is on by default. Display the number of errors and the number of iterations performed.

```
[decoded,actualNumIter] = tpcdec(-llr,n,k,s,maxnumiter);
numErr = biterr(msg,decoded);
disp(['Terminated after ' num2str(actualNumIter) ' iterations.' ...
    ' Number of errors = ' num2str(numErr) '.']);
```

```
Terminated after 4 iterations. Number of errors = 0.
```

Input Arguments

`llr` — Log likelihood ratios

column vector

Log likelihood ratios, specified as a column vector.

- For full-length codes, the length of the input column vector is the product of the elements in `N`.
- For shortened codes, the length of the input column vector is the product of the elements in (`N`–`K`+`S`).

Data Types: `double` | `single`

`N` — Codeword length

two-element integer vector

Codeword length, specified as a two-element integer vector, [N_R; N_C]. N_R represents the number of rows in the product code matrix. N_C represents the number of columns in the product code matrix. For more information about N_R and N_C, see “Turbo Product Code Decoding” on page 2-819. For a list of valid ($N(i)$, $K(i)$) code pairs, see “More About” on page 2-818.

Data Types: `double`

`K` — Message length

two-element integer vector

Message length, specified as a two-element integer vector, [K_R; K_C]. For a full-length message, the input column vector containing the input LLRs is arranged into a K_R-by-K_C matrix. K_R represents the number of rows in the message matrix. K_C represents the number of columns in the message matrix. For more information about K_R and K_C, see “Turbo Product Code Decoding” on page 2-819. For a list of valid ($N(i)$, $K(i)$) code pairs, see “More About” on page 2-818.

Data Types: `double`

`S` — Shortened message length

two-element integer vector

Shortened message length, specified as a two-element integer vector, [S_R; S_C]. For a shortened message, the input column vector containing the input LLRs is arranged into an S_R-by-S_C matrix. S_R represents the number of rows in the matrix. S_C represents the number of columns in the matrix. For more information about S_R and S_C, see “Turbo Product Code Decoding” on page 2-819.

When you specify this parameter, specify N and K vectors for the full-length TPC codes that are shortened to ($N(i)$ – $K(i)$ + $S(i)$, $S(i)$) codes.

Data Types: `double`

`maxnumiter` — Maximum number of decoding iterations

4 (default) | positive integer

Maximum number of decoding iterations, specified as a positive integer.

Data Types: `double`

`earlyterm` — Enable early termination

`true` (default) | `false`

Enable early termination of decoding, specified as a logical. When `earlyterm` is `true` the decoding terminates early if the calculated syndrome or parity-check of the component code evaluates to zero before `maxnumiter` decoding iterations.

Data Types: `double`

Output Arguments

decoded — TPC decoded message

column vector

TPC decoded message, returned as a column vector.

- For full-length codes, the length of the returned column vector is the product of the elements in `K`.
- For shortened codes, the length of the returned column vector is the product of the elements in `S`.

Data Types: `logical`

`actualnumiter` — Actual number of decoding iterations

positive integer

Actual number of decoding iterations performed, returned as a positive integer.

Data Types: `double`

More About

Component Codes

This table lists the supported component code pairs for the row (N_R,K_R) and column (N_C,K_C) parameters.

- N_R and K_R represent the number of rows in the product code matrix and message matrix, respectively.
- N_C and K_C represent the number of columns in the product code matrix and message matrix, respectively.

Within each code type, any two component code pairs can form a 2-D TPC code. The table also includes the error-correction capability for each code pair.

Code type	**Component Code Pairs**(N_R,K_R) and (N_C,K_C)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1
	(15,11)	1
	(7,4)	1

Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2
	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-

Turbo Product Code Decoding

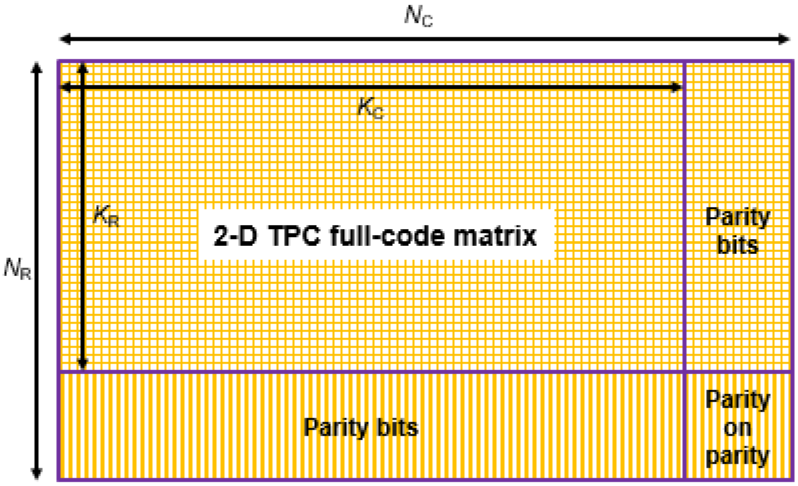
Turbo product codes (TPC) are a form of concatenated codes used as forward error correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. The TPC decoder achieves near-optimum decoding of product codes using Chase decoding and the Pyndiah algorithm to perform iterative soft input, soft output decoding. For a detailed description, see [1] and [2]. This decoder implements an iterative soft input, soft output 2-D product code decoding, as described in [2], using two “Linear Block Codes”. The decoder expects the soft bit log likelihood ratios (LLRs) obtained from digital demodulation as the input signal.

Note The TPC decoder expects a positive bipolar mapped input, specifically –1 mapped to 0 and +1 mapped to 1. The output from demodulators in the Communications Toolbox is negative bipolar mapping, specifically +1 mapped to 0 and –1 mapped to +1. Therefore, the LLR output from demodulators must be negated to provide the positive bipolar mapped input expected by the TPC decoder.

The TPC decoder decodes either full-length or shortened codes.

TPC Decoding Full-Length Messages

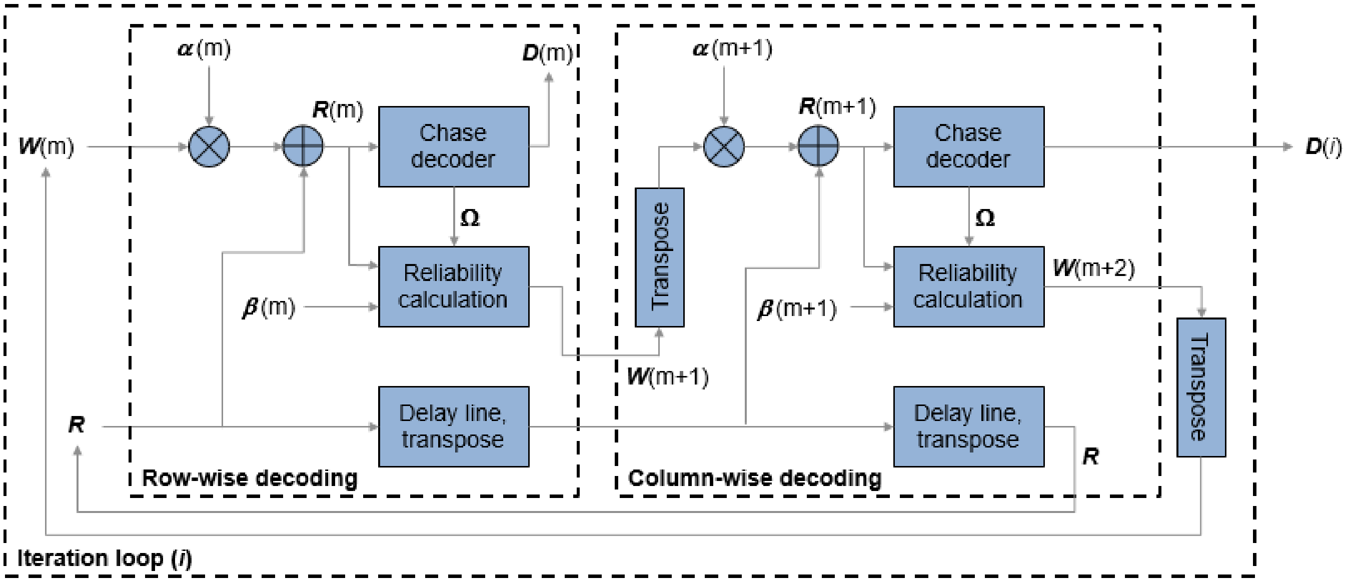
TPC encoded full-length input messages are decoded using specified 2-D TPC code pairs. Row-wise decoding uses the (N_C,K_C) code pair and column-wise decoding uses the (N_R,K_R) code pair. The input vector length must be $N_R \times N_C$. To perform the 2-D TPC decoding, the column vector of the input LLRs, composed of the message and parity bits, is arranged into an N_R-by-N_C matrix.



The TPC decoder achieves near-optimum decoding of product codes using Chase decoding and the Pyndiah algorithm to perform iterative soft input, soft output decoding. Chase decoding forms a set of possible codewords for each row or column. The Pyndiah algorithm calculates soft information required for the next decoding step.

Iterative Soft Input, Soft Output Decoder

The iterative soft input, soft output decoding, as shown in the block diagram, carries out two decoding steps for each iteration.



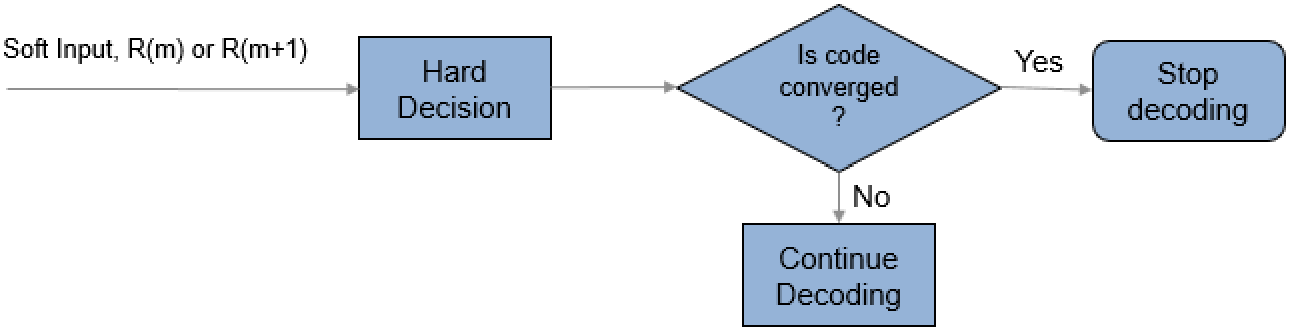
The soft inputs for decoding are $\boldsymbol{R}(m) = \boldsymbol{R} + \alpha(m)\boldsymbol{W}(m)$.

- Iteration loop counter i increments from $i = 1$ to the specified number of iterations.
- $m = 2i - 1$ is the decoding step index.
- $\boldsymbol{R}$ is the received LLR matrix.
- $\boldsymbol{R}(m)$ is the soft input for the mth decoding step.
- $\boldsymbol{W}(m)$ is the input extrinsic information for the mth decoding step.
- $\alpha(m) = [0,0.2,0.3,0.5,0.7,0.9,1,1, \ldots]$, where α is a weighting factor applied based on the decoding step index. For higher decoding steps, $\alpha = 1$.
- $\beta(m) = [0.2,0.4,0.6,0.8,1,1, \ldots]$, where β is a reliability factor applied based on the decoding step index. For higher decoding steps, $\beta = 1$.
- $\boldsymbol{D}$ contains the decoded message bits. The output message bits are formed from $\boldsymbol{D}$ by mapping –1 to 0 and +1 to 1, then reshaping the message block into a column vector.

The output message bits are formed after iterating through the specified number of iterations, or, if early termination is enabled, after code convergence.

Early Termination of TPC Decoding

If early termination is enabled, a code convergence check is performed on the hard decision of the soft input in each row-wise and column-wise decoding step. Early termination can be triggered after either the row-wise decoding or column-wise decoding converges.



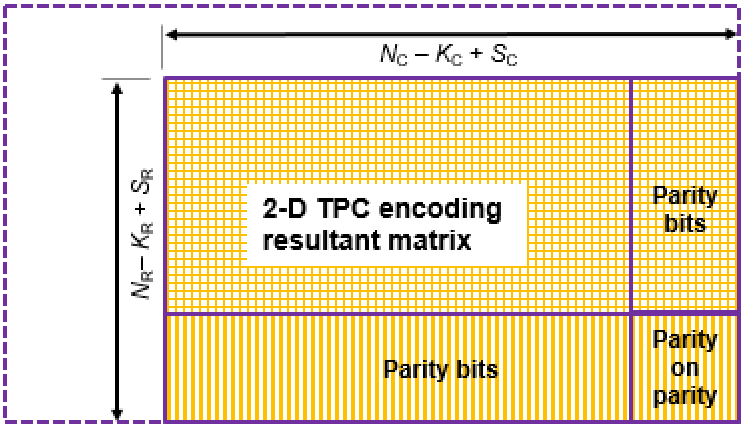
The code is converged if, for all rows or all columns,

- The syndrome evaluates to zero in the codes (Hamming codes, Extended Hamming codes, BCH codes, or Extended BCH codes).
- The parity check is evaluated to zero in parity check codes.

The reported number of iterations evaluates to the iteration value that is currently in progress. For example, if the code convergence check is satisfied after row-wise decoding in the third iteration (after 2.5 decoding steps), then the number of iteration returned is 3.

TPC Decoding Shortened Messages

TPC encoded shortened input messages are decoded using specified 2-D TPC code pairs. Row-wise decoding uses the $(N_C - K_C + S_C, S_C)$ code pair and column-wise decoding uses the $(N_R - K_R + S_R, S_R)$ code pair. The input vector length must be $(N_R - K_R + S_R) \times (N_C- K_C + S_C)$. To perform the 2-D TPC decoding of shortened messages, the column vector of the input LLRs, composed of the shortened message and parity bits, is arranged into an (N_R – K_R + S_R)-by-(N_C – K_C + S_C) matrix.



The TPC decoder processes the received shortened message LLRs similar to full length codes, with these exceptions:

- The shortened bit positions in the received codeword are set to –1.
- The Chase algorithm does not consider the shortened bit positions while choosing the least reliable bits.

References

[1] Chase, D. "Class of Algorithms for Decoding Block Codes with Channel Measurement Information." *IEEE Transactions on Information Theory*, Volume 18, Number 1, January 1972, pp. 170–182.

[2] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Volume 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- TPC parameters `N`, `K`, and `S` must be constant values. If the value used for each of these parameters does not change, then you can assign them by expression or variable.

See Also

Functions
`bchdec` | `tpcenc`

Objects
`comm.BCHDecoder`

Blocks

TPC Decoder

Introduced in R2018a

tpcenc

Turbo product code (TPC) encoder

Syntax

```
code = tpcenc(msg,N,K)
code = tpcenc(msg,N,K,S)
```

Description

`code = tpcenc(msg,N,K)` performs 2-D TPC encoding of the input message, `msg`, using two linear block codes specified by codeword length `N` and message length `K`. For a description of 2-D TPC encoding, see “Turbo Product Code Construction” on page 2-827.

`code = tpcenc(msg,N,K,S)` performs 2-D TPC encoding on the shortened input message of length `S`, using a 2-D TPC encoder specified by codeword length (`N`–`K`+`S`) and message length `S`.

Examples

Encode Using Full-Length TPC Codes

Encode a random bit vector using 2-D turbo product coding (TPC) with extended Hamming codes and extended BCH codes.

Specify (N,K) code pairs for TPC encoding.

```
N = [32;64];
K = [21;57];
```

Generate a column vector of random message bits. The desired length for the message bits is the product of elements in `K`.

```
msg = randi([0 1],prod(K),1);
```

TPC-encode the message.

```
code = tpcenc(msg,N,K);
```

Verify that the length of the encoded codeword is the product of elements in `N`.

```
size(code)
```

```
ans = 1×2

        2048           1
```

```
prod(N)
```

```
ans = 2048
```

Encode Shortened Message Using Turbo Product Coding

Encode a random bit vector using 2-D turbo product coding (TPC), applying message shortening.

Specify (N,K) code pairs and S for TPC encoding.

```
N = [32;64];
K = [21;57];
S = [19;24];
```

Generate a column vector of random message bits. The desired length for the shortened message bits is the product of the elements in S.

```
msg = randi([0 1],prod(S),1);
```

TPC-encode the shortened message.

```
code = tpcenc(msg,N,K,S);
```

Verify that the length of the encoded codeword is the product of elements in (N-K+S).

```
size(code)
```

```
ans = 1×2

   930     1

```

```
prod(N-K+S)
```

```
ans = 930
```

Input Arguments

msg — Input message bits to encode

column vector

Input message bits to encode, specified as a column vector.

- For a full-length input messages, the length of the column vector must be the product of the elements in K.
- For a shortened input messages, the length of the column vector must be the product of the elements in S.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | logical

N — Codeword length

two-element integer vector

Codeword length, specified as a two-element integer vector, [N_R; N_C]. N_R represents the number of rows in the product code matrix. N_C represents the number of columns in the product code matrix. For more information about N_R and N_C, see “Turbo Product Code Construction” on page 2-827. For a list of valid ($N(i)$,$K(i)$) code pairs, see “Component Codes” on page 2-826.

Data Types: `double`

K — Message length

two-element integer vector

Message length, specified as a two-element integer vector, [K_R; K_C]. For a full-length message, the input column vector containing the message bits to encode is arranged into a K_R-by-K_C matrix. K_R represents the number of rows in the message matrix. K_C represents the number of columns in the message matrix. For more information about K_R and K_C, see "Turbo Product Code Construction" on page 2-827. For a list of valid ($N(i)$,$K(i)$) code pairs, see "Component Codes" on page 2-826.

Data Types: `double`

S — Shortened message length

two-element integer vector

Shortened message length, specified as a two-element integer vector, [S_R; S_C]. For a shortened message, the input column vector containing the message bits to encode is arranged into an S_R-by-S_C matrix. S_R represents the number of rows in the matrix. S_C represents the number of columns in the matrix. For more information about S_R and S_C, see "Turbo Product Code Construction" on page 2-827.

When you specify this parameter, specify N and K vectors for the full-length TPC codes that are shortened to ($N(i)$–$K(i)$+$S(i)$, $S(i)$) codes.

Data Types: `double`

Output Arguments

code — TPC-encoded message

column vector

TPC-encoded message, returned as a column vector with the same data type as the input message bits.

- For full-length input messages, the length of the returned column vector is the product of the elements in `N`.
- For shortened input messages, the length of the returned column vector is the product of the elements in (`N–K+S`).

More About

Component Codes

This table lists the supported component code pairs for the row (N_R,K_R) and column (N_C,K_C) parameters.

- N_R and K_R represent the number of rows in the product code matrix and message matrix, respectively.
- N_C and K_C represent the number of columns in the product code matrix and message matrix, respectively.

Within each code type, any two component code pairs can form a 2-D TPC code. The table also includes the error-correction capability for each code pair.

Code type	**Component Code Pairs**(N_R,K_R) and (N_C,K_C)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1
	(15,11)	1
	(7,4)	1
Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2
	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-

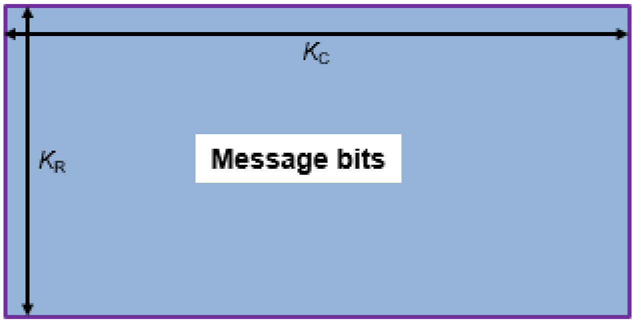
Turbo Product Code Construction

Turbo product codes (TPC) are a form of concatenated codes used as forward error-correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. This encoder implements 2-D product code encoding, as described in [1], using two “Linear Block Codes”.

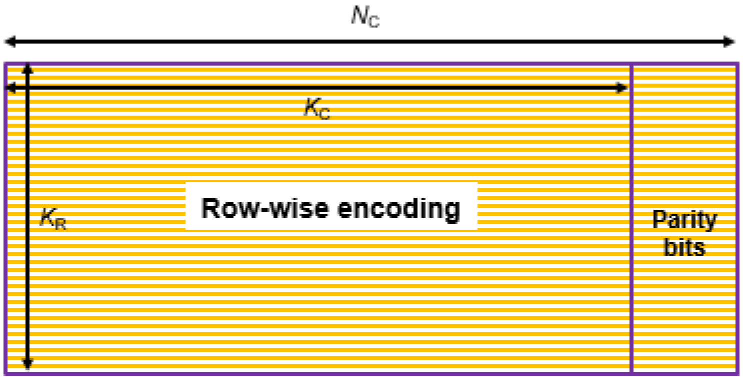
The TPC encoder accepts either full-length or shortened messages.

Construction of Full-Length Message Product Codes

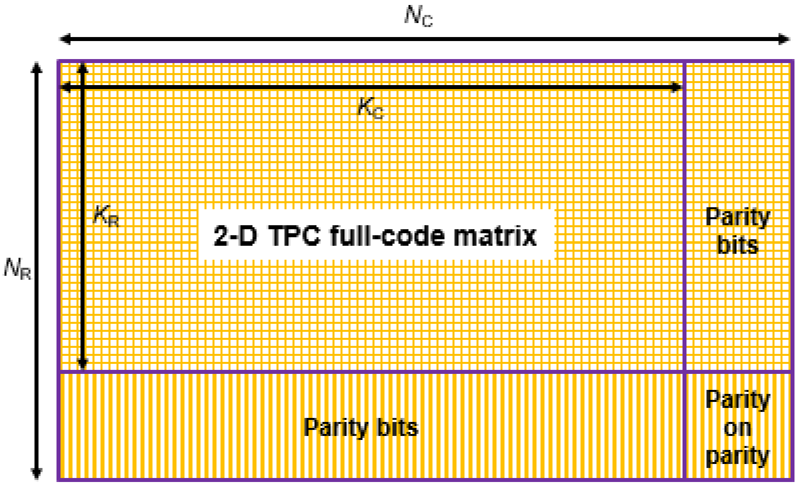
Full-length input messages are encoded using specified 2-D TPC code pairs. Row-wise encoding uses the (N_C,K_C) code pair and column-wise encoding uses the (N_R,K_R) code pair. The input vector length must be $K_R \cdot K_C$. The input message bits vector is arranged into a K_R-by-K_C matrix.



Row-wise encoding uses an (N_C,K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in a K_R-by-N_C matrix that includes parity bits added to each row.



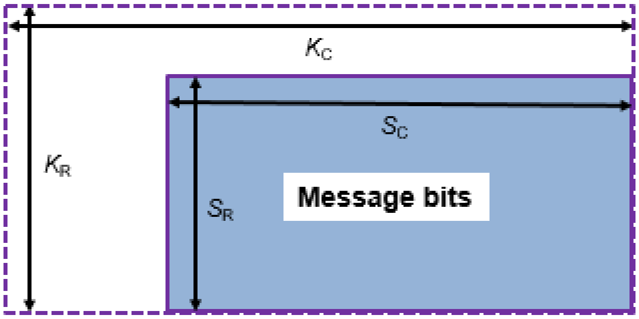
Next, column-wise encoding uses an (N_R,K_R) systematic linear block encoder on each of the N_C columns. Applying this 2-D TPC encoding to the initial K_R-by-K_C matrix results in an N_R-by-N_C matrix that includes parity bits added to each row and column.



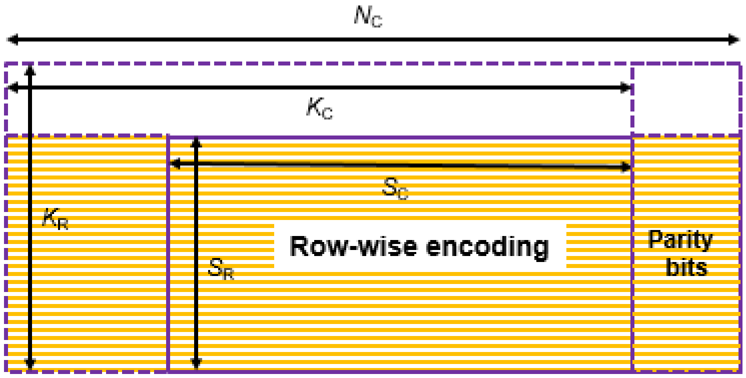
The 2-D TPC full-code matrix is reshaped into a column vector of length $N_R \cdot N_C$ and returned as the TPC-encoded output.

Construction of Shortened Message Product Codes

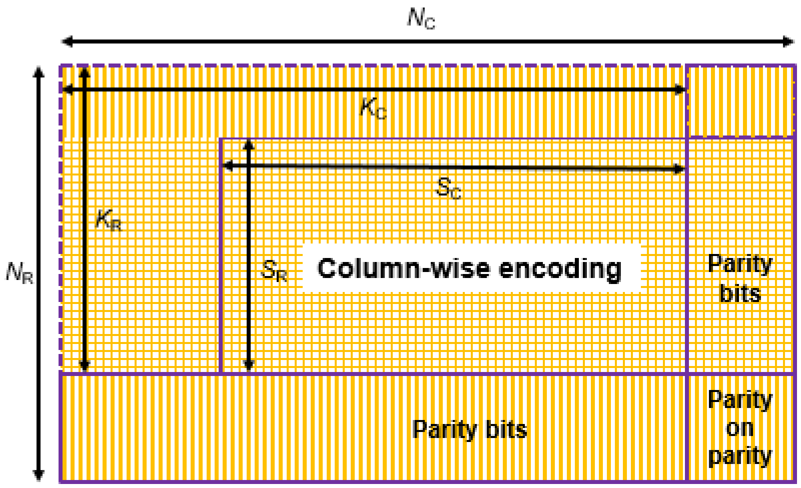
Shortened input messages are encoded using specified 2-D TPC code pairs. Row-wise encoding uses the (N_C,K_C) code pair and column-wise encoding uses an (N_R,K_R) code pair. The input vector length must be $S_R \cdot S_C$. The input shortened message bits vector is arranged into an S_R-by-S_C matrix. The shortened message matrix prepends two dimensions by padding the beginning of the message matrix with zeros. The resulting matrix is a K_R-by-K_C matrix.



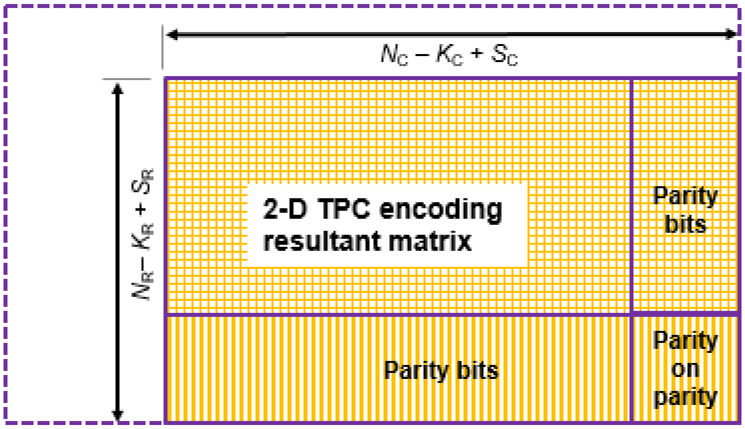
Row-wise encoding uses an (N_C,K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in a K_R-by-N_C matrix that includes parity bits added to each row.



Next, the column-wise encoding uses an (N_R,K_R) systematic linear block encoder on each of the N_C columns.



Applying this 2-D TPC encoding to the initial K_R-by-K_C matrix and excluding the zero-padded bits from the output results in an (N_R-K_R+S_R)-by-(N_C-K_C+S_C) matrix. This matrix includes parity bits added to each row and column.



The 2-D TPC shortened-code matrix is reshaped into a column vector of length (N_R–K_R+S_R) · (N_C–K_C +S_C) and returned as the TPC-encoded output.

References

[1] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Volume 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- TPC parameters `N`, `K`, and `S` must be constant values. If the value used for each of these parameters does not change, then you can assign them by expression or variable.

See Also

Functions

`bchenc` | `tpcdec`

Objects

`comm.BCHEncoder`

Blocks

TPC Encoder

Introduced in R2018a

varlms

(To be removed) Construct variable-step-size least mean square (LMS) adaptive algorithm object

Note will be removed in a future release. Use `comm.LinearEqualizer` or `comm.DecisionFeedback` instead.

Syntax

```
alg = varlms(initstep,incstep,minstep,maxstep)
```

Description

The `varlms` function creates an adaptive algorithm object that you can use with the `lineareq` function or `dfe` function to create an equalizer object. You can then use the equalizer object with the `equalize` function to equalize a signal. To learn more about the process for equalizing a signal, see “Equalization”.

`alg = varlms(initstep,incstep,minstep,maxstep)` constructs an adaptive algorithm object based on the variable-step-size least mean square (LMS) algorithm. `initstep` is the initial value of the step size parameter. `incstep` is the increment by which the step size changes from iteration to iteration. `minstep` and `maxstep` are the limits between which the step size can vary.

Properties

The table below describes the properties of the variable-step-size LMS adaptive algorithm object. To learn how to view or change the values of an adaptive algorithm object, see “Equalization”.

Property	Description
`AlgType`	Fixed value, `'Variable Step Size LMS'`
`LeakageFactor`	LMS leakage factor, a real number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, while a value of 0 corresponds to a memoryless update algorithm.
`InitStep`	Initial value of step size when the algorithm starts
`IncStep`	Increment by which the step size changes from iteration to iteration
`MinStep`	Minimum value of step size
`MaxStep`	Maximum value of step size

Also, when you use this adaptive algorithm object to create an equalizer object (via the `lineareq` or `dfe` function), the equalizer object has a `StepSize` property. The property value is a vector that lists the current step size for each weight in the equalizer.

Examples

Configuring Linear Equalizers

This example configures the recommended `comm.LinearEqualizer` System object™ and the legacy `lineareq` feature with comparable settings.

Initialize Variables and Supporting Objects

```
d = randi([0 3],1000,1);
x = pskmod(d,4,pi/4);
r = awgn(x,25);
sps = 2; %samples per symbol for oversampled cases
nTaps = 6;
txFilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',nTaps, ...
    'OutputSamplesPerSymbol',4);
rxFilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',nTaps, ...
    'InputSamplesPerSymbol',4,'DecimationFactor',2);
x2 = txFilter(x);
r2 = rxFilter(awgn(x2,25,0.5));
filterDelay = txFilter.FilterSpanInSymbols/2 + ...
    rxFilter.FilterSpanInSymbols/2; % Total filter delay in symbols
```

To compare the equalized output, plot the constellations using code such as:

```
% plot(yNew,'*')
% hold on
% plot(yOld,'o')
% hold off; legend('New Eq','Old Eq'); grid on
```

Use LMS Algorithm with Linear Equalizer

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The `LeakageFactor` property has been removed from LMS algorithm. The `comm.LinearEqualizer` System object™ assumes that leakage factor is always 1.

```
eqOld = lineareq(5,lms(0.05),pskmod(0:3,4,pi/4))
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 5
  nSampPerSym: 1
  RefTap: 1
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0500
  LeakageFactor: 1
  Weights: [0 0 0 0 0]
  WeightInputs: [0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',5,'Algorithm','LMS','StepSize',0.05, ...
    'Constellation',pskmod(0:3,4,pi/4),'ReferenceTap',1)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 5
  StepSize: 0.0500
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 1
  InputDelay: 0
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers.

```
yOld = equalize(eqOld,r);
yNew = eqNew(r);
```

Use Linear Equalizers Considering Signal Delays

Configure `lineareq` and `comm.LinearEqualizer` objects with comparable settings. The transmit and receive filters result in a signal delay between the transit and receive signals. Account for this delay by setting the `RefTap` property of the `lineareq` to a value close to the delay value in samples. Additionally, `nWeights` must be set to a value greater than `RefTap`.

```
eqOld = lineareq(filterDelay*sps+4,lms(0.01),pskmod(0:3,4,pi/4),sps);
eqOld.RefTap = filterDelay*sps+1 % Adjust to synchronize with delayed signal
```

```
eqOld =
  EqType: 'Linear Equalizer'
  AlgType: 'LMS'
  nWeights: 16
  nSampPerSym: 2
  RefTap: 13
  SigConst: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  StepSize: 0.0100
  LeakageFactor: 1
  Weights: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  WeightInputs: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
  ResetBeforeFiltering: 1
  NumSamplesProcessed: 0
```

```
eqNew = comm.LinearEqualizer('NumTaps',16,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',filterDelay*sps+1,'InputDelay',0)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 16
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 13
  InputDelay: 0
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

Call the equalizers. When `ResetBeforeFiltering` is set to `true`, each call of the `equalize` object resets the equalizer. To get the equivalent behavior call `reset` after each call of the `comm.LinearEqualizer` object.

```
yOld1 = equalize(eqOld,r,x(1:100));
yOld2 = equalize(eqOld,r,x(1:100));

yNew1 = eqNew(r,x(1:100));
reset(eqNew)
yNew2 = eqNew(r,x(1:100));
```

In the `comm.LinearEqualizer` object, `InputDelay` is used to synchronize with the delayed signal. `NumTaps` and `ReferenceTap` are independent of delay value. We can reduce the number of taps by utilizing the `InputDelay` to synchronize instead of reference tap. Reducing the number of taps also reduces equalizer self noise.

```
eqNew = comm.LinearEqualizer('NumTaps',11,'Algorithm','LMS','StepSize',0.01, ...
    'Constellation',pskmod(0:3,4,pi/4),'InputSamplesPerSymbol',sps, ...
    'ReferenceTap',6,'InputDelay',filterDelay*sps)
```

```
eqNew = comm.LinearEqualizer with properties:
  Algorithm: 'LMS'
  NumTaps: 11
  StepSize: 0.0100
  Constellation: [0.7071 + 0.7071i -0.7071 + 0.7071i -0.7071 - 0.7071i 0.7071 - 0.7071i]
  ReferenceTap: 6
  InputDelay: 12
  InputSamplesPerSymbol: 2
  TrainingFlagInputPort: false
  AdaptAfterTraining: true
  InitialWeightsSource: 'Auto'
  WeightUpdatePeriod: 1
```

```
yNew1 = eqNew(r2,x(1:100));
reset(eqNew)
yNew2 = eqNew(r2,x(1:100));
```

Algorithms

Referring to the schematics presented in “Equalization”, define *w* as the vector of all current weights w_i and define *u* as the vector of all inputs u_i. Based on the current step size, μ, this adaptive algorithm first computes the quantity

$\mu_0 = \mu + (\texttt{IncStep})\ \mathrm{Re}(gg_{\mathrm{prev}})$

where $g = ue^*$, g_{prev} is the analogous expression from the previous iteration, and the * operator denotes the complex conjugate.

Then the new step size is given by

- μ_0, if it is between `MinStep` and `MaxStep`
- `MinStep`, if $\mu_0 <$ `MinStep`
- `MaxStep`, if $\mu_0 >$ `MaxStep`

The new set of weights is given by

$(\texttt{LeakageFactor})\ w + 2\ \mu\ g^*$

Compatibility Considerations

`varlms` will be removed

Warns starting in R2020a

- varlms will be removed in a future release. Consider using `comm.LinearEqualizer` or `comm.DecisionFeedback` instead with the adaptive algorithm set to LMS.
- The `comm.LinearEqualizer` or `comm.DecisionFeedback` System objects do not have a leakage factor property. This is equivalent to setting `LeakageFactor` to `1` in the varlms function.
- For examples comparing setup of `comm.LinearEqualizer` to `lineareq`, see “Configuring Linear Equalizers” on page 2-832.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

See Also

Objects
`comm.DecisionFeedback` | `comm.LinearEqualizer` | `comm.MLSEEqualizer`

Topics
“Equalization”

Introduced before R2006a

vec2mat

(Not recommended) Change dimension

Note is not recommended. Use reshape instead. For more information, see “Compatibility Considerations”.

Syntax

```
mat = vec2mat(vec,matcol)
mat = vec2mat(vec,matcol,padding)
[mat,padded] = vec2mat( ___ )
```

Description

mat = vec2mat(vec,matcol) converts vector vec to matrix mat with matcol columns. The function creates the matrix one row at a time, filling the rows with elements from vec in order. If the length of vec is not a multiple of matcol, then the function pads the last row of mat with zeros until the row contains matcol elements.

mat = vec2mat(vec,matcol,padding) specifies values for the function to use to pad the last row of mat. The function uses the value from padding in order.

[mat,padded] = vec2mat(___) also returns padded, the number of padded elements in the last row of mat. You can specify any of the input argument combinations from previous syntaxes.

Examples

Reshape Vector and Add Padding

This example uses shows you how to add padding, as needed, when converting a vector to matrix.

Create a vector that will be converted to a matrix and a vector to provide padding values.

```
vec = [10;20;30;40;50];
padding = [1,2;3,4;5,6];
n = 4;
```

When using vec2mat to convert the vector to a matrix, the function determines needed padding.

```
[mat4,numPadded4] = vec2mat(vec,n,padding)
```

```
mat4 =
    10    20    30    40
    50     1     3     5
numPadded4 =
     3
```

When using reshape to convert the vector to a matrix, the needed padding must be computed and appended to the vector before converting the vector to a matrix.

```
numPadded = mod(numel(vec),n);
if numPadded > 0
    numPadded = n - numPadded
    mat = reshape([vec.' padding(1:numPadded)], n, []).'
else
    numPadded % No padding required
    mat = reshape(vec.', n, []).'
end

numPadded =
     3
mat =
    10    20    30    40
    50     1     3     5
```

Input Arguments

vec — Input array

vector

Input array, specified as a vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical`
Complex Number Support: Yes

`matcol` — Number of columns

positive integer

Number of columns for the output matrix `mat`, specified as a positive integer. If the length of `vec` is not a multiple of `matcol`, then the function pads the last row of `mat` with zeros until the row contains `matcol` elements.

Data Types: `double`

padding — Padding values

vector | matrix

Padding values for the last row of `mat`, specified as a vector or matrix. The `padding` input inherits the data type of the `vec` input. The function uses the values from `padding` in order. If `padding` has fewer elements than what the function needs to complete the last row of `mat`, then the function repeats the last element of `padding` until `mat` is full.

Output Arguments

mat — Output array

matrix

Output array, returned as a matrix with elements from `vec` and having `matcol` columns. The output inherits the data type of the input. The number of rows is equal to `ceil(length(vec)/matcol)`.

padded — Number of padded elements

positive integer

Number of padded elements in the last row of `mat`, returned as a positive integer.

Compatibility Considerations

`vec2mat` is not recommended

Not recommended starting in R2020a

- vec2mat is not recommended. Use `reshape` instead.
- Given a vector input, `reshape` creates its corresponding matrix one column at a time (instead of one row at a time).
- `reshape` requires its input and output arrays to have the same number of elements, whereas `vec2mat` pads its output matrix if necessary.
- For an example comparing use of `reshape` to `vec2mat`, see “Reshape Vector and Add Padding” on page 2-837.

See Also

`reshape`

Introduced before R2006a

vitdec

Convolutionally decode binary data by using Viterbi algorithm

Syntax

```
decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype)
decodedout = vitdec(codedin,trellis,tbdepth,opmode,'soft',nsdec)
decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype,puncpat)
decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype,puncpat,eraspat)
decodedout = vitdec(codedin,trellis,tbdepth,'cont',dectype, ___ ,imetric,
istate,iinput)
[decodedout,fmetric,fstate,finput] = vitdec(codedin,trellis,tbdepth,'cont',
___ )
```

Description

`decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype)` decodes each symbol of the `codedin` input by using the Viterbi algorithm. All other inputs specify the convolutional coding trellis, traceback depth, operating mode, and decision type, respectively and collectively configure the Viterbi algorithm at runtime.

`decodedout = vitdec(codedin,trellis,tbdepth,opmode,'soft',nsdec)` configures the Viterbi algorithm for soft-decision decoding for `dectype` and with `nsdec` bits of quantization.

`decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype,puncpat)` decodes each symbol of the punctured `codedin` input, where `puncpat` is the puncture pattern.

`decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype,puncpat,eraspat)` specifies an erasure pattern, `eraspat`. To not use puncturing, specify `puncpat` as `[]`.

`decodedout = vitdec(codedin,trellis,tbdepth,'cont',dectype, ___ ,imetric, istate,iinput)` specifies a continuous operation mode for `opmode` for any of the preceding syntaxes. The decoder starts with its initial state metrics, traceback states, and traceback inputs specified by `imetric`, `istate`, and `iinput`, respectively.

Continuous operation mode enables you to save the internal state information of the decoder for use in subsequent calls to this function. Repeated calls to this function can be useful if your data is partitioned into a series of vectors that you process within a loop. For workflows that require repeated calls to the Viterbi decoding algorithm, see “Tips” on page 2-849.

`[decodedout,fmetric,fstate,finput] = vitdec(codedin,trellis,tbdepth,'cont', ___ )` also returns the final state metrics, traceback states, and traceback inputs at the end of the decoding process when using a continuous operation mode for any of the preceding syntaxes. Use `fmetric`, `fstate`, and `finput` as the initial settings of `imetric`, `istate`, and `iinput`, respectively, in subsequent calls to this function. For workflows that require repeated calls to the Viterbi decoding algorithm, see “Tips” on page 2-849.

Examples

Decode Convolutional Code by Using Viterbi Decoder

Convolutionally encode a vector of 1s by using the `convenc` function, and decode it by using the `vitdec` function.

Define a trellis structure, by using the `poly2trellis` function. Use the trellis structure to configure the `convenc` function when encoding a vector of ones.

```
trellis = poly2trellis([4 3],[4 5 17;7 4 2]);
x = ones(100,1);
code = convenc(x,trellis);
```

When decoding the encoded message, configure the Viterbi decoder to use the trellis structure defined previously, a traceback depth of 2, the truncated operating mode, and hard decisions.

```
tb = 2;
decoded = vitdec(code,trellis,tb,'trunc','hard');
```

Verify that the decoded message is a vector of 100 1s.

```
isequal(decoded,ones(100,1))
```

```
ans = logical
   1
```

Decode Punctured Signal Using Viterbi Algorithm

Apply Viterbi decoding to a punctured signal. The puncturing changes the code rate from 1/2 to 3/4.

Initialize parameters for the encoding and decoding operations.

```
trellis = poly2trellis(7,[171 133])
```

```
trellis = struct with fields:
     numInputSymbols: 2
    numOutputSymbols: 4
           numStates: 64
          nextStates: [64x2 double]
             outputs: [64x2 double]
```

```
puncpat = [1;1;0;1;1;0];
tbdepth = 96;
opmode = 'trunc';
dectype = 'hard';
```

Calculate the unpunctured and punctured code rates.

```
K = log2(trellis.numInputSymbols); % Number of input streams
N = log2(trellis.numOutputSymbols); % Number of output streams
unpunc_coderate = K/N % Unpunctured code rate
```

```
unpunc_coderate = 0.5000
```

```
punc_coderate = (K/N)*(length(puncpat)/sum(puncpat)) % Punctured code rate
punc_coderate = 0.7500
```

Convolutionally encode an all 1s bit message with puncturing applied to the coded output.

```
msg = ones(100*length(puncpat),1);
puncturedcode = convenc(msg,trellis,puncpat);
```

Show the lengths of the message, the punctured code, and the puncture pattern.

```
length(msg)
```

```
ans = 600
```

```
length(puncturedcode)
```

```
ans = 800
```

```
length(puncpat)
```

```
ans = 6
```

Apply Viterbi decoding to the punctured coded message. Compare the decoded output to the original message. Even with puncturing applied to the coded message, the Viterbi decoding recovered the message with zero error.

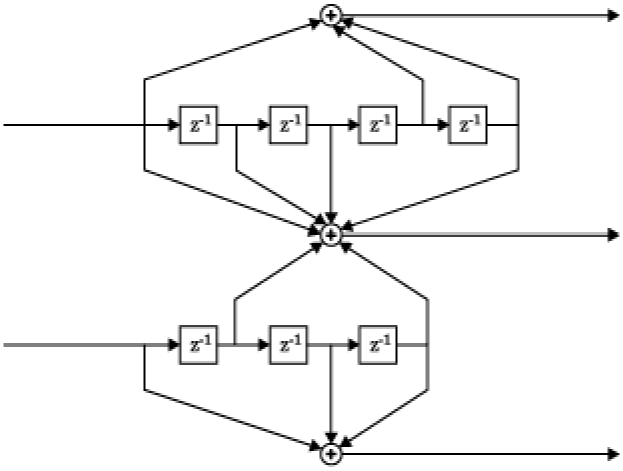
```
codedin = puncturedcode;
decodedout = vitdec(codedin,trellis,tbdepth,opmode,dectype,puncpat);
```

```
isequal(msg,decodedout)
```

```
ans = logical
   1
```

Estimate BER for Rate 2/3 Convolutional Code

Estimate the bit error rate (BER) simulation for a link that uses a rate 2/3 convolutional code, applies 16-QAM modulation, and transmits data through an AWGN channel. This diagram shows a rate 2/3 encoder with two input streams, three output streams, and seven shift registers.



Define the convolutional coding trellis represented by the diagram.

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

```
trellis = struct with fields:
     numInputSymbols: 4
    numOutputSymbols: 8
           numStates: 128
          nextStates: [128x4 double]
             outputs: [128x4 double]
```

```
K = log2(trellis.numInputSymbols); % Number of input bit streams
N = log2(trellis.numOutputSymbols); % Number of output bit streams
coderate = K/N;

fprintf('K is %d and N is %d. The code rate is %3.2f.\n', ...
    K,N,coderate)
```

```
K is 2 and N is 3. The code rate is 0.67.
```

Set the modulation order, and compute the number of bits per modulation symbol. Generate random binary data. The input bit stream must be a multiple of number of the input bit streams (K) for the coding operation and must be a multiple of the number of bits per modulation symbol (bps) for the modulation operation.

```
M = 16; % Modulation order
bps = log2(M); % Bits per modulation symbol
numSymPerFrame = 5000;
dataIn = randi([0 1],K*bps*numSymPerFrame,1);
```

Convolutionally encode the input data.

```
codedout = convenc(dataIn,trellis);
```

Apply 16-QAM modulation to the encoded symbols.

```
txSig = qammod(codedout,M,'InputType','bit');
```

Using the number of bits per symbol (bps) and the code rate (coderate), convert the ratio of energy per bit to noise power spectral density (EbNo) to an signal-to-noise (snr) value for use by the awgn function. Convert a 10 dB Eb/No to an equivalent SNR ratio. Pass the signal through an AWGN channel.

```
EbNo = 9;
snr = EbNo + 10*log10(bps*coderate);
rxSig = awgn(txSig,snr,'measured');
```

Demodulate the received signal.

```
demodSig = qamdemod(rxSig,M,'OutputType','bit');
```

Specify the traceback depth of the Viterbi decoder.

```
tbdepth = 16;
```

Decode the binary demodulated signal by using a Viterbi decoder operating in a continuous termination mode.

```
dataOut = vitdec(demodSig,trellis,tbdepth,'cont','hard');
```

Calculate the delay through the decoder, and account for the decoding delay when computing the BER. Compare the coded BER with the theoretical uncoded BER to see the improved BER for the coded data.

```
decDelay = K*tbdepth;
berCoded = biterr(dataIn(1:end-decDelay),dataOut(decDelay+1:end))/length(dataOut(decDelay+1:end)
berUncoded = berawgn(EbNo,'qam',M);
fprintf('  The coded BER is %6.5f.\nThe uncoded BER is %6.5f.\n', ...
    berCoded,berUncoded)
```

```
  The coded BER is 0.00060.
The uncoded BER is 0.00439.
```

Input Arguments

codedin — Convolutionally encoded message

vector of binary values | vector of numeric values

Convolutionally encoded message, specified as a vector of binary or numeric values. Each symbol in `codedin` consists of log2(`trellis`.`numOutputSymbols`) bits.

Data Types: `double` | `logical`

trellis — Trellis description

structure

Trellis description, specified as a MATLAB structure that contains the trellis description for a rate *K*/*N* code. *K* represents the number of input bit streams, and *N* represents the number of output bit streams.

The trellis structure contains these fields. You can either use the `poly2trellis` function to create the trellis structure or create it manually. For more about this structure, see “Trellis Description of a Convolutional Code” and the `istrellis` function.

numInputSymbols — Number of symbols input to encoder

2^K

Number of symbols input to the encoder, specified as an integer equal to 2^K, where *K* is the number of input bit streams.

Data Types: `double`

numOutputSymbols — Number of symbols output from encoder

2^N

Number of symbols output from the encoder, specified as an integer equal to 2^N, where *N* is the number of output bit streams.

Data Types: `double`

numStates — Number of states in encoder

power of 2

Number of states in the encoder, specified as a power of 2.

Data Types: double

nextStates — Next states
matrix of integers

Next states for all combinations of current states and current inputs, specified as a matrix of integers. The matrix size must be numStates-by-2^K.

Data Types: double

outputs — Outputs
matrix of octal numbers

Outputs for all combinations of current states and current inputs, specified as a matrix of octal numbers. The matrix size must be numStates-by-2^K.

Data Types: double

Data Types: struct

tbdepth — Traceback depth
positive integer

Traceback depth, specified as a positive integer. For more information, see "Traceback Depth Estimates" on page 2-849.

Data Types: double

opmode — Operating mode
'cont' | 'term' | 'trunc'

Operating mode, specified as 'cont', 'term', or 'trunc'. This input indicates the operating mode of the decoder and these assumptions made about the operation of the corresponding encoder.

- 'cont' — Specifies continuous operating mode. In continuous operating mode, 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 input tbdepth symbols elapses before the first decoded symbol appears in the output. This mode is appropriate when you call this function repeatedly and want to preserve continuity between successive calls. For workflows that require repeated calls to the Viterbi decoding algorithm, see "Tips" on page 2-849.
- 'term' — Specifies terminated operating mode. In terminated operating mode, the encoder is assumed to have started and ended at the all-zeros state, which is true for the default syntax of the convenc function. The decoder traces back from the all-zeros state. This mode incurs zero delay.

  This mode is appropriate when the message input to the convenc function has enough zeros at its end to fill all memory registers of the encoder. The zero-valued tail bits flush all message data bits out of the encoder. Using the polynomial description of the encoder, for an encoder with K input bits and the constraint length vector ConstraintLength, the number of zeros required to flush the encoder is K × max(ConstraintLength – 1) . The constraint length vector is the first input argument to the poly2trellis function.
- 'trunc' — Specifies truncated operating mode. In truncated operating mode, the encoder is assumed to have started at the all-zeros state. The decoder traces back from the state with the best metric. This mode incurs zero delay. This mode is appropriate when you cannot assume the encoder ended at the all-zeros state and when you do not want to preserve continuity between successive calls to this function.

For the `'term'` and `'trunc'` modes, the traceback depth, `tbdepth`, must be a positive integer, less than or equal to the number of input symbols in input `codedin`.

For more information, see “Traceback and Decoding Delay” on page 2-848 and “Traceback Depth Estimates” on page 2-849.

Data Types: `char` | `string`

`dectype` — Decoding type
`'unquant'` | `'hard'` | `'soft'`

Decoding type, specified as `'unquant'`, `'hard'`, or `'soft'`. This parameter indicates the type of decoding decision that the decoder makes and influences the type of data the decoder expects as input in `codedin`.

- `'unquant'` — The decoder expects signed numeric input values, where positive values map to a logical `0` and negative values map to a logical `1`.
- `'hard'` — The decoder expects binary input values of `0` or `1`.
- `'soft'` — The decoder expects integer input values in the range [0, (2^{nsdec} – 1)]. The Viterbi algorithm decision criteria regards 0 as the most confident 0 and 2^{nsdec} – 1 as the most confident 1.

Data Types: `char` | `string`

`nsdec` — Number of soft decision quantization bits
integer in the range [1, 13]

Number of soft decision quantization bits, specified as a integer in the range [1, 13]. For reference, soft decision decoding with 3 bits of quantization improves error decoding recovery by approximately 2 dB as compared to hard decision decoding.

Dependencies

To enable this input argument set the `dectype` input argument to `'soft'`.

Data Types: `double`

`puncpat` — Puncture pattern
vector of binary values

Puncture pattern, specified as a vector of binary values. Indicate punctured bits with `0`s and unpunctured bits with `1`s. The input code length divided by the number of 1s in the puncture pattern times the length of the puncture pattern must be an integer multiple of the number of bits in an input symbol.

Data Types: `double`

`eraspat` — Erasure pattern
vector of binary values

Erasure pattern, specified as a vector of binary values. Indicate erased bits with `1`s and nonerased bits with `0`s. The length of the erasures pattern must be the same as the input code length.

Data Types: `double`

`imetric` — Decoder state metrics
integer | vector of integer values

Decoder state metrics, specified as an integer or a vector of integer values. Each value in `imetric` represents the starting state metric of the corresponding decoder state. When you set `imetric` to a vector, the length must be `trellis.numStates`. To use the default decoder state metrics, specify `imetric` as `[]`.

Dependencies

To enable this input argument set the `opmode` input argument to `'cont'`.

Data Types: `double`

`istate` — Decoder initial traceback states

matrix of integer values

Decoder initial traceback states, specified as a `trellis.numStates`-by-`tbdepth` matrix of integer values in the range [0, (`trellis.numStates` – 1)]. To use the default decoder initial traceback states, specify `istate` as `[]`.

Inputs `istate` and `iinput` jointly specify the initial traceback memory of the decoder. If the encoder schematic has more than one input stream, the shift register that receives the first input stream provides the least significant bits in `istate`, and the shift register that receives the last input stream provides the most significant bits in `istate`.

Dependencies

To enable this input argument set the `opmode` input argument to `'cont'`.

Data Types: `double`

`iinput` — Decoder initial traceback inputs

matrix of integer values

Decoder initial traceback inputs, specified as a `trellis.numStates`-by-`tbdepth` matrix of integer values in the range [0, (`trellis.numStates` – 1)]. To use the default decoder initial traceback inputs, specify `iinput` as `[]`.

Inputs `istate` and `iinput` jointly specify the initial traceback memory of the decoder.

Dependencies

To enable this input argument set the `opmode` input argument to `'cont'`.

Data Types: `double`

Output Arguments

`decodedout` — Decoded message

vector of binary values

Decoded message, returned as a vector of binary values. Each symbol in the vector `decodedout` consists of log2(`trellis.numInputSymbols`) bits.

`fmetric` — Decoder final state metrics

vector of integer values

Decoder final state metrics, returned as an vector of integer values with `trellis.numStates` elements. Each value in `fmetric` represents the final state metric of the corresponding decoder state.

When calling `vitdec` in continuous mode, `fmetric` is typically used to set `imetric` for subsequent calls to the `vitdec` function.

Dependencies

This output applies when the `opmode` parameter is set to `'cont'`.

Data Types: `double`

`fstate` — Decoder final traceback states

matrix of integer values

Decoder final traceback states, returned as a `trellis.numStates`-by-`tbdepth` matrix of integer values in the range [0, (`trellis.numStates` – 1)].

Outputs `fstate` and `finput` jointly describe the final traceback memory of the decoder. If the encoder schematic has more than one input stream, the shift register that receives the first input stream provides the least significant bits in `fstate`, and the shift register that receives the last input stream provides the most significant bits in `fstate`.

When calling `vitdec` in continuous mode, `fstate` is typically used to set `istate` for subsequent calls to the `vitdec` function.

Dependencies

This output applies when the `opmode` parameter is set to `'cont'`.

Data Types: `double`

`finput` — Decoder final traceback inputs

matrix of integer values

Decoder final traceback inputs, returned as a `trellis.numStates`-by-`tbdepth` matrix of integer values in the range [0, (`trellis.numStates` – 1)].

Outputs `fstate` and `finput` jointly specify the final traceback memory of the decoder.

When calling `vitdec` in continuous mode, `finput` is typically used to set `iinput` for subsequent calls to the `vitdec` function.

Dependencies

This output applies when the `opmode` parameter is set to `'cont'`.

Data Types: `double`

More About

Traceback and Decoding Delay

The traceback depth influences the decoding delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

- For the continuous operating mode, the decoding delay is equal to the number of traceback depth symbols.
- For the truncated or terminated operating mode, the decoding delay is zero. In this case, the traceback depth must be less than or equal to the number of symbols in each input.

Traceback Depth Estimates

As a general estimate, a typical traceback depth value is approximately two to three times (*ConstraintLength* – 1) / (1 – *coderate*). The constraint length of the code, *ConstraintLength*, is equal to (log2(`trellis.numStates`) + 1). The *coderate* is equal to (*K* / *N*) × (length(*PuncturePattern*) / sum(*PuncturePattern*).

K is the number of input symbols, *N* is the number of output symbols, and *PuncturePattern* is the puncture pattern vector.

For example, applying this general estimate, results in these approximate traceback depths.

- A rate 1/2 code has a traceback depth of 5(*ConstraintLength* – 1).
- A rate 2/3 code has a traceback depth of 7.5(*ConstraintLength* – 1).
- A rate 3/4 code has a traceback depth of 10(*ConstraintLength* – 1).
- A rate 5/6 code has a traceback depth of 15(*ConstraintLength* – 1).

For more information, see [7].

Tips

- Consider using the `comm.ViterbiDecoder` System object when successive calls to the Viterbi algorithm are needed. The System object simplifies the required state retention operation by inherently retaining state metrics, traceback states, and inputs between calls.

References

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

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

[3] Heller, J., and I. Jacobs. “Viterbi Decoding for Satellite and Space Communication.” *IEEE Transactions on Communication Technology* 19, no. 5 (October 1971): 835–48. https://doi.org/10.1109/TCOM.1971.1090711.

[4] Yasuda, Y., K. Kashiki, and Y. Hirata. “High-Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding.” *IEEE Transactions on Communications* 32, no. 3 (March 1984): 315–19. https://doi.org/10.1109/TCOM.1984.1096047.

[5] Haccoun, D., and G. Begin. “High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 37, no. 11 (November 1989): 1113–25. https://doi.org/10.1109/26.46505.

[6] Begin, G., D. Haccoun, and C. Paquin. “Further Results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 38, no. 11 (November 1990): 1922–28. https://doi.org/10.1109/26.61470.

[7] Moision, B. "A Truncation Depth Rule of Thumb for Convolutional Codes." *In Information Theory and Applications Workshop* (January 27 2008-February 1 2008, San Diego, California), 555-557. New York: IEEE, 2008.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

The input arguments `trellis`, `opmode`, `tbdepth`, `dectype`, and `puncpat` must be compile-time constants. For more information, see `coder.Constant`.

See Also

Functions

`convenc` | `distspec` | `istrellis` | `poly2trellis`

Objects

`comm.APPDecoder` | `comm.ConvolutionalEncoder` | `comm.TurboDecoder` | `comm.ViterbiDecoder`

Topics

"Convolutional Codes"
"Trellis Description of a Convolutional Code"
"Estimate BER for Hard and Soft Decision Viterbi Decoding"

Introduced before R2006a

wgn

Generate white Gaussian noise samples

Syntax

```
noise = wgn(m,n,power)
noise = wgn(m,n,power,imp)
noise = wgn(m,n,power,imp,randobject)
noise = wgn(m,n,power,imp,seed)

noise = wgn( ___ ,powertype)
noise = wgn( ___ ,outputtype)
```

Description

`noise = wgn(m,n,power)` generates an `m`-by-`n` matrix of white Gaussian noise samples in volts. `power` specifies the power of `noise` in dBW.

`noise = wgn(m,n,power,imp)` specifies the load impedance in ohms.

`noise = wgn(m,n,power,imp,randobject)` specifies a random number stream object to use when generating the matrix of white Gaussian noise samples. For information about producing repeatable noise samples, see “Tips” on page 2-853.

`noise = wgn(m,n,power,imp,seed)` specifies a seed value for initializing the normal random number generator that is used when generating the matrix of white Gaussian noise samples. For information about producing repeatable noise samples, see “Tips” on page 2-853.

`noise = wgn( ___ ,powertype)` specifies the units of `power` as `'dBW'`, `'dBm'`, or `'linear'` in addition to the input arguments in any of the previous syntaxes.

`noise = wgn( ___ ,outputtype)` specifies the output type as `'real'` or `'complex'` in addition to the input arguments in any of the previous syntaxes.

Examples

Generate White Gaussian Noise

Generate real and complex white Gaussian noise (WGN) samples. Check the power of output WGN matrices.

Generate a 1000-element column vector of real WGN samples and confirm that the power is approximately 1 watt, which is 0 dBW.

```
y1 = wgn(1000,1,0);
var(y1)
```

```
ans = 0.9979
```

Generate a 1000-element column vector of complex WGN samples and confirm that the power is approximately 0.25 watts, which is –6 dBW.

```
y2 = wgn(1000,1,-6,'complex');
var(y2)
```

```
ans = 0.2522
```

Input Arguments

m — Number of white Gaussian noise samples

positive integer

Number of white Gaussian noise samples desired per channel, specified as a positive integer.

Data Types: double

n — Number of channels

positive integer

Number of channels of white Gaussian noise samples desired, specified as a positive integer.

Data Types: double

power — Power of noise samples

scalar

Power of noise samples, specified as a scalar. The default units for power is dBW. Use powertype to change the units of power.

Data Types: double

imp — Load impedance

1 (default) | scalar

Load impedance in ohms, specified as a scalar.

Data Types: double

randobject — Random number stream object

RandStream object

Random number stream object, specified as a RandStream object. The state of the random stream object determines the sequence of numbers produced by the randn function. Configure the random stream object using the reset (RandStream) function and its properties.

wgn generates normal random noise samples using randn. The randn function uses one or more uniform values from the RandStream object to generate each normal value.

For information about producing repeatable noise samples, see “Tips” on page 2-853.

seed — Random number generator seed

nonnegative integer

Random number generator seed, specified as a nonnegative integer. For more information on the random number generator, see randn.

powertype — Signal power unit
'dBW' (default) | 'dBm' | 'linear'

Signal power unit, specified as 'dBW', 'dBm', or 'linear'. Linear power is in watts.

outputtype — Output type
'real' (default) | 'complex'

Output type, specified as 'real' or 'complex'. If outputtype is 'complex', then the real and imaginary parts of noise each have a noise power of (power / 2).

Output Arguments

noise — Output white Gaussian noise samples
scalar | vector | array

Output white Gaussian noise samples in volts, returned as an m-by-n matrix.

Note Unless the default impedance for imp is changed, a load of 1 ohm is used for power calculations.

Tips

- To generate repeatable white Gaussian noise samples, use one of these tips:
  - Provide a static seed value as an input to wgn.
  - Use the reset (RandStream) function on the randobject before passing it as an input to wgn.
  - Provide randobject in a known state as an input to wgn. For more information, see RandStream.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Code generation supported, except for syntaxes that include a RandStream object.

See Also

Functions
RandStream | awgn | randn

Topics
"Sources and Sinks"

Introduced before R2006a

winner2.AntennaArray

Create antenna array

Syntax

```
antArray = winner2.AntennaArray
antArray = winner2.AntennaArray(Name,Value)
```

Description

Download Required: To use this function, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

`antArray = winner2.AntennaArray` returns a structure representing an antenna array with one isotropic antenna element. Both the antenna array and the single element have no rotation and are located at the origin, [0;0;0].

`antArray = winner2.AntennaArray(Name,Value)` returns a structure representing an antenna array defined using one or more `Name,Value` pair arguments.

For more information, see “Antenna Array Model” on page 2-858.

Examples

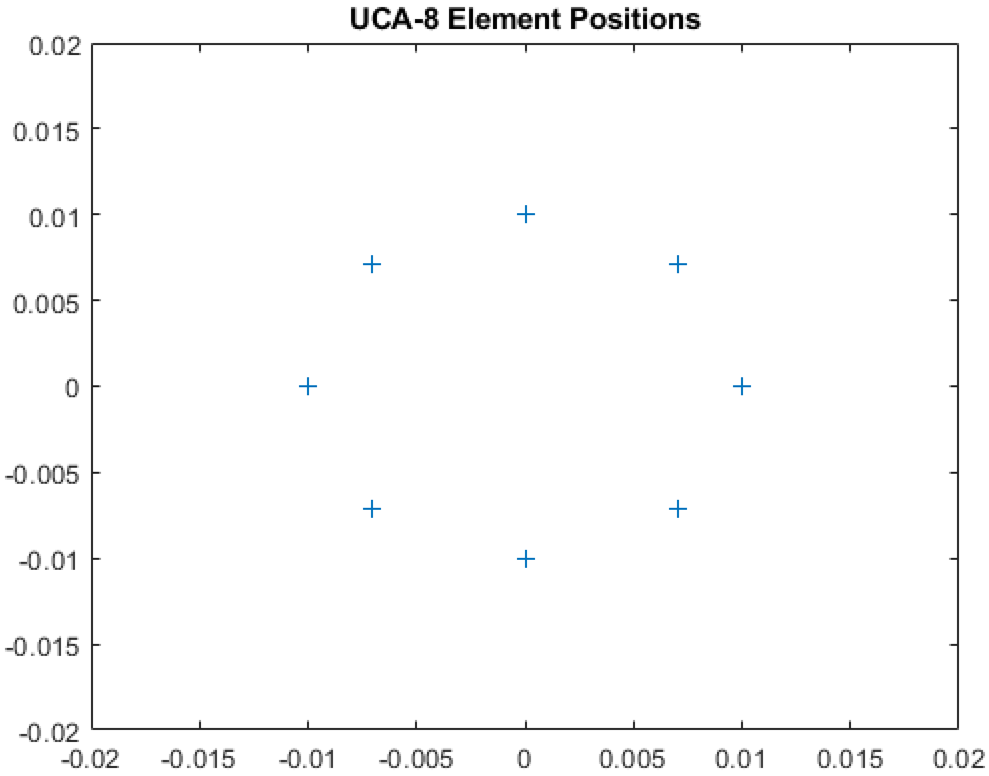
Create WINNER II Eight Element Uniform Circular Array

Use the `winner2.AntennaArray` function to create an eight element uniform circular array (UCA-8) with a 1 cm radius.

```
UCA8 = winner2.AntennaArray('UCA',8,0.01);
```

Plot element positions.

```
pos = {UCA8.Element(:).Pos};
plot(cellfun(@(x) x(1),pos),cellfun(@(x) x(2),pos),'+');
xlim([-0.02 0.02]);
ylim([-0.02 0.02]);
title('UCA-8 Element Positions');
```



Create WINNER II Two Element Uniform Linear Array

Use the `winner2.AntennaArray` function to create a two element uniform linear array (ULA-2) with 50 cm spacing and the dipole elements slanted at +45 and -45 degrees.

```
az = -180:179; % 1-degree spacing
pattern = cat(1,shiftdim(winner2.dipole(az,45),-1), ...
    shiftdim(winner2.dipole(az,-45),-1));
ULA2 = winner2.AntennaArray('ULA',2,0.5, ...
    'FP-ECS',pattern,'Azimuth',az);
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Pos',[1 0 0; 0 1 0],'Rot',[0 0 0; 0 pi() 0]` indicates the coordinates and rotation angles for two antenna elements.

Pos — Position of each antenna element

0 (default) | column vector | matrix

Position of each antenna element, specified as the comma-separated pair consisting of 'Pos' and a column vector or an N_E-by-3 matrix. The three columns represent the x-, y-, and z-coordinates in meters from the origin. N_E indicates the number of elements in the antenna array. The elements have no rotation. When there is more than one element, the 'Element' field of antArray is a row vector of structures representing all the elements.

Example: 'Pos',[63.1 10.2 11.5; 62 11 12] indicates the coordinates for two antenna elements.

Data Types: double

Rot — Rotation angle of each antenna element

0 (default) | column vector | matrix | optional

Rotation angle of each antenna element, specified as the comma-separated pair consisting of 'Rot' and a column vector or an N_E-by-3 matrix. The three columns represent the Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians. N_E indicates the number of elements in the antenna array. Rot only applies when Pos is specified. If not specified with Pos, the rotation angle is 0.

Example: 'Rot',[2 1.5 0; 0 pi() 0] indicates the rotation angles for two antenna elements.

Data Types: double

UCA — Uniform circular antenna array

N,1 (default) | N,Rad

Uniform circular antenna array, specified as the comma-separated pair consisting of 'UCA' and N,Rad. In this argument, N indicates the number of elements (N_E) and Rad indicates the radius in meters. If Rad is not specified, the default radius is 1 meter.

Example: 'UCA',8,0.5 indicates an eight element uniform circular array with 0.5 meter radius.

Data Types: double

ULA — Uniform linear antenna array

N,1/N (default) | N,Spacing

Uniform linear antenna array, specified as the comma-separated pair consisting of 'ULA' and N,Spacing. In this argument, N indicates the number of elements (N_E) and Spacing indicates the separation between adjacent elements in meters. If Spacing is not specified, the default separation is 1/N meters.

ULA elements are placed along *x-axis* with the center of the array at [0;0;0]. For an even number of elements, there is no antenna element at [0;0;0].

Example: 'ULA',3,0.25 indicates a three element uniform linear array with 0.25 meter spacing between adjacent elements.

Data Types: double

FP-ECS — Field pattern of element coordinate system

4-D array

Field pattern of element coordinate system, specified as the comma-separated pair consisting of `'FP-ECS'` and a P-by-2-by1-by-N_{AZ} array.

- The first dimension, P, can be either 1 or any number greater than or equal to the number of elements in the antenna array (N_E). When $P = 1$, the same pattern applies to all elements. When $P > N_E$, the first N_E rows apply.
- The second dimension, `2`, indicates that two polarizations characterize the field pattern. The first dimension in the field pattern stores vertical polarization, and the second one stores horizontal polarization.
- The third dimension, `1`, indicates that one elevation angle characterizes the field pattern.
- The fourth dimension, N_{AZ}, is the number of field pattern samples taken between –180 and 180 degrees. N_{AZ} equals the number of elements specified in `Azimuth` or when `Azimuth` is not present it equals the number of equidistant field pattern samples taken over azimuth angle.

Data Types: `double`

`FP-ACS` — Field pattern array coordinate system

4-D array

Field pattern array coordinate system, specified as the comma-separated pair consisting of `'FP-ACS'` and a P-by-2-by1-by-N_{AZ} array. Array format is the same as the `FP-ECS` syntax, except that the field pattern is specified in the array-coordinate-system (ACS).

- The first dimension, P, can be either 1 or any number greater than or equal to the number of elements in the antenna array (N_E). When $P = 1$, the same pattern applies to all elements. When $P > N_E$, the first N_E rows apply.
- The second dimension, `2`, indicates that two polarizations characterize the field pattern. The first dimension in the field pattern stores vertical polarization, and the second one stores horizontal polarization. Missing polarization dimensions of the field pattern are substituted with zeros.
- The third dimension, `1`, indicates that one elevation angle characterizes the field pattern.
- The fourth dimension, N_{AZ}, is the number of field pattern samples taken between –180 and 180 degrees. N_{AZ} equals the number of elements specified in `Azimuth` or when `Azimuth` is not present it equals the number of equidistant field pattern samples taken over azimuth angle.

Data Types: `double`

`Azimuth` — Azimuth angles for 'FP-ACS' or 'FP-ECS' field patterns

row vector

Azimuth angles for `FP-ACS` or `FP-ECS` field patterns in degrees , specified as the comma-separated pair consisting of `'Azimuth'` and an 1-by-N_{AZ} row vector. The values in the row vector indicate azimuth angles for elements in the field patterns.

Note `Azimuth` applies only when `FP-ACS` or `FP-ECS` are defined. If `Azimuth` is not specified, uniform spacing is used for elements in the field pattern.

Example: `'Azimuth',[0 10 20 90 180 270 340 350]`

Data Types: `double`

Output Arguments

`antArray` — Antenna array definition
structure

Antenna array definition, returned as a structure containing these fields.

`Name` — Antenna array name
character vector

Antenna array name, returned as a character vector.

`Pos` — Antenna array position
vector

Antenna array position, returned as a 3-by-1 vector, representing the x-, y-, and z-coordinates in meters from the origin.

`Rot` — Antenna array rotation
vector

Antenna array rotation, returned as a 3-by-1 vector, representing the Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians.

`Element` — Element definition
row vector of structures

Element definition, returned as a row vector of structures, with each structure representing one element and containing these fields.

`Pos` — Antenna array position
vector

Antenna array position, returned as a 3-by-1 vector, representing the x-, y-, and z-coordinates in meters from the origin.

`Rot` — Antenna array rotation
vector

Antenna array rotation, returned as a 3-by-1 vector, representing the Rot_X, Rot_Y, and Rot_Z rotation angles of each antenna element in radians.

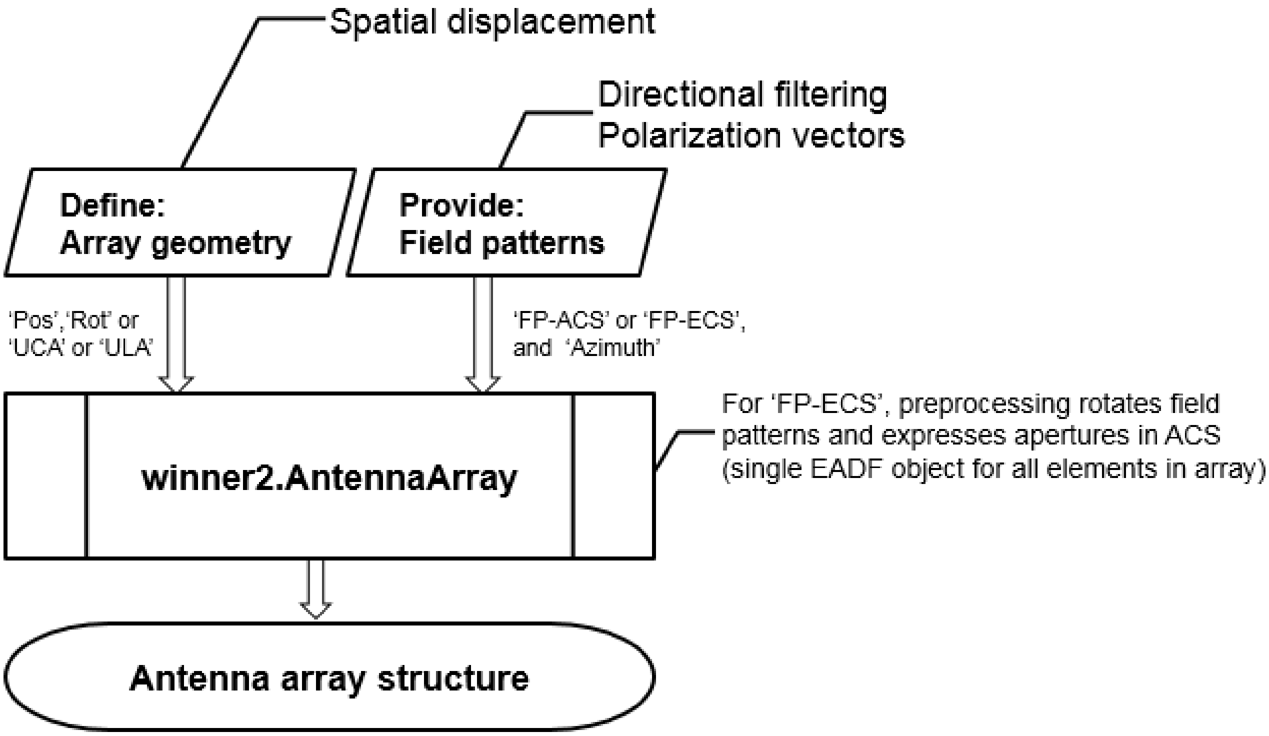
`Aperture` — Aperture definition
structure

Aperture definition, returned as a structure representing the antenna aperture.

More About

Antenna Array Model

To create an antenna array model, you must define the geometry of array elements (positions and rotation) and the element field patterns. The arguments provided to `winner2.AntennaArray` are always processed such that the array geometry is created first, and then the field patterns are assigned.



For a detailed description of the antenna array specification for the WINNER channel model, see WINNER II Channel Models [1], Section 4.1.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

`winner2.dipole` | `winner2.layoutparset`

Introduced in R2017a

winner2.dipole

Calculate field pattern of half-wavelength dipole

Syntax

```
pat = winner2.dipole(az)
pat = winner2.dipole(az,slant)
```

Description

Download Required: To use this function, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

`pat = winner2.dipole(az)` returns the azimuth field pattern of a 0-degree slanted dipole at the azimuth angles specified in `az`.

`pat = winner2.dipole(az,slant)` returns the azimuth field pattern of a slanted dipole at the azimuth angles specified in `az`.

Examples

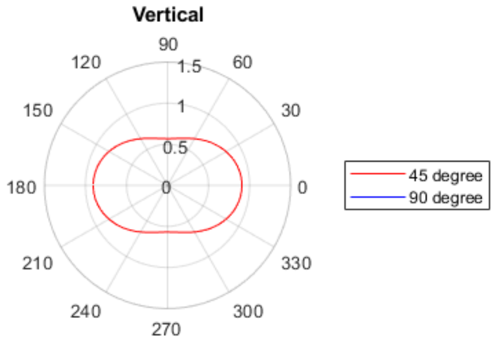
Create 45 and 90 Degree Slanted Dipoles

Create 45 and 90 degree slanted dipoles by using the `winner2.dipole` function.

```
az = -180:179; % 1 degree spacing
pattern45 = squeeze(winner2.dipole(az,45));
pattern90 = squeeze(winner2.dipole(az,90));
```

Display the antenna pattern by using the `polarplot` function.

```
fh = figure;
set(fh, 'Position', [100 100 1000 500]);
fh.Name = 'Dipole Pattern Plots';
subplot(1,2,1);
polarplot(az/180*pi,pattern45(1,:),'r');
hold on;
polarplot(az/180*pi,pattern90(1,:),'b');
rlim([0 1.5]);
legend('45 degree','90 degree');
title('Vertical');

subplot(1,2,2);
polarplot(az/180*pi,pattern45(2,:),'r');
hold on;
polarplot(az/180*pi,pattern90(2,:),'b');
rlim([0 1.5]);
legend('45 degree','90 degree');
title('Horizontal');
```



Input Arguments

az — Azimuth angles

vector

Azimuth angles, specified as a vector indicating the azimuth angles to compute the field pattern gain. Units are in degrees.

Data Types: `double`

`slant` — Slant angle

scalar

Slant angle, specified as a scalar representing the counterclockwise angle seen from the front of the dipole. Units are in degrees.

Data Types: `double`

Output Arguments

pat — Field pattern

3-D array

Field pattern, returned as a 2-by-1-by-N_{AZ} array representing the vertical and horizontal field pattern, where N_{AZ} is the number of elements in the `az` input vector.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

`winner2.AntennaArray` | `winner2.layoutparset`

Introduced in R2017a

winner2.layoutparset

WINNER II layout parameter configuration

Syntax

```
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays)
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax)
cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax,seed)
```

Description

Download Required: To use this function, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays)` returns a structure of randomly generated WINNER II network layout parameters given mobile station (MS) indices, base station (BS) indices, BS to MS links, and antenna array configurations.

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax)` additionally specifies the maximum layout range used when generating MS and BS positions.

`cfgLayout = winner2.layoutparset(msIdx,bsIdx,K,arrays,rmax,seed)` additionally specifies a seed value for repeatability. To assign `seed` when not assigning `rmax`, specify `rmax` as [ ].

Examples

Create Two MS to One BS WINNER II System Layout

Create a WINNER II system layout with two mobile stations (MS) connecting to the same base station (BS).

Define antenna arrays for one BS and two MS.

```
BSAA  = winner2.AntennaArray('UCA', 8, 0.02);  % UCA-8 array for BS
MSAA1 = winner2.AntennaArray('ULA', 2, 0.01);  % ULA-2 array for MS
MSAA2 = winner2.AntennaArray('ULA', 4, 0.005); % ULA-4 array for MS
```

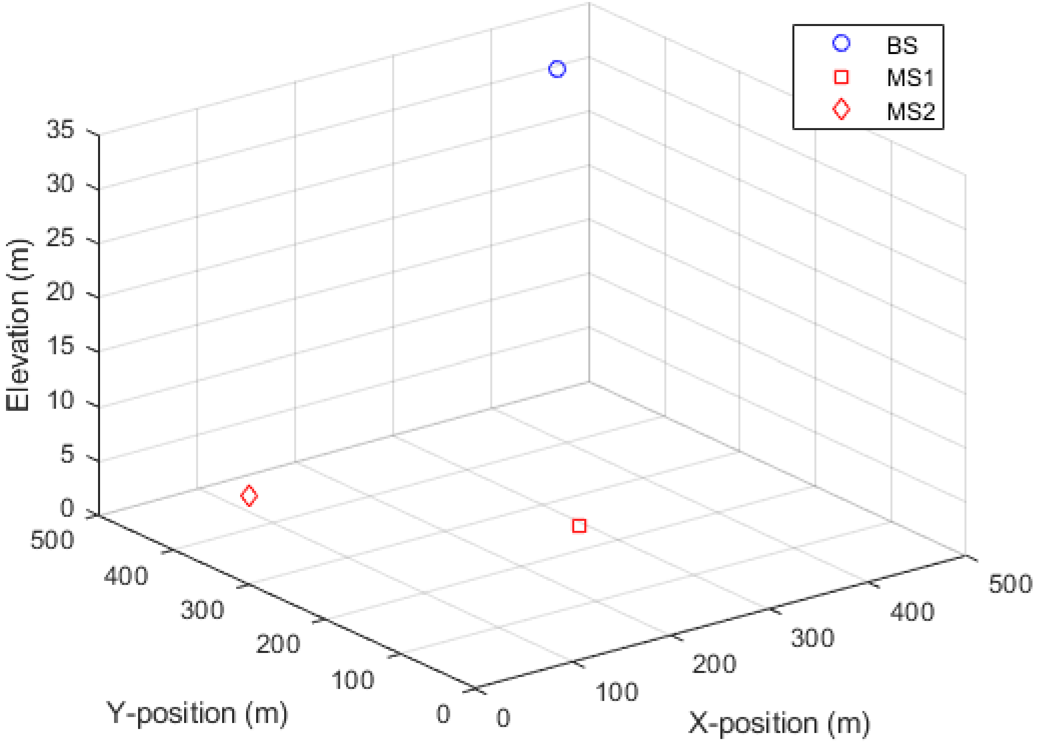
Create system layout by using the `winner2.layoutparset` function.

```
MSIdx = [2 3];
BSIdx = {1};
K = 2;
rndSeed = 5;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx, ...
    K,[BSAA,MSAA1,MSAA2],[],rndSeed);
```

Visualize BS and MS positions.

```
BSPos  = cfgLayout.Stations(cfgLayout.Pairing(1,1)).Pos;
MS1Pos = cfgLayout.Stations(cfgLayout.Pairing(2,1)).Pos;
```

```
MS2Pos = cfgLayout.Stations(cfgLayout.Pairing(2,2)).Pos;

plot3(BSPos(1),BSPos(2),BSPos(3),'bo', ...
    MS1Pos(1),MS1Pos(2),MS1Pos(3),'rs', ...
    MS2Pos(1),MS2Pos(2),MS2Pos(3),'rd');
grid on;
xlim([0 500]);
ylim([0 500]);
zlim([0 35]);
xlabel('X-position (m)');
ylabel('Y-position (m)');
zlabel('Elevation (m)');
legend('BS','MS1','MS2','Location','northeast');
```



Input Arguments

msIdx — Mobile station index

row vector

Mobile station index, specified as a row vector indicating the indices in `arrays` to serve as mobile stations.

Data Types: `double`

bsIdx — Base station index

column cell array

Base station index, specified as a column cell array, with each element representing one base station. Each cell element is an integer-valued row vector to indicate the indices in `arrays` to serve as different sectors of that base station.

Data Types: `double`

`K` — Number of links
scalar

Number of links, specified as a scalar representing the number of BS-MS links to be formulated.

Data Types: `double`

`arrays` — Antenna array configurations
vector of structures

Antenna array configurations, specified as a vector of structures defining all available arrays. All MS and BS sectors are chosen from this vector. The array of elements is typically created using the `winner2.AntennaArray` function.

Data Types: `double`

`rmax` — Maximum layout range
`500` (default) | scalar

Maximum layout range, specified as a scalar representing the maximum layout range in meters used to randomly generate the MS and BS positions.

Data Types: `double`

`seed` — Seed value
integer

Seed value used to provide repeatability, specified as an integer. When `seed` is not specified, the global random number generator is used. To assign `seed` when not assigning `rmax`, specify `rmax` as `[]`.

Data Types: `double`

Output Arguments

`cfgLayout` — Configuration layout
structure

Configuration layout, returned as a structure containing these fields, which represent the location and orientation parameters for all simulated stations.

`Stations` — Active stations
row vector of structures

Active stations, returned as a row vector of structures describing the antenna arrays for active stations. `Stations` is created from the `arrays` input and adds an additional `Velocity` field. The row ordering specifies base station (BS) sectors first, followed by the mobile stations (MS). The BS sector and MS positions are randomly assigned. The BS sectors have no velocity. Each MS has a velocity of about 1.42 m/s with a randomly assigned direction.

NofSect — Number of sectors
vector

Number of sectors, returned as a vector indicating the number of sectors in each BS.

Pairing — BS to MS pairing
matrix

BS to MS pairing, returned as a 2-by-N_L matrix, where N_L specifies the number of links to be modeled. See Stations for BS and MS row ordering.

ScenarioVector — Spatial scenario
1 (default) | vector

Spatial scenario, returned as a 1-by-N_L vector of scenario numbers. The default is 1, which specifies scenario A1.

The scenarios numbers map as {1=A1, 2=A2, 3=B1, 4=B2, 5=B3, 6=B4, 10=C1, 11=C2, 12=C3, 13=C4, 14=D1, 15=D2a}.

For more information, see WINNER II Channel Models [1], Section 2.3.

PropagConditionVector — Propagation condition
1 (default) | vector

Propagation condition, returned as a 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link. The default is 1.

StreetWidth — Street width
20 (default) | vector

Street width, returned as a 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. StreetWidth is used for the path loss model of the B1 and B2 scenarios. See ScenarioVector for the scenario number mapping. All elements must have the same value. StreetWidth applies only when the PathLossModelUsed field from winner2.wimparset is set to 'yes'.

Dist1 — Distances from BS to the last LOS point
NaN (default) | vector

Distances from BS to the last LOS point, returned as a 1-by-N_L vector. Dist1 is used for the path loss model of the B1 and B2 scenarios. The default value of NaN indicates that the distance is randomly determined in path loss function. See ScenarioVector for the scenario number mapping. Dist1 applies only when the PathLossModelUsed field from winner2.wimparset is set to 'yes'.

For more information, see WINNER II Channel Models [1], Figure 4-3.

NumFloors — Floor numbers
1 (default) | vector

Floor numbers, returned as a 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The NumFloors property is used for the path loss model of the A2 and B4 scenarios only. See ScenarioVector for the scenario number mapping. NumFloors applies only when the PathLossModelUsed field from winner2.wimparset is set to 'yes'.

`NumPenetratedFloors` — Number of floors penetrated

`0` (default) | vector

Number of floors penetrated, returned as a 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The `NumPenetratedFloors` property is used for the NLOS path loss model of the A1 scenario. See `ScenarioVector` for the scenario number mapping. `NumPenetratedFloors` applies only when the `PathLossModelUsed` field from `winner2.wimparset` is set to `'yes'`.

For more information, see WINNER II Channel Models [1], Table 4-4.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
`comm.WINNER2Channel`

Functions
`winner2.AntennaArray` | `winner2.wim` | `winner2.wimparset`

Introduced in R2017a

winner2.wim

Generate channel coefficients using WINNER II channel model

Syntax

```
chanCoef = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays] = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout)
[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout,initCond)
```

Description

Download Required: To use this function, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

`chanCoef = winner2.wim(cfgWim,cfgLayout)` returns channel coefficients based on the WINNER II model parameters for all links defined in the WINNER II network layout.

`[chanCoef,pathDelays] = winner2.wim(cfgWim,cfgLayout)` also returns the path delays for all links.

`[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout)` also returns the final condition of the system after generating the channel coefficients.

`[chanCoef,pathDelays,finalCond] = winner2.wim(cfgWim,cfgLayout,initCond)` generates the channel coefficients by using the initial system conditions rather than of performing random initialization. `initCond` is of the same form as `finalCond` and is typically the `finalCond` output from the prior call of this function. Use this syntax to repeatedly generate channel coefficients for continuous time samples.

Examples

Continuously Generate WINNER II Channel Coefficients

Continuously generate channel coefficients for each link in a two-link system layout.

Configure model parameters.

```
cfgWim = winner2.wimparset;
cfgWim.SampleDensity = 20;
cfgWim.RandomSeed = 10; % For repeatability
```

Configure layout parameters.

```
BSAA  = winner2.AntennaArray('UCA',8,0.02);  % UCA-8 array for BS
MSAA1 = winner2.AntennaArray('ULA',2,0.01);  % ULA-2 array for MS1
MSAA2 = winner2.AntennaArray('ULA',4,0.005); % ULA-4 array for MS2
MSIdx = [2,3];
```

```
BSIdx = {1};
NL = 2;
rndSeed = 5;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx,NL,[BSAA,MSAA1,MSAA2],[],rndSeed);
```

Generate channel coefficients for the first time.

```
[H1,~,finalCond] = winner2.wim(cfgWim,cfgLayout);
```

Generate a second set of channel coefficients.

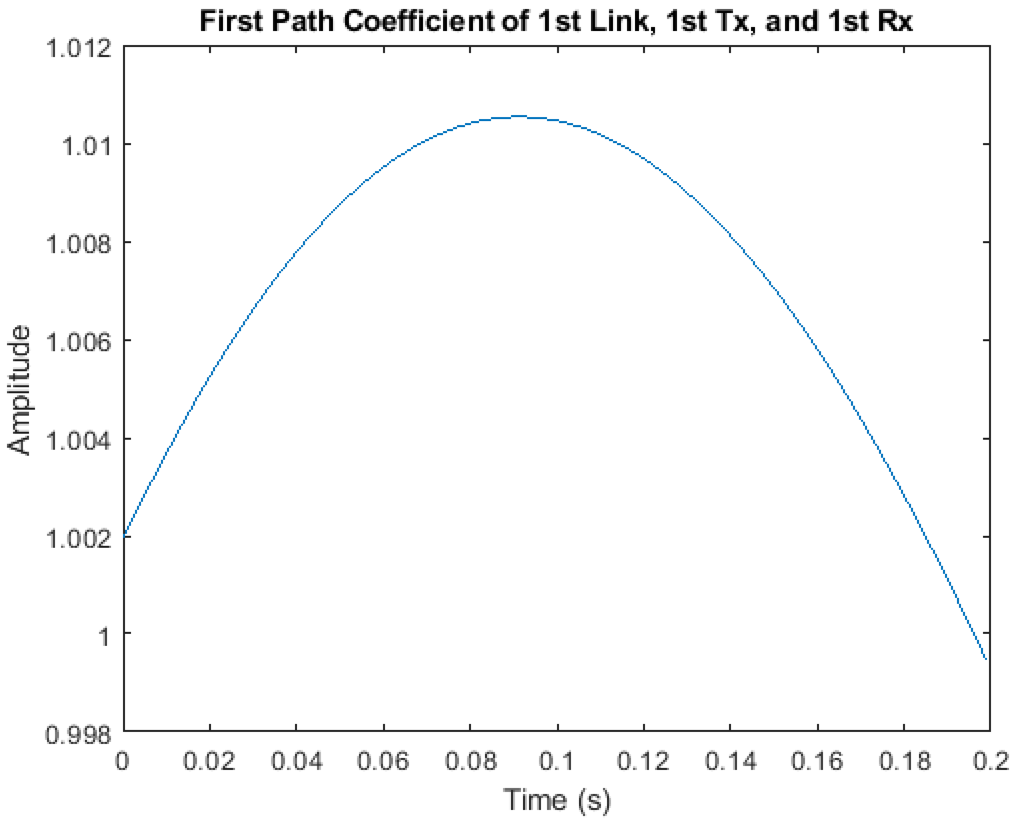
```
[H2,~,finalCond] = winner2.wim(cfgWim,cfgLayout,finalCond);
```

Concatenate H1 and H2 in time domain.

```
H = cellfun(@(x,y) cat(4,x,y),H1,H2,'UniformOutput',false);
```

Plot H for the first link, 1st Tx, 1st Rx, and 1st path. The plot shows the channel continuity over the two outputs from the `winner2.wim` function.

```
figure;
Ts = finalCond.delta_t(1);  % Sample time for the 1st link
plot(Ts*(0:2*cfgWim.NumTimeSamples-1)', ...
    abs(squeeze(H{1}(1,1,1,:))));
xlabel('Time (s)');
ylabel('Amplitude');
title('First Path Coefficient of 1st Link, 1st Tx, and 1st Rx');
```



Input Arguments

`cfgWim` — Configuration layout

structure

Configuration model, specified as a structure containing these fields. `cfgWim` is typically created using the `winner2.wimparset` function.

`NumTimeSamples` — Number of time samples

`100` (default) | scalar

Number of time samples, specified as a scalar.

`FixedPdpUsed` — Use predefined path delays and powers for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path delays and powers for specific scenarios, specified as `'no'` or `'yes'`.

`FixedAnglesUsed` — Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios, specified as `'yes'` or `'no'`.

`IntraClusterDsUsed` — Divide each of the two strongest clusters into three subclusters per link

`'yes'` (default) | `'no'`

Divide each of the two strongest clusters into three subclusters per link, specified as `'yes'` or `'no'`.

`PolarisedArrays` — Use dual-polarized arrays

`'yes'` (default) | `'no'`

Use dual-polarized arrays, specified as `'yes'` or `'no'`.

`UseManualPropCondition` — Use manually defined propagation conditions

`'yes'` (default) | `'no'`

Use manually defined propagation conditions, specified as `'yes'` or `'no'`. Set to `'yes'` to enforce the use of manually defined propagation conditions (LOS/NLOS) in the `PropagConditionVector` structure field returned by `winner2.layoutparset`. Set to `'no'` to draw propagation conditions from pre-defined LOS probabilities.

`CenterFrequency` — Carrier frequency

`5.25e9` (default) | scalar

Carrier frequency in Hz, specified as a scalar.

`UniformTimeSampling` — Enforce uniform time sampling

`'no'` (default) | `'yes'`

Enforce all links to be sampled at the same time instants, specified as `'no'` or `'yes'`.

`SampleDensity` — Number of time samples per half wavelength

`2e6` (default) | scalar

Number of time samples per half wavelength, specified as a scalar.

`DelaySamplingInterval` — Sampling interval

`5e-9` (default) | scalar

Sampling interval, specified as an scalar indicating the input signal sample time in seconds. `DelaySamplingInterval` defines the sampling grid to which the path delays are rounded. A value of `0` seconds indicates no rounding on path delays.

`ShadowingModelUsed` — Use shadow fading

`'no'` (default) | `'yes'`

Use shadow fading, specified as `'no'` or `'yes'`.

`PathLossModelUsed` — Use path loss model

`'no'` (default) | `'yes'`

Use path loss model, specified as `'no'` or `'yes'`.

`PathLossModel` — Path loss model

`'pathloss'` (default) | character vector

Path loss model, specified as a character vector representing a valid function name. `PathLossModel` applies only when `PathLossModelUsed` is set to `'yes'`.

`PathLossOption` — Wall material

`'CR_light'` (default) | `'CR_heavy'` | `'RR_light'` | `'RR_heavy'`

Wall material, specified as `'CR_light'`, `'CR_heavy'`, `'RR_light'`, or `'RR_heavy'`, indicating the wall material for the *A1* scenario NLOS path loss calculation. `PathLossOption` applies only when `PathLossModelUsed` is set to `'yes'`.

`RandomSeed` — Seed for random number generators

`[]` (default) | scalar

Seed for random number generators, specified as a scalar or empty brackets. Empty brackets, `[]`, indicate that the global random stream is used.

`cfgLayout` — Configuration layout

structure

Configuration layout, specified as a structure containing these fields, which represent the location and orientation parameters for all simulated stations. `cfgLayout` is typically created using the `winner2.layoutparset` function.

`Stations` — Active stations

row vector of structures

Active stations, specified as a row vector of structures describing the antenna arrays for active stations. `Stations` is created from the `arrays` input of `winner2.layoutparset` and adds an additional `Velocity` field. The row ordering specifies base station (BS) sectors first, followed by the mobile stations (MS). The BS sector and MS positions are randomly assigned. The BS sectors have no velocity. Each MS has a velocity of about 1.42 m/s with a randomly assigned direction.

`NofSect` — Number of sectors

vector

Number of sectors, specified as a vector indicating the number of sectors in each BS.

`Pairing` — BS to MS pairing

matrix

BS to MS pairing, specified as a 2-by-N_L matrix, where N_L specifies the number of links to be modeled. See `Stations` for BS and MS row ordering.

`ScenarioVector` — Spatial scenario

`1` (default) | vector

Spatial scenario, specified as a 1-by-N_L vector of scenario numbers. The default is `1`, which specifies scenario A1.

The scenarios numbers map as `{1=`A1, `2=`A2, `3=`B1, `4=`B2, `5=`B3, `6=`B4, `10=`C1, `11=`C2, `12=`C3, `13=`C4, `14=`D1, `15=`D2a`}`.

For more information, see WINNER II Channel Models [1], Section 2.3.

`PropagConditionVector` — Propagation condition

`1` (default) | vector

Propagation condition, specified as a 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link.

`StreetWidth` — Street width

`20` (default) | vector

Street width, specified as a 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. `StreetWidth` is used for the path loss model of the B1 and B2 scenarios. See `ScenarioVector` for the scenario number mapping. All elements must have the same value. `StreetWidth` applies only when `cfgWim.PathLossModelUsed` is set to `'yes'`.

`Dist1` — Distances from BS to the last LOS point

`NaN` (default) | vector

Distances from BS to the last LOS point, specified as a 1-by-N_L vector. `Dist1` is used for the path loss model of the B1 and B2 scenarios. The default value of `NaN` indicates that the distance is randomly determined in path loss function. See `ScenarioVector` for the scenario number mapping. `Dist1` applies only when `cfgWim.PathLossModelUsed` is set to `'yes'`.

For more information, see WINNER II Channel Models [1], Figure 4-3.

`NumFloors` — Floor numbers

`1` (default) | vector

Floor numbers, specified as a 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The default value is `1`. The `NumFloors` field is used for the path loss model of the A2 and B4 scenarios only. See `ScenarioVector` for the scenario number mapping. `NumFloors` applies only when `cfgWim.PathLossModelUsed` is set to `'yes'`.

`NumPenetratedFloors` — Number of floors penetrated

0 (default) | vector

Number of floors penetrated, specified as a 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The default value is `0`. The `NumPenetratedFloors` is used for the NLOS path loss model of the A1 scenario. See ScenarioVector for the scenario number mapping. NumPenetratedFloors field applies only when cfgWim.PathLossModelUsed is set to 'yes'.

For more information, see WINNER II Channel Models [1], Table 4-4.

initCond — Initial system condition
structure | optional

Initial system condition, specified as a structure. initCond is of the same form as finalCond and is typically the finalCond output from the prior call of winner2.wim.

Data Types: struct

Output Arguments

chanCoef — Channel coefficients
cell array containing 4-D arrays of complex values

Channel coefficients, returned as an N_L-by-1 cell array. N_L is the number of links in the system. The ith element of chanCoef is an $N_R(i)$-by-$N_T(i)$-by-$N_P(i)$-by-N_S array. N_R, N_T, and N_P are link specific. N_S is the same for all the links.

- $N_R(i)$ is the number of receive antenna elements at MS for the ith link.
- $N_T(i)$ is the number of transmit antenna elements at BS for the ith link.
- $N_P(i)$ is the number of paths for the ith link.
- N_S is the number of time samples given by cfgWim.NumTimeSamples.

For more information , see “Channel Power” on page 2-874.

Data Types: cell

pathDelays — Path delays
matrix

Path delays, returned as an N_L-by-$maxN_P$ matrix. N_L is the number of links in the system and $maxN_P$ is the maximum number of paths among all links. Each row of the matrix applies to each link. When a link has fewer than $maxN_P$ paths, the corresponding row in pathDelays is NaN padded.

Data Types: double

finalCond — Final system condition
structure

Final system condition, returned as a structure. When generating channel coefficients for continuous time samples, use finalCond as the initCond input for the next call to winner2.wim.

For more information, see WINNER II Channel Models [1], Section 5.2.

Data Types: struct

More About

Channel Power

When path loss and shadowing are off, path gains of the computed WINNER channel are normalized. Specifically, path gains are normalized when the `ShadowingModelUsed` and `PathLossModelUsed` parameters are set to `'no'`.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
`comm.WINNER2Channel`

Functions
`winner2.AntennaArray` | `winner2.layoutparset` | `winner2.wimparset`

Introduced in R2017a

winner2.wimparset

WINNER II model parameter configuration

Syntax

```
cfgWim = winner2.wimparset
```

Description

Download Required: To use this function, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

`cfgWim = winner2.wimparset` returns a structure of WINNER II model parameters with their default values.

Examples

Create a WINNER II model parameter set

Use the `winner2.wimparset` function to create a WINNER II model parameter set.

```
cfgwim = winner2.wimparset;
```

Adjust default settings.

```
cfgwim.RandomSeed = 31; % Set the rng seed for repeatability
cfgwim.NumTimeSamples = 250;
cfgwim.CenterFrequency = 4e9;
```

Display the WINNER II model parameter settings.

```
cfgwim
```

```
cfgwim = struct with fields:
            NumTimeSamples: 250
              FixedPdpUsed: 'no'
           FixedAnglesUsed: 'no'
        IntraClusterDsUsed: 'yes'
           PolarisedArrays: 'yes'
    UseManualPropCondition: 'yes'
           CenterFrequency: 4.0000e+09
        UniformTimeSampling: 'no'
             SampleDensity: 2000000
     DelaySamplingInterval: 5.0000e-09
        ShadowingModelUsed: 'no'
         PathLossModelUsed: 'no'
             PathLossModel: 'pathloss'
            PathLossOption: 'CR_light'
                RandomSeed: 31
```

Output Arguments

`cfgWim` — Configuration layout

structure

Configuration model, returned as a structure containing these fields.

`NumTimeSamples` — Number of time samples

`100` (default) | scalar

Number of time samples, specified as a scalar.

`FixedPdpUsed` — Use predefined path delays and powers for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path delays and powers for specific scenarios, specified as `'no'` or `'yes'`.

`FixedAnglesUsed` — Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios

`'no'` (default) | `'yes'`

Use predefined path angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios, specified as `'yes'` or `'no'`.

`IntraClusterDsUsed` — Divide each of the two strongest clusters into three subclusters per link

`'yes'` (default) | `'no'`

Divide each of the two strongest clusters into three subclusters per link, specified as `'yes'` or `'no'`.

`PolarisedArrays` — Use dual-polarized arrays

`'yes'` (default) | `'no'`

Use dual-polarized arrays, specified as `'yes'` or `'no'`.

`UseManualPropCondition` — Use manually defined propagation conditions

`'yes'` (default) | `'no'`

Use manually defined propagation conditions, specified as `'yes'` or `'no'`. Set to `'yes'` to enforce the use of manually defined propagation conditions (LOS/NLOS) in the `PropagConditionVector` structure field returned by `winner2.layoutparset`. Set to `'no'` to draw propagation conditions from pre-defined LOS probabilities.

`CenterFrequency` — Carrier frequency

`5.25e9` (default) | scalar

Carrier frequency in Hz, specified as a scalar.

`UniformTimeSampling` — Enforce uniform time sampling

`'no'` (default) | `'yes'`

Enforce all links to be sampled at the same time instants, specified as `'no'` or `'yes'`.

SampleDensity — Number of time samples per half wavelength

2e6 (default) | scalar

Number of time samples per half wavelength, specified as a scalar.

DelaySamplingInterval — Sampling interval

5e-9 (default) | scalar

Sampling interval, specified as an scalar indicating the input signal sample time in seconds. DelaySamplingInterval defines the sampling grid to which the path delays are rounded. A value of 0 seconds indicates no rounding on path delays.

ShadowingModelUsed — Use shadow fading

'no' (default) | 'yes'

Use shadow fading, specified as 'no' or 'yes'.

PathLossModelUsed — Use path loss model

'no' (default) | 'yes'

Use path loss model, specified as 'no' or 'yes'.

PathLossModel — Path loss model

'pathloss' (default) | character vector

Path loss model, specified as a character vector representing a valid function name. PathLossModel applies only when PathLossModelUsed is set to 'yes'.

PathLossOption — Wall material

'CR_light' (default) | 'CR_heavy' | 'RR_light' | 'RR_heavy'

Wall material, specified as 'CR_light', 'CR_heavy', 'RR_light', or 'RR_heavy', indicating the wall material for the *A1* scenario NLOS path loss calculation. PathLossOption applies only when PathLossModelUsed is set to 'yes'.

RandomSeed — Seed for random number generators

[] (default) | scalar

Seed for random number generators, specified as a scalar or empty brackets. Empty brackets, [], indicate that the global random stream is used.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4-027756 WINNER II, September 2007.

See Also

Objects
comm.WINNER2Channel

Functions
winner2.layoutparset | winner2.wim

Introduced in R2017a

zadoffChuSeq

Generate root Zadoff-Chu sequence

Syntax

```
seq = zadoffChuSeq(R,N)
```

Description

`seq = zadoffChuSeq(R,N)` generates the `R`th root Zadoff-Chu sequence with length `N`, as defined in 3GPP TS 36.211.

The function generates the sequence using the algorithm given by

seq(m+1) = exp(-j·π·R·m·(m+1)/N), for *m = 0, ..., N-1*.

The function uses a negative polarity on the argument of the exponent, that is, a clockwise sequence of phases.

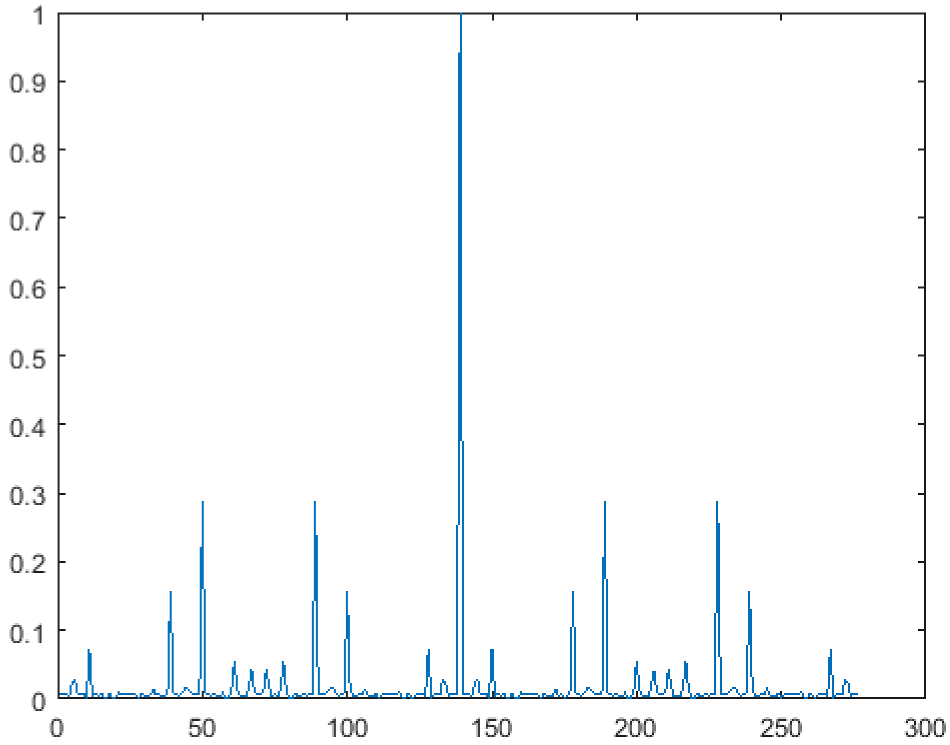
Examples

Examine Correlation Properties of Root Zadoff-Chu Sequence

Generate the 25th root Zadoff-Chu sequence with a length of 139.

Plot the absolute values of the output sequence.

```
seq = zadoffChuSeq(25,139);
plot(abs(xcorr(seq)./length(seq)))
```



Input Arguments

R — Root of Zadoff-Chu sequence

positive integer

Root of the Zadoff-Chu sequence, specified as a positive integer.

Example: 25

Data Types: double

N — Length of Zadoff-Chu sequence

odd positive integer

Length of the Zadoff-Chu sequence, specified as an odd positive integer.

Example: 139

Data Types: double

Output Arguments

seq — Rth root Zadoff-Chu sequence

column vector of complex values

Rth root Zadoff-Chu sequence, returned as an N-by-1 vector of complex values.

Compatibility Considerations

`lteZadoffChuSeq` was renamed to `zadoffChuSeq`

Behavior changed in R2019a

In release R2019a, the `lteZadoffChuSeq` function was renamed to `zadoffChuSeq`.

References

[1] 3GPP TS 36.211. "Evolved Universal Terrestrial Radio Access (E-UTRA); Physical Channels and Modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*..

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`comm.GoldSequence` | `comm.PNSequence`

Introduced in R2012b

bleAngleEstimate

Estimate AoA or AoD of BLE Signal

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

`angle = bleAngleEstimate(IQsamples,cfg)`

Description

`angle = bleAngleEstimate(IQsamples,cfg)` estimates the angle of arrival (AoA) or angle of departure (AoD), `angle`, for the given in-phase and quadrature (IQ) samples, `IQsamples`, and Bluetooth low energy (BLE) angle estimation configuration object, `cfg`.

Examples

Estimate AoA of the BLE Signal

Create a BLE angle estimation configuration object, specifying the values of antenna array size, slot duration, and antenna switching pattern.

```
cfg = bleAngleEstimateConfig('ArraySize',2,'SlotDuration',2,...
    'SwitchingPattern',[1 2])
```

```
cfg = 
  bleAngleEstimateConfig with properties:

           ArraySize: 2
      ElementSpacing: 0.5000
        SlotDuration: 2
    SwitchingPattern: [1 2]
```

Set the IQ samples such that they define a connection data channel protocol data unit (PDU) with an AoA constant tone extension (CTE) of 2 *μs* slots, CTE time of 16 *μs*, and azimuth rotation of 70 degrees.

```
IQsamples = [0.8507+0.5257i;-0.5257 + 0.8507i;-0.8507 - 0.5257i;...
    0.5257 - 0.8507i;0.8507+0.5257i;-0.5257 + 0.8507i;...
    -0.8507 - 0.5257i;0.5257 - 0.8507i;-0.3561 + 0.9345i];
```

Estimate the AoA of the BLE signal.

```
angle = bleAngleEstimate(IQsamples,cfg)
```

```
angle = 70
```

Input Arguments

`IQsamples` — IQ samples

complex-valued column vector

IQ samples, specified as a complex-valued column vector. This argument corresponds to the 8 µs value of the reference period and slot duration.

Data Types: `single` | `double`

`cfg` — BLE angle estimation configuration object

`bleAngleEstimateConfig` object

BLE angle estimation configuration object, specified as a `bleAngleEstimateConfig` object.

Output Arguments

`angle` — AoA or AoD

real number | two-element row vector of real numbers

AoA or AoD, returned as one of these values.

- Real number – This value is the estimated broadside angle. If *elevation* is 0, the estimated broadside angle represents the azimuth angle.
- Two-element row vector of real numbers in the form [*azimuth elevation*] – *azimuth* and *elevation* are the estimated azimuth angle and elevation angle in degrees, respectively.

The size of this output argument is equal to the size of the “ArraySize” on page 3-0 property of the `bleAngleEstimateConfig` object.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Wooley, Martin. *Bluetooth Direction Finding: A Technical Overview*. Bluetooth Special Interest Group (SIG), Accessed April 6, 2020, https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleIdealReceiver` | `bleWaveformGenerator` | `getElementPosition` | `getNumElements`

Objects
`bleAngleEstimateConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”
“Bluetooth Location and Direction Finding”

Introduced in R2020b

bleATTPDU

Generate BLE ATT PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
attPDU = bleATTPDU(cfgATT)
```

Description

`attPDU = bleATTPDU(cfgATT)` generates a Bluetooth low energy (BLE) attribute protocol data unit (ATT PDU) corresponding to the BLE ATT PDU configuration object `cfgATT`.

Examples

Generate BLE ATT PDUs

Generate two unique BLE ATT PDUs of type `'Read by type request'` and `'Error response'`.

Create a BLE ATT PDU configuration object with default settings.

```
cfgATT = bleATTPDUConfig;
```

Change the BLE ATT PDU opcode as `'Read by type request'`. View the applicable properties of the opcode `'Read by type request'`.

```
cfgATT.Opcode = 'Read by type request'
```

```
cfgATT = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

Generate a BLE ATT PDU using the corresponding configuration object.

```
attPDU = bleATTPDU(cfgATT)
```

```
attPDU = 7x2 char array
    '08'
    '01'
    '00'
    'FF'
    'FF'
    '00'
    '28'
```

Create another BLE ATT PDU configuration object, this time using the name-value pairs. Change the BLE ATT PDU opcode to `'Error response'`. View the applicable properties of the opcode `'Error response'`.

```
cfgATT = bleATTPDUConfig('Opcode','Error response')
```

```
cfgATT = 
  bleATTPDUConfig with properties:

             Opcode: 'Error response'
    RequestedOpcode: 'Read request'
    AttributeHandle: '0001'
       ErrorMessage: 'Invalid handle'
```

Generate a BLE ATT PDU corresponding to this configuration object.

```
attPDU = bleATTPDU(cfgATT)
```

```
attPDU = 5x2 char array
    '01'
    '0A'
    '01'
    '00'
    '01'
```

Input Arguments

`cfgATT` — BLE ATT PDU configuration object

`bleATTPDUConfig` object

BLE ATT PDU configuration object, specified as a `bleATTPDUConfig` object. This value defines the type of BLE ATT PDU and its applicable properties.

Output Arguments

`attPDU` — Generated BLE ATT PDU

character array

Generated BLE ATT PDU, returned as a character array. Each row in this array is the hexadecimal representation of an octet.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleATTPDUDecode`

Objects

`bleATTPDUConfig`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleATTPDUDecode

Decode BLE ATT PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[status,cfgATT] = bleATTPDUDecode(attPDU)
```

Description

`[status,cfgATT] = bleATTPDUDecode(attPDU)` decodes the specified Bluetooth low energy (BLE) attribute protocol data unit (ATT PDU), returning the corresponding BLE ATT PDU configuration object, `cfgATT`, and the decoding status, `status`.

Examples

Decode BLE ATT PDUs

Decode two unique BLE ATT PDUs of type `'Read by type request'` and `'Error response'`.

Create a BLE ATT PDU configuration object with default settings.

```
cfgATT = bleATTPDUConfig;
```

Change the BLE ATT PDU opcode as `'Read by type request'`. View the applicable properties of the opcode `'Read by type request'`.

```
cfgATT.Opcode = 'Read by type request'
```

```
cfgATT = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

Generate a BLE ATT PDU using the corresponding configuration object.

```
attPDU = bleATTPDU(cfgATT)
```

```
attPDU = 7x2 char array
    '08'
    '01'
    '00'
    'FF'
    'FF'
    '00'
    '28'
```

Decode the generated BLE ATT PDU. The returned status indicates decoding is successful. View the applicable properties of the opcode `'Read by type request'`.

```
[status, cfg] = bleATTPDUDecode(attPDU)
```

```
status = 
Success

cfg = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

Create another BLE ATT PDU configuration object, this time using the name-value pairs. Change the BLE ATT PDU opcode as `'Error response'`. View the applicable properties of the opcode `'Error response'`.

```
cfgATT = bleATTPDUConfig('Opcode','Error response')
```

```
cfgATT = 
  bleATTPDUConfig with properties:

             Opcode: 'Error response'
    RequestedOpcode: 'Read request'
    AttributeHandle: '0001'
       ErrorMessage: 'Invalid handle'
```

Generate a BLE ATT PDU using the corresponding configuration object.

```
attPDU = bleATTPDU(cfgATT)
```

```
attPDU = 5x2 char array
    '01'
    '0A'
    '01'
    '00'
    '01'
```

Decode the generated BLE ATT PDU. The returned status indicates decoding is successful. View the applicable properties of the opcode `'Error response'`.

```
[status, cfg] = bleATTPDUDecode(attPDU)
```

```
status = 
Success

cfg = 
  bleATTPDUConfig with properties:
```

```
             Opcode: 'Error response'
    RequestedOpcode: 'Read request'
    AttributeHandle: '0001'
       ErrorMessage: 'Invalid handle'
```

Decode Corrupted BLE ATT PDU

Specify a BLE ATT PDU containing corrupted data values.

```
attPDU = ['09'; '03'; '01'; '00'; '18'; '0D']; % Sample corrupted BLE ATT PDU
```

Decode the specified BLE ATT PDU. The returned status indicates that the decoding failed due to mismatched attribute data lengths. In case of failed decoding, BLE ATT PDU configuration object, `'cfgATT'`, displays no properties.

```
[status, cfgATT] = bleATTPDUDecode(attPDU)
```

```
status = 
MismatchAttributeDataLength

cfgATT = 
  bleATTPDUConfig with properties:
```

Input Arguments

`attPDU` — BLE ATT PDU

character vector | string scalar | numeric vector | character array

BLE ATT PDU, specified as one of these values:

- Character vector — This vector represent octets in hexadecimal format.
- String scalar — This scalar represent octets in hexadecimal format.
- Numeric vector of elements in the range [0,255] — This vector represent octets in decimal format.
- n-by-2 character array — Each row represent an octet in hexadecimal format.

Data Types: `char` | `uint8` | `uint16` | `uint32` | `double` | `string`

Output Arguments

`cfgATT` — BLE ATT PDU configuration object

`bleATTPDUConfig` object

BLE ATT PDU configuration object, returned as a `bleATTPDUConfig` object. This value denotes the decoded BLE ATT PDU configuration.

`status` — Packet decoding status

nonpositive integer

Packet decoding status, returned as a nonpositive number of type `blePacketDecodeStatus`. This value represents the result of an ATT PDU decoding. Each value of `status` corresponds to a member of the `blePacketDecodeStatus` enumeration class, which indicates the packet decoding status according to this table.

Enumeration Value	Member of Enumeration Class	Decoding Status
0	`Success`	Packet decoding succeeded
–401	`UnsupportedATTOpcode`	Invalid ATT opcode
–402	`IncompleteATTPDU`	Incomplete ATT PDU
–403	`InvalidATTReqOpcodeInErrorResp`	Invalid requested opcode in "Error Response" PDUs
–404	`InvalidATTErrorCode`	Invalid error code
–405	`InvalidATTRxMTU`	Invalid received MTU
–406	`InvalidAttributeHandleRange`	Invalid attribute handle range
–407	`InvalidAttributeType`	Invalid attribute type flag
–408	`InvalidATTExecuteWriteFlag`	Invalid execute write flag
–409	`MismatchAttributeDataLength`	Length mismatches with actual length
–410	`InvalidATTDataFormat`	Invalid Data Format

An enumeration value other than `0` means that the BLE ATT PDU decoding failed. If the decoding fails, object `cfgATT` displays no output.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleATTPDU`

Objects

`bleATTPDUConfig`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleGAPDataBlock

Generate BLE GAP data block

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
dataBlock = bleGAPDataBlock(cfgGAP)
```

Description

`dataBlock = bleGAPDataBlock(cfgGAP)` generates a Bluetooth low energy (BLE) generic access profile (GAP) data block of type advertising data (AD) or scan response data (SRD) corresponding to the BLE GAP data block configuration object `cfgGAP`.

Examples

Generate BLE GAP AD Blocks

Generate three unique BLE GAP AD blocks: first one with AD types `'Flags'` and `'Tx power level'`, the second one with AD types `'Advertising interval'` and `'Local name'` and the third one with AD type `'Flags'` and having simultaneous support for low energy (LE) and basic rate/enhanced data rate (BR/EDR) at the host.

Create a configuration object for a BLE GAP AD block and specify the AD types as `'Flags'` and `'Tx power level'`. Assign the values of LED discoverability as `'Limited'` and Tx power level as `45`.

```
cfgGAP = bleGAPDataBlockConfig;
cfgGAP.AdvertisingDataTypes   = {'Flags';'Tx power level'};
cfgGAP.LEDiscoverability      = 'Limited';
cfgGAP.TxPowerLevel           = 45;
```

Generate the BLE GAP AD block from the corresponding configuration object.

```
dataBlock = bleGAPDataBlock(cfgGAP)
```

```
dataBlock = 6x2 char array
    '02'
    '01'
    '05'
    '02'
    '0A'
    '2D'
```

Create a configuration object for a BLE GAP AD block, this time with advertising data types as `'Advertising interval'` and `'Local name'`. Specify the values of the advertising interval as `48`, the local name as `'MathWorks'` and the local name shortening as `true`.

```
cfgGAP = bleGAPDataBlockConfig('AdvertisingDataTypes', ...
    {'Advertising interval', ...
    'Local name'});
cfgGAP.AdvertisingInterval  = 48;
cfgGAP.LocalName            = 'MathWorks';
cfgGAP.LocalNameShortening  = true;
```

Generate the BLE GAP AD block from the corresponding configuration object.

```
dataBlock = bleGAPDataBlock(cfgGAP)
```

```
dataBlock = 15x2 char array
    '03'
    '1A'
    '30'
    '00'
    '0A'
    '08'
    '4D'
    '61'
    '74'
    '68'
    '57'
    '6F'
    '72'
    '6B'
    '73'
```

Create a configuration object for a BLE GAP AD block with type `'Flags'`. Specify the values of LE discoverability as `'Limited'`, BR/EDR support as `true`, and simultaneous support for LE and BR/EDR as `'Host'`.

```
cfgGAP = bleGAPDataBlockConfig;
cfgGAP.LEDiscoverability      = 'Limited';
cfgGAP.BREDR                  = true;
cfgGAP.LE                     = 'Host';
```

Generate the BLE GAP AD block from the corresponding configuration object.

```
dataBlock = bleGAPDataBlock(cfgGAP)
```

```
dataBlock = 3x2 char array
    '02'
    '01'
    '11'
```

Input Arguments

`cfgGAP` — BLE GAP data block configuration object

`bleGAPDataBlockConfig` (default) | object

BLE GAP data block configuration object, specified as a `bleGAPDataBlockConfig` object. This value defines the type of BLE GAP data block and its applicable properties.

Output Arguments

`dataBlock` — Generated BLE GAP data block

character array

Generated BLE GAP data block, returned as a character array. Each row in this array is the hexadecimal representation of an octet.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Supplement to the Bluetooth Core Specification." CSS Version 7 (2016).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleGAPDataBlockDecode`

Objects
`bleGAPDataBlockConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleGAPDataBlockDecode

Decode BLE GAP data block

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[status,cfgGAP] = bleGAPDataBlockDecode(dataBlock)
```

Description

`[status,cfgGAP] = bleGAPDataBlockDecode(dataBlock)` decodes a Bluetooth low energy (BLE) generic access profile (GAP) data block, `dataBlock`, of the type advertising data (AD) or scan response data (SRD), returning the decoding status, `status`, and the BLE GAP data block configuration object, `cfgGAP`.

Examples

Decode BLE GAP AD Blocks

Decode two unique BLE GAP AD blocks: one with AD types `'Flags'` and `'Tx power level'` and the other with AD types `'Advertising interval'` and `'Local name'`.

Create a configuration object for a BLE GAP AD block. Specify the AD types as `'Flags'` and `'Tx power level'`. Set the values of LE discoverability as `'Limited'` and Tx power level to `45`. View the properties of the configuration object `'cfgGAP'`.

```
cfgGAP = bleGAPDataBlockConfig;
cfgGAP.AdvertisingDataTypes = {'Flags'; 'Tx power level'};
cfgGAP.LEDiscoverability = 'Limited';
cfgGAP.TxPowerLevel = 45
```

```
cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
       LEDiscoverability: 'Limited'
                   BREDR: 0
            TxPowerLevel: 45
```

Generate a BLE GAP AD block from the corresponding configuration object.

```
dataBlock = bleGAPDataBlock(cfgGAP);
```

Decode the generated BLE GAP AD block. The returned status indicates decoding was successful. View the output of `'status'` and `'cfgGAP'`.

```
[status, cfgGAP] = bleGAPDataBlockDecode(dataBlock)
```

```
status = 
Success

cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
          LEDiscoverability: 'Limited'
                      BREDR: 0
               TxPowerLevel: 45
```

Create another BLE GAP AD block configuration object, this time specifying AD types `'Advertising interval'` and `'Local name'`. Set the values of advertising interval as `48`, local name as `'MathWorks'`, and local name shortening as `true`. View the properties of the configuration object `'cfgGAP'`.

```
cfgGAP = bleGAPDataBlockConfig('AdvertisingDataTypes', ...
    {'Advertising interval','Local name'});
cfgGAP.AdvertisingInterval = 48;
cfgGAP.LocalName = 'MathWorks';
cfgGAP.LocalNameShortening = true
```

```
cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'MathWorks'
     LocalNameShortening: 1
     AdvertisingInterval: 48
```

Generate the BLE GAP AD block from the corresponding configuration object.

```
dataBlock = bleGAPDataBlock(cfgGAP);
```

Decode the generated BLE GAP AD block. The returned status indicates decoding was successful. View the output of `'status'` and `'cfgGAP'`.

```
[status, cfgGAP] = bleGAPDataBlockDecode(dataBlock)
```

```
status = 
Success

cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'MathWorks'
     LocalNameShortening: 1
     AdvertisingInterval: 48
```

Decode Corrupted BLE GAP AD Block

Specify a BLE GAP AD block containing corrupted data values.

```
dataBlock = ['010106010202']; % Sample BLE GAP AD block with corrupted data values
```

Decode the specified BLE GAP AD block. The returned status indicates that the decoding failed due to the corrupted input BLE GAP AD block. In this case, when decoding fails, the BLE GAP AD block configuration object, `'cfgGAP'`, displays no properties.

```
[status, cfgGAP] = bleGAPDataBlockDecode(dataBlock)
```

```
status = 
MismatchGAPADLength

cfgGAP = 
  bleGAPDataBlockConfig with properties:
```

Input Arguments

`dataBlock` — BLE GAP data block

character vector | string scalar | numeric vector | character array

BLE GAP data block, specified as one of these values:

- Character vector — This vector represents octets in hexadecimal format.
- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0, 255] — This vector represents octets in decimal format.
- *n*-by-2 character array — Each row represents an octet in hexadecimal format.

Data Types: `char` | `string` | `double`

Output Arguments

`status` — BLE GAP data block decoding status

nonpositive integer

BLE GAP data block decoding status, returned as a nonpositive integer of type `blePacketDecodeStatus`. This value represents the result of an BLE GAP data block decoding. Each value of `status` corresponds to a member of the `blePacketDecodeStatus` enumeration class, which indicates the packet decoding status according to this table.

Enumeration Value	Member of Enumeration Class	Decoding Status
0	`Success`	Packet decoding succeeded
-201	`InvalidGAPADLength`	GAP AD length is not valid
-202	`MismatchGAPADLength`	Received AD length does not match with actual length

-203	UnsupportedGAPADType	Advertising data type is not valid or not supported
-204	InvalidGAPAdvertisingInterval	Advertising interval is not valid
-205	InvalidGAPConnectionIntervalRange	Invalid connection interval
-206	InvalidGAPConnectionIntervalMinimum	Invalid interval minimum
-207	InvalidGAPConnectionIntervalMaximum	Invalid interval maximum

An enumeration value other than 0 means that the BLE GAP data block decoding failed. If the decoding fails, object cfgGAP displays no output.

cfgGAP — BLE GAP data block configuration object
bleGAPDataBlockConfig | object

BLE GAP data block configuration object, returned as a bleGAPDataBlockConfig object. This value defines the type of BLE GAP data block and its applicable properties.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Supplement to the Bluetooth Core Specification." CSS Version 7 (2016).

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
bleGAPDataBlock

Objects
bleGAPDataBlockConfig

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleIdealReceiver

Ideal receiver for BLE PHY waveform

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[bits,accessAddr] = bleIdealReceiver(waveform)
[bits,accessAddr] = bleIdealReceiver(waveform,Name,Value)
[ ___ ,IQsamples] = bleIdealReceiver( ___ )
```

Description

`[bits,accessAddr] = bleIdealReceiver(waveform)` decodes the Bluetooth low energy (BLE) `waveform`, generated by `bleWaveformGenerator` , and returns the received bits, `bits`, and the access address information, `accesAddr`.

`[bits,accessAddr] = bleIdealReceiver(waveform,Name,Value)` also specifies options using one or more name-value pair arguments. For example, `'Mode','LE2M'` specifies the physical (PHY) layer transmission mode of the desired BLE waveform.

`[ ___ ,IQsamples] = bleIdealReceiver( ___ )` also returns the in-phase and quadrature (IQ) samples, `IQsamples`, corresponding to constant tone extension (CTE).

Examples

Decode BLE Waveform Using Default Settings

Create an input message column vector of length 1000 containing random binary values. Generate a BLE transmit waveform from the transmission bits by using the `bleWaveformGenerator` function.

```
txBits = randi([0 1],1000,1);
txWaveform = bleWaveformGenerator(txBits);
```

Pass the transmit waveform through a noisy channel and obtain the received waveform.

```
snr = 30; % specified in dB
rxWaveform = awgn(txWaveform,snr);
```

Recover data bits from the received BLE waveform using `bleIdealReceiver`. Check for the number of bit errors in the recovered bits. The returned value indicates that the BLE waveform was successfully decoded.

```
[rxBits,accessAddr] = bleIdealReceiver(rxWaveform);
numErr = biterr(txBits,rxBits)
```

```
numErr = 0
```

Decode BLE Waveform Using Specified Name-Value Pairs

Specify the values of PHY generating mode, channel index and samples per symbol (sps).

```
phyMode = 'LE125K';
chanIndex = 2;
sps = 4;
```

Generate transmission bits containing random binary values. Obtain the BLE transmit waveform from the transmission bits and the specified name-value pairs using the `bleWaveformGenerator` function.

```
txBits = randi([0 1],100,1);
txWaveform = bleWaveformGenerator(txBits,'Mode',phyMode,...
    'SamplesPerSymbol',sps,'ChannelIndex',chanIndex);
```

Recover the data bits, and then compare them with the transmission bits. The recovered data bits match the transmission bits, indicating there are no errors in the decoded BLE waveform.

```
rxBits = bleIdealReceiver(txWaveform,'Mode',phyMode,...
    'SamplesPerSymbol',sps,'ChannelIndex',chanIndex);
isequal(txBits,rxBits)
```

```
ans = logical
   1
```

Decode BLE Waveform With CTE For Connectionless Scenario

Specify a connectionless advertising channel protocol data unit (PDU) for angle of arrival (AoA) CTE.

```
pduHex = '02049B0327';
pdu = de2bi(hex2dec(pduHex),40)';
```

Generate and append cyclic redundancy check (CRC) to the PDU.

```
crcGen = comm.CRCGenerator('z^24+z^10+z^9+z^6+z^4+z^3+z+1',...
    'InitialConditions',de2bi(hex2dec('555551'),'left-msb',24),...
    'DirectMethod',true);
pduCRC = crcGen(pdu);
```

Generate the BLE transmit waveform using specified name-value pair arguments.

```
txWaveform = bleWaveformGenerator(pduCRC,'ChannelIndex',36,...
    'DFPacketType','ConnectionlessCTE');
```

Recover the data bits by demodulating, dewhitening, and IQ sampling for a slot duration of 2 μs.

```
[bits,accAddr,iqSamples] = bleIdealReceiver(txWaveform,...
    'ChannelIndex',36,'DFPacketType','ConnectionlessCTE');
```

Input Arguments

`waveform` — Received time-domain signal

complex-valued vector

Received time-domain signal, specified as a complex-valued signal with size N_s-by-1, where N_s represents the number of received samples. The values of N_s depend on the `'Mode'` and `'SamplesPerSymbol'` (sps) name-value pairs, according to the constraints specified in this table. For example, if the value of `'Mode'` is `'LE1M'` and the value of `'SamplesPerSymbol'` is 4 then the value of N_s must be greater than or equal to 160 and a multiple of `'SamplesPerSymbol'`.

Value of `'Mode'`	Value of N_s	Multiple of
`'LE1M'`	≥40×*sps*	*sps*
`'LE2M'`	≥48×*sps*	*sps*
`'LE500K'`	≥376×*sps*	2×*sps*
`'LE125K'`	≥376×*sps*	8×*sps*

Data Types: `double` | `single`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bleIdealReceiver(waveform,'Mode','LE2M','ChannelIndex',36)`

`Mode` — PHY transmission mode

`'LE1M'` (default) | `'LE2M'` | `'LE500K'` | `'LE125K'`

PHY transmission mode, specified as the comma-separated pair consisting of `'Mode'` and `'LE1M'`, `'LE2M'`, `'LE500K'`, or `'LE125K'`. This value indicates the type of PHY used to decode the received BLE waveform.

Data Types: `string` | `char`

`ChannelIndex` — Channel Index

`37` (default) | integer in the range [0, 39]

Channel index, specified as the comma-separated pair consisting of `'ChannelIndex'` and an integer in the range [0, 39]. For data channels, this value must be in the range [0, 36]. This value is used by the data-dewhitening block.

Data Types: `single` | `double`

`SamplesPerSymbol` — Samples per symbol

`8` (default) | positive integer

Samples per symbol, specified as the comma-separated pair consisting of `'SamplesPerSymbol'` and a positive integer. The object uses this value for Gaussian frequency shift keying (GFSK) modulation.

Data Types: `single` | `double`

`DFPacketType` — Type of direction finding packet

`'Disabled'` (default) | `'ConnectionlessCTE'` | `'ConnectionCTE'`

Type of direction finding packet, specified as the comma-separated pair consisting of `'DFPacketType'` and `'ConnectionlessCTE'`, `'ConnectionCTE'`, or `'Disabled'`.

Data Types: `string` | `char`

`SlotDuration` — Switch and sample slot duration
2 (default) | 1

Switch and sample slot duration, specified as the comma-separated pair consisting of `'SlotDuration'` and `1` or `2`. This value must be expressed in microseconds.

Data Types: `double`

Output Arguments

`bits` — Payload bits
column vector

Payload bits, returned as a column vector of maximum length 260 bytes. This output represents the recovered information bits.

Data Types: `int8`

`accessAddr` — Access address information
32-bit column vector

Access address information, returned as a 32-bit column vector. This output is used by the higher layers for validating a packet.

Data Types: `int8`

`IQsamples` — IQ samples
complex-valued column vector

IQ samples, specified as a complex-valued column vector. This argument corresponds to the 8 µs value of the reference period and slot duration. If the value of “'DFPacketType'” on page 2-0 argument is `'ConnectionlessCTE'` or `'ConnectionCTE'`, then the function returns this argument.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleWaveformGenerator` | `bluetoothWaveformGenerator` | `bluetoothIdealReceiver` | `bleAngleEstimate`

Objects
`bleAngleEstimateConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"
"Bluetooth Location and Direction Finding"

Introduced in R2019b

bleL2CAPFrame

Generate BLE L2CAP frame

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
L2CAPFrame = bleL2CAPFrame(cfgL2CAP)
L2CAPFrame = bleL2CAPFrame(cfgL2CAP,SDU)
```

Description

`L2CAPFrame = bleL2CAPFrame(cfgL2CAP)` generates a Bluetooth low energy (BLE) logical link control and adaptation protocol (L2CAP) frame, `L2CAPFrame`, for a given BLE L2CAP configuration object, `cfgL2CAP`. Use this syntax to generate the signaling frames.

`L2CAPFrame = bleL2CAPFrame(cfgL2CAP,SDU)` generates a BLE L2CAP frame, `L2CAPFrame`, for a given BLE L2CAP configuration object `cfgL2CAP` and the upper-layer payload service data unit (SDU), `SDU`. Use this syntax to generate the data frames.

Examples

Generate BLE L2CAP Signaling Command Frame

Create a BLE L2CAP configuration object, `'cfgL2CAP'`, and view the corresponding default properties.

```
cfgL2CAP = bleL2CAPFrameConfig
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Credit based connection request'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 1
```

Change the value of credits to `10`.

```
cfgL2CAP.Credits            = 10;
```

Generate a BLE L2CAP signaling command frame from the corresponding configuration object.

```
l2capFrame = bleL2CAPFrame(cfgL2CAP)
```

```
l2capFrame = 18x2 char array
    '0E'
    '00'
    '05'
    '00'
    '14'
    '01'
    '0A'
    '00'
    '1F'
    '00'
    '40'
    '00'
    '17'
    '00'
    '17'
    '00'
    '0A'
    '00'
```

Generate BLE L2CAP Data Frames

Generate two unique BLE L2CAP data frames: one with SDU from the attribute protocol (ATT) layer as `'0A0100'` and the other with an upper-layer payload SDU,`'0A01E2D3'`.

Create a BLE L2CAP configuration object, `'cfgL2CAP'`, and view the default properties.

```
cfgL2CAP = bleL2CAPFrameConfig
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Credit based connection request'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 1
```

Change the value of channel identifier to `'0004'`.

```
cfgL2CAP.ChannelIdentifier = '0004'; % Channel identifier for ATT
```

Generate a BLE L2CAP data frame from `'cfgL2CAP'`, specifying the upper-layer payload SDU from the ATT layer as `'0A0100'`.

```
l2capFrame = bleL2CAPFrame(cfgL2CAP,"0A0100")
```

```
l2capFrame = 7x2 char array
    '03'
    '00'
    '04'
    '00'
    '0A'
    '01'
    '00'
```

Create another BLE L2CAP configuration object, `'cfgL2CAP'`, with default properties. Set the value of channel identifier to `'007A'`.

```
cfgL2CAP = bleL2CAPFrameConfig;
cfgL2CAP.ChannelIdentifier = '007A'; % Dynamic channel identifier
```

Generate a BLE L2CAP data frame from `'cfgL2CAP'`, specifying the upper-layer payload SDU as `'0A01E2D3'`.

```
l2capFrame = bleL2CAPFrame(cfgL2CAP,['0A'; '01'; 'E2'; 'D3'])
```

```
l2capFrame = 10x2 char array
    '06'
    '00'
    '7A'
    '00'
    '04'
    '00'
    '0A'
    '01'
    'E2'
    'D3'
```

Input Arguments

`cfgL2CAP` — BLE L2CAP configuration object

`bleL2CAPFrameConfig` object

BLE L2CAP configuration object, specified as a `bleL2CAPFrameConfig` object. The function uses this object to configure the BLE L2CAP frame and its applicable properties.

SDU — Upper-layer payload

character vector | string scalar | numeric vector | character array

Upper-layer payload, specified as one of these types:

- Character vector — This vector represents octets in hexadecimal format.
- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0, 255] — This vector represents octets in decimal format.
- *n*-by-2 character array — Each row represents an octet in hexadecimal format.

Data Types: `char` | `double` | `string`

Output Arguments

`L2CAPFrame` — Generated BLE L2CAP frame

character array

Generated BLE L2CAP frame, returned as a character array. Each row of the array represents an octet in hexadecimal format. This value represents the output BLE L2CAP frame.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleL2CAPFrameDecode`

Objects
`bleL2CAPFrameConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleL2CAPFrameDecode

Decode BLE L2CAP frame

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[status,cfgL2CAP,SDU] = bleL2CAPFrameDecode(L2CAPFrame)
```

Description

`[status,cfgL2CAP,SDU] = bleL2CAPFrameDecode(L2CAPFrame)` decodes the specified Bluetooth low energy (BLE) logical link control and adaptation protocol (L2CAP) frame, `L2CAPFrame`. The function returns the decoding status, `status`, the corresponding BLE L2CAP configuration object, `cfgL2CAP`, and the upper-layer payload service data unit (SDU), `SDU`.

Examples

Decode BLE L2CAP Data Frame with SDU

Create a BLE L2CAP configuration object, `'cfgL2CAP'`, with default properties and view the applicable properties. Change the value of channel identifier to `'0004'`.

```
cfgL2CAP = bleL2CAPFrameConfig
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Credit based connection request'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 1
```

```
cfgL2CAP.ChannelIdentifier = '0004'; % Channel identifier for ATT
```

Generate a BLE L2CAP data frame from `'cfgL2CAP'`, specifying the upper-layer payload SDU from attribute protocol (ATT) layer as `'0A0100'`.

```
L2CAPFrame = bleL2CAPFrame(cfgL2CAP, "0A0100");
```

Decode the generated BLE L2CAP data frame. The returned status indicates decoding was successful. View the output of `'status'`, `'cfgL2CAP'` and the `'SDU'`.

```
[status, cfgL2CAP, SDU] = bleL2CAPFrameDecode(L2CAPFrame)
```

```
status = 
Success

cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'


SDU = 3x2 char array
    '0A'
    '01'
    '00'
```

Decode Corrupted BLE L2CAP Frame

Specify a BLE L2CAP frame containing corrupted data values.

```
l2capFrame = ['090005000107040060005000']; % Sample frame
```

Decode the specified BLE L2CAP frame. The returned status indicates that the decoding failed due to the corrupted input L2CAP frame. In this case, when decoding fails, the output displays the reason of failure and the BLE L2CAP frame configuration object, `'cfgL2CAP'`, displays no properties.

```
[status, cfgL2CAP, SDU] = bleL2CAPFrameDecode(l2capFrame)
```

```
status = 
MismatchL2CAPHeaderLength

cfgL2CAP = 
  bleL2CAPFrameConfig with properties:


SDU =

  1x0 empty char array
```

Input Arguments

L2CAPFrame — BLE L2CAP frame

character vector | string scalar | numeric vector | character array

BLE L2CAP frame, specified as one of these values:

- Character vector — This vector represents octets in hexadecimal format.
- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0, 255] — This vector represents octets in decimal format.
- *n*-by-2 character array — Each row represents an octet in hexadecimal format.

Data Types: `char` | `double` | `string`

Output Arguments

`status` — Packet decoding status

nonpositive integer

Packet decoding status, returned as a nonpositive integer of type `blePacketDecodeStatus`. This value represents the result of decoding a BLE L2CAP frame. Each value of `status` corresponds to a member of the `blePacketDecodeStatus` enumeration class, which indicates the packet decoding status according to this table.

Enumeration Value	Member of Enumeration Class	Decoding Status
0	`Success`	Packet decoding succeeded
–301	`InvalidL2CAPConnectionIntervalRange`	Invalid connection intervals
–302	`InvalidL2CAPSlaveLatency`	Invalid slave latency
–303	`InvalidLECredits`	Invalid low energy (LE) credits
–304	`L2CAPSegmentationUnsupported`	Segmentation is not supported
–305	`MismatchL2CAPHeaderLength`	Length mismatches with actual length
–306	`IncompleteL2CAPDataFrame`	L2CAP data frame is not sufficient to decode
–307	`InvalidL2CAPChannelIdentifier`	Invalid L2CAP channel identifier
–308	`InvalidL2CAPCommand`	Invalid L2CAP command code
–309	`InvalidL2CAPCommandRejectReason`	Invalid command reject reason code
–310	`InvalidL2CAPParameterUpdateResult`	Invalid parameters update result
-311	`InvalidL2CAPConnectionResult`	Invalid connection result code
-312	`IllegalL2CAPSignalIdentifier`	Illegal signal identifier in L2CAP
-313	`InvalidL2CAPConnectionIntervalMinimum`	Invalid interval minimum
-314	`InvalidL2CAPConnectionIntervalMaximum`	Invalid interval maximum
-315	`InvalidL2CAPConnectionTimeout`	Invalid connection timeout
-316	`InvalidLEPSM`	Invalid LE protocol/service multiplexer

-317	`InvalidL2CAPChannelMTU`	Invalid maximum transmission unit
-318	`InvalidL2CAPChannelMPS`	Invalid maximum PDU payload size
-319	`InvalidL2CAPSDULength`	Invalid SDU length
-320	`MismatchL2CAPSignalFrameLength`	Length mismatches with actual length
-321	`IncompleteL2CAPSignalFrame`	L2CAP signal frame is not valid or not sufficient

An enumeration value other than `0` means that the BLE ATT PDU decoding failed. If the decoding fails, object `cfgATT` displays no output.

`cfgL2CAP` — BLE L2CAP frame configuration object
`bleL2CAPFrameConfig` object

BLE L2CAP frame configuration object, returned as a `bleL2CAPFrameConfig` object. This value denotes the decoded BLE L2CAP frame configuration.

`SDU` — Upper-layer payload
character array

Upper-layer payload, returned as a character array. Each row represents an octet in hexadecimal format.

Data Types: `char` | `string`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleL2CAPFrame`

Objects
`bleL2CAPFrameConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLAdvertisingChannelPDU

Generate BLE LL advertising channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see "Get and Manage Add-Ons". Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

pdu = bleLLAdvertisingChannelPDU(cfgLLAdv)

Description

pdu = bleLLAdvertisingChannelPDU(cfgLLAdv) generates a Bluetooth low energy (BLE) link layer (LL) advertising channel protocol data unit (PDU) corresponding to the BLE LL advertising channel PDU configuration object cfgLLAdv.

Examples

Generate BLE LL Advertising Channel PDUs

Generate two unique BLE LL advertising channel PDUs: first one of the type 'Advertising indication' using advertising data '020106' and the other of type 'Connection indication' using a set of data channels to be used.

Create a BLE LL advertising channel PDU configuration object, 'cfgLLAdv', with the opcode as 'Advertising indication' by using advertising data '020106'. View the configured properties corresponding to the opcode.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig;
cfgLLAdv.AdvertisingData = '020106'
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Advertising indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
          AdvertisingData: [3x2 char]
```

Generate the BLE LL advertising channel PDU by using the corresponding configuration object. Display the PDU length in octets.

```
pdu = bleLLAdvertisingChannelPDU(cfgLLAdv);
numel(pdu)/8
```

```
ans = 14
```

Display the first octet of the generated BLE LL advertising channel PDU.

```
pdu(1:8)
```

```
ans = 8×1

     0
     0
     0
     0
     0
     0
     1
     0
```

Create another BLE LL advertising channel PDU configuration object, this time using the name-value pairs. Change the BLE LL advertising channel PDU opcode to `'Connection indication'`. View the active configured properties of the specified opcode.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig('PDUType', ...
    'Connection indication')
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Connection indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
     InitiatorAddressType: 'Random'
         InitiatorAddress: '0123456789CD'
            AccessAddress: '01234567'
        CRCInitialization: '012345'
               WindowSize: 1
             WindowOffset: 0
       ConnectionInterval: 6
             SlaveLatency: 0
        ConnectionTimeout: 10
             UsedChannels: [1x37 double]
             HopIncrement: 5
       SleepClockAccuracy: '251 to 500 ppm'
```

Specify the value of connection interval as `8` and the set of data channels as `[0 4 12 16 18 24 25]`. View the configured properties.

```
cfgLLAdv.ConnectionInterval = 8; % in milliseconds
cfgLLAdv.UsedChannels       = [0 4 12 16 18 24 25]
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Connection indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
     InitiatorAddressType: 'Random'
```

```
         InitiatorAddress: '0123456789CD'
            AccessAddress: '01234567'
        CRCInitialization: '012345'
               WindowSize: 1
             WindowOffset: 0
       ConnectionInterval: 8
             SlaveLatency: 0
        ConnectionTimeout: 10
             UsedChannels: [0 4 12 16 18 24 25]
             HopIncrement: 5
       SleepClockAccuracy: '251 to 500 ppm'
```

Generate the BLE LL advertising channel PDU from the corresponding configuration object. Display the PDU length in octets.

```
pdu = bleLLAdvertisingChannelPDU(cfgLLAdv);
numel(pdu)/8
```

```
ans = 39
```

Display the first octet of the generated BLE LL advertising channel PDU.

```
pdu(1:8)
```

```
ans = 8×1

     1
     0
     1
     0
     0
     0
     1
     1
```

Input Arguments

cfgLLAdv — BLE LL advertising channel PDU configuration object
bleLLAdvertisingChannelPDUConfig object

BLE LL advertising channel PDU configuration object, specified as a bleLLAdvertisingChannelPDUConfig object. This value defines the type of generated BLE LL advertising channel PDU and its applicable properties.

Output Arguments

pdu — Generated BLE LL advertising channel PDU
binary column vector

Generated BLE LL advertising channel PDU, returned as a binary column vector. This value represents the output BLE LL advertising channel PDU.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleLLAdvertisingChannelPDUDecode`

Objects
`bleLLAdvertisingChannelPDUConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLAdvertisingChannelPDUDecode

Decode BLE LL advertising channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[status,cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu)
[status,cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu,Name,Value)
```

Description

`[status,cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu)`decodes the specified Bluetooth low energy (BLE) link layer (LL) advertising channel protocol data unit (PDU), returning the decoding status, `status`, and the corresponding BLE LL advertising channel PDU configuration object, `cfgLLAdv`.

`[status,cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `'InputFormat','octets'` specifies the input BLE LL advertising channel PDU in the form of octets.

Examples

Decode BLE LL Advertising Channel PDU Given in Bits

Create a BLE LL advertising channel PDU configuration object, `'cfgLLAdv'`, with default settings and view the corresponding applicable properties.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Advertising indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
          AdvertisingData: [3x2 char]
```

Generate the BLE LL advertising channel PDU from the corresponding configuration object.

```
pdu = bleLLAdvertisingChannelPDU(cfgLLAdv);
```

Decode the generated BLE LL advertising channel PDU. The returned status indicates decoding is successful. View the output of `'status'`and `'cfgLLAdv'`.

```
[status, cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu)
```

```
status = 
Success

cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                PDUType: 'Advertising indication'
       ChannelSelection: 'Algorithm1'
  AdvertiserAddressType: 'Random'
      AdvertiserAddress: '0123456789AB'
        AdvertisingData: [3x2 char]
```

Decode BLE LL Advertising Channel PDU Given in Octets

Specify a sample BLE LL advertising channel PDU in octets.

```
pdu = 'C409AB8967452301020106A8F1DF'; % Sample PDU in octets
```

Decode the specified BLE LL advertising channel PDU by specifying `'InputFormat'` to `'octets'`. The returned status indicates decoding is successful. View the output of `'status'` and `'cfgLLData'`.

```
[status, cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu, ...
    'InputFormat','octets')
```

```
status = 
Success

cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                PDUType: 'Scan response'
       ChannelSelection: 'Algorithm1'
  AdvertiserAddressType: 'Random'
      AdvertiserAddress: '0123456789AB'
       ScanResponseData: [3x2 char]
```

Decode Corrupted BLE LL Advertising Channel PDU

Specify a BLE LL advertising channel PDU containing corrupted data values.

```
pdu = 'D409AB89674523010201'; % Sample corrupted PDU
```

Decode the specified BLE LL advertising channel PDU. The returned status indicates that the decoding failed due to corrupted input BLE LL advertising channel PDU. In case of failed decoding, the reason of failure is indicated and the BLE LL advertising channel PDU configuration object, `'cfgLLAdv'`, displays no properties.

```
[status, cfgLLAdv] = bleLLAdvertisingChannelPDUDecode(pdu, ...
    'InputFormat','octets')
```

```
status = 
CRCFailed

cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:
```

Input Arguments

pdu — BLE LL advertising channel PDU

character vector | string scalar | numeric vector | character array | binary vector

BLE LL advertising channel PDU, specified as one of these types:

- Character vector — This vector represents octets in hexadecimal format.
- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0,255] — This vector represents octets in decimal format.
- `n-by-2` character array — Each row represents an octet in hexadecimal format.
- Binary vector — This vector represents the BLE LL advertising channel PDU bits.

Data Types: `char` | `double` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[status,cfgLLAdv] = bleLLAdvertisingChannelPDUDecode('pdu','InputFormat','octets')`

`InputFormat` — BLE LL advertising channel PDU format

`'bits'` (default) | `'octets'`

BLE LL advertising channel PDU format, specified as `'bits'` or`'octets'`. When specified as `'bits'`, the `InputFormat` is a binary vector. When specified as `'octets'`, `InputFormat` is a numeric vector representing octets in decimal format or a character array or a string scalar representing octets in hexadecimal format.

Data Types: `char` | `string`

Output Arguments

`status` — BLE LL advertising channel PDU decoding status

nonpositive integer

BLE LL advertising channel PDU decoding status, returned as a nonpositive number of type `blePacketDecodeStatus`. This value represents the result of BLE LL advertising channel PDU decoding. Each value of `status` corresponds to a member of the `blePacketDecodeStatus` enumeration class, which indicates the packet decoding status listed in this table.

Enumeration Value	Member of Enumeration Class	Decoding Status
0	`Success`	Packet decoding succeeded
-1	`CRCFailed`	Link Layer PDU is corrupted
-2	`LLPDULengthMismatch`	Length field does not match with actual PDU length
-3	`InvalidLLSlaveLatency`	Invalid slave latency
-4	`InvalidLLConnectionTimeout`	Invalid connection timeout
-5	`InvalidLLWindowSize`	Invalid window size
-6	`InvalidLLWindowOffset`	Invalid window offset
-7	`InvalidLLConnectionInterval`	Invalid connection interval
-8	`InvalidLLChannelMap`	Invalid channel map
-51	`IncompleteLLAdvertisingChannelPDU`	Insufficient octets in advertising channel PDU
-52	`InvalidLLHopIncrement`	Invalid hop increment value
-53	`InvalidLLAdvertisingDataLength`	Invalid advertising data
-54	`InvalidLLScanResponseDataLength`	Invalid scan response data
-55	`UnsupportedLLAdvertisingPDUType`	Unsupported advertising channel PDU

An enumeration value other than `0` means that the BLE LL advertising channel PDU decoding failed. If the decoding fails, object `cfgLLAdv` displays no output.

`cfgLLAdv` — BLE LL advertising channel PDU configuration object
`bleLLAdvertisingChannelPDUConfig` object

BLE LL advertising channel configuration object, returned as a `bleLLAdvertisingChannelPDUConfig` object. This value represents the decoded BLE LL advertising channel PDU configuration.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleLLAdvertisingChannelPDU`

Objects
`bleLLAdvertisingChannelPDUConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLDataChannelPDU

Generate BLE LL data channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
pdu = bleLLDataChannelPDU(cfgLLData)
pdu = bleLLDataChannelPDU(cfgLLData,LLPayload)
```

Description

`pdu = bleLLDataChannelPDU(cfgLLData)` generates a Bluetooth low energy (BLE) link layer (LL) data channel protocol data unit (PDU), `pdu`, for a given BLE LL data channel configuration object `cfgLLData`. This syntax is used for generating BLE LL control PDUs.

`pdu = bleLLDataChannelPDU(cfgLLData,LLPayload)` generates a BLE LL data channel PDU, `pdu`, containing the upper-layer payload `LLPayload` for a given BLE LL data channel configuration object `cfgLLData`. This syntax is used for generating BLE LL data PDUs.

Examples

Generate BLE LL Control PDU of Type Connection Update Indication

Create a BLE LL control PDU configuration object for a control PDU using the default configuration. View the corresponding default properties.

```
cfgControl = bleLLControlPDUConfig
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

                Opcode: 'Connection update indication'
            WindowSize: 1
          WindowOffset: 0
    ConnectionInterval: 6
          SlaveLatency: 0
     ConnectionTimeout: 10
               Instant: 0
```

Create a BLE LL data channel PDU configuration object for a control PDU of type `'Connection update indication'` by configuring the values of link layer identifier (LLID) as `'Control'` and ControlConfig as `'cfgControl'`. View the corresponding properties.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', 'Control', ...
    'ControlConfig', cfgControl)
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Control'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: '012345'
        ControlConfig: [1x1 bleLLControlPDUConfig]
```

Generate a BLE LL data channel PDU of type `'Connection update indication'` by using the configuration object `'cfgLLData'`. Display the PDU length in octets.

```
pdu = bleLLDataChannelPDU(cfgLLData);
numel(pdu)/8
```

```
ans = 17
```

Display the first octet of the generated BLE LL data channel PDU.

```
pdu(1:8)
```

```
ans = 8×1

     1
     1
     0
     0
     0
     0
     0
     0
```

Generate BLE LL Data PDU Using Upper-Layer Payload

Create a BLE LL data channel PDU configuration object, `'cfgLLData'`, for a data PDU by using the default configuration. View the corresponding default properties.

```
cfgLLData = bleLLDataChannelPDUConfig
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Data (continuation fragment/empty)'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: '012345'
```

Generate a BLE LL data PDU by using the corresponding configuration object, `'cgLLData'` and the upper-layer payload `'030004000A0100'`. Display the PDU length in octets.

```
pdu = bleLLDataChannelPDU(cfgLLData,'030004000A0100');
numel(pdu)/8

ans = 12
```

Display the first octet of the generated BLE LL data PDU.

```
pdu(1:8)

ans = 8×1

     1
     0
     0
     0
     0
     0
     0
     0
```

Input Arguments

`cfgLLData` — BLE LL data channel configuration object

`bleLLDataChannelPDUConfig` object

BLE LL data channel configuration object, specified as a `bleLLDataChannelPDUConfig` object. This object is used to configure the BLE LL data channel PDU and its applicable properties.

LLPayload — Upper-layer payload

character vector | string scalar | numeric vector | character array

Upper-layer payload, specified as one of these types:

- Character vector — This vector represents octets in hexadecimal format.
- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0,255] — This vector represents octets in decimal format.
- `n-by-2` character array — Each row represents an octet in hexadecimal format.

Data Types: `char` | `double` | `string`

Output Arguments

pdu — Generated BLE LL data channel PDU

binary vector

Generated BLE LL data channel PDU, returned as a binary column vector. This value represents the output BLE LL data channel PDU.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When `CRCInitialization` property of the `bleLLDataChannelPDUConfig` object is set to constant, code generation is possible.

See Also

Functions

`bleLLDataChannelPDUDecode`

Objects

`bleLLDataChannelPDUConfig` | `bleLLControlPDUConfig`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLDataChannelPDUDecode

Decode BLE LL data channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[status,cfgLLData,LLPayload] = bleLLDataChannelPDUDecode(pdu,CRCinit)
[status,cfgLLData,LLPayload] = bleLLDataChannelPDUDecode(pdu,CRCinit
Name,Value)
```

Description

`[status,cfgLLData,LLPayload] = bleLLDataChannelPDUDecode(pdu,CRCinit)` decodes a Bluetooth low energy (BLE) link layer (LL) data channel protocol data unit (PDU), returning the decoding status, `status`, the BLE LL data channel PDU configuration object, `cfgLLData`, and the upper-layer payload, `LLPayload`. The `CRCinit` denotes the initialization value of cyclic redundancy check (CRC).

`[status,cfgLLData,LLPayload] = bleLLDataChannelPDUDecode(pdu,CRCinit Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `'InputFormat','bits'` specifies the format of input BLE LL data channel PDU in bits.

Examples

Decode BLE LL Data Channel PDU Given in Bits

Create a BLE LL data channel PDU configuration object by using default settings and view the corresponding applicable properties.

```
cfgLLData = bleLLDataChannelPDUConfig
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Data (continuation fragment/empty)'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: '012345'
```

Initialize and set the LLID value to `'start fragment/complete'` indicating the BLE LL data channel PDU with the upper-layer payload. Initialize the cyclic redundancy check (CRC) value to `'ED321C'`.

```
cfgLLData.LLID = 'Data (start fragment/complete)';
cfgLLData.CRCInitialization = 'ED321C';
crcInit = 'ED321C'; % CRC initialization value
```

Generate the BLE LL data channel PDU with upper-layer payload given in hexadecimal octets.

```
pdu = bleLLDataChannelPDU(cfgLLData,'030004000A0100');
```

Decode the generated BLE LL data channel PDU. The returned status indicates decoding is successful. View the output of `'status'`, `'cfgLLData'` and `'llPayload'`.

```
[status, cfgLLData, llPayload] = bleLLDataChannelPDUDecode(pdu,crcInit)
```

```
status =
Success

cfgLLData =
  bleLLDataChannelPDUConfig with properties:

               LLID: 'Data (start fragment/complete)'
               NESN: 0
     SequenceNumber: 0
           MoreData: 0
  CRCInitialization: '012345'


llPayload = 7x2 char array
    '03'
    '00'
    '04'
    '00'
    '0A'
    '01'
    '00'
```

Decode BLE LL Data Channel PDU Given in Octets

Specify a sample BLE LL data channel PDU in octets.

```
pdu = '030C00010000060000000A000000FCD2A6'; % Sample PDU in octets
```

Initialize the CRC value.

```
crcInit = 'ED323C'; % CRC initialization value
```

Decode the specified BLE LL data channel PDU by specifying `'InputFormat'` to `'octets'`. The specified PDU is a control PDU. The returned status indicates decoding is successful. View the output of `'status'`, `'cfgLLData'` and `'llPayload'`. You can see the decoded configuration of the specified PDU in the `'ControlConfig'` property of `'cfgLLData'`.

```
[status, cfgLLData, llPayload] = bleLLDataChannelPDUDecode(...
    pdu, crcInit, ...
    'InputFormat', ...
    'octets')
```

```
status = 
Success

cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Control'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
    CRCInitialization: '012345'
       ControlConfig: [1x1 bleLLControlPDUConfig]


llPayload =

  1x0 empty char array
```

Decode Corrupted BLE LL Data Channel PDU

Specify a BLE LL data channel PDU containing corrupted data values. Initialize the CRC value.

```
pdu = '040C00010000060000000A00'; % Sample corrupted PDU
crcInit = 'CD3234'; % CRC initialization value
```

Decode the specified BLE LL data channel PDU. The returned status indicates that the decoding failed due to the corrupted BLE LL data channel PDU. If the decoding fails, the reason is indicated and the BLE LL data channel PDU configuration object, `'cfgLLData'`, displays no properties.

```
[status, cfgLLData, llPayload] = bleLLDataChannelPDUDecode(...
    pdu, crcInit, ...
    'InputFormat', ...
    'octets')
```

```
status = 
CRCFailed

cfgLLData = 
  bleLLDataChannelPDUConfig with properties:


llPayload =

  1x0 empty char array
```

Input Arguments

pdu — BLE LL data channel PDU

character vector | string scalar | numeric vector | character array | binary vector

BLE LL data channel PDU, specified as one of these values:

- Character vector — This vector represents octets in hexadecimal format.

- String scalar — This scalar represents octets in hexadecimal format.
- Numeric vector of elements in the range [0,255] — This vector represents octets in decimal format.
- `n-by-2` character array — Each row represents an octet in hexadecimal format.
- Binary vector — This vector represents BLE LL data channel PDU bits.

Data Types: `char` | `string` | `double`

`CRCinit` — CRC initialization value

character vector | string scalar

CRC initialization value, specified as a 6-element character vector or a string scalar representing 3-octet hexadecimal value.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `[status,cfgLLData,LLPayload] = bleLLDecodeDataChannelPDU('pdu','CRCInit','InputFormat','octets')`

`InputFormat` — BLE LL data channel PDU format

`'bits'` (default) | `'octets'`

BLE LL data channel PDU format, specified as `'bits'` or`'octets'`. When specified as `'bits'`, `InputFormat` is a binary vector. When specified as `'octets'`, `InputFormat` is a numeric vector representing octets in decimal format or a character array or a string scalar representing octets in hexadecimal format.

Data Types: `char` | `string` | `double`

Output Arguments

`status` — BLE LL data channel PDU decoding status

nonpositive integer

BLE LL data channel PDU decoding status, returned as a nonpositive number of type `blePacketDecodeStatus`. This value represents the result of a BLE LL data channel PDU decoding. Each value of `status` corresponds to a member of the `blePacketDecodeStatus` enumeration class, which indicates the packet decoding status as listed in this table.

Enumeration Value	Member of Enumeration Class	Decoding Status
0	`Success`	Packet decoding succeeded
-1	`CRCFailed`	Link Layer PDU is corrupted
–2	`LLPDULengthMismatch`	Length field does not match with actual PDU length
–3	`InvalidLLSlaveLatency`	Invalid slave latency

–4	`InvalidLLConnectionTimeout`	Invalid connection timeout
–5	`InvalidLLWindowSize`	Invalid window size
–6	`InvalidLLWindowOffset`	Invalid window offset
–7	`InvalidLLConnectionInterval`	Invalid connection interval
–8	`InvalidLLChannelMap`	Invalid channel map
-101	`IncompleteLLDataChannelPDU`	Insufficient octets in data channel PDU
-102	`InvalidLLID`	Invalid LLID
-103	`UnsupportedLLOpCode`	Unsupported opcode
-104	`InvalidLLErrorCode`	Invalid error code
-105	`InvalidBluetoothVersion`	Invalid version
-106	`ExpectedNonZeroPayload`	Nonzero payload expected
-107	`MICNotSupported`	Payload contains MIC

An enumeration value other than `0` means that the BLE LL data channel PDU decoding failed. If decoding fails, object `cfgLLData` displays no output.

`cfgLLData` — BLE LL data channel configuration object
`bleLLDataChannelPDUConfig` object

BLE LL data channel configuration object, returned as a `bleLLDataChannelPDUConfig` object. This value represents the decoded BLE LL data channel PDU configuration.

`LLPayload` — Upper-layer payload
array

Upper-layer payload, returned as a character array where each row is the hexadecimal representation of an octet.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When `CRCInitialization` property of the `bleLLDataChannelPDUConfig` object is set to constant, code generation is possible.

See Also

Functions
`bleLLDataChannelPDU`

Objects
`bleLLDataChannelPDUConfig` | `bleLLControlPDUConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2019b

bluetoothIdealReceiver

Ideal receiver for Bluetooth BR/EDR PHY waveform

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[bits,decodedInfo] = bluetoothIdealReceiver(waveform,rxConfig)
[ ___ ,pktValidStatus,decodedCRC] = bluetoothIdealReceiver( ___ )
```

Description

`[bits,decodedInfo] = bluetoothIdealReceiver(waveform,rxConfig)` demodulates and decodes a synchronized time-domain Bluetooth waveform, `waveform`, generated by the `bluetoothWaveformGenerator` function for a given system configuration object, `rxConfig`. The function returns the decoded payload bits, `bits`, and decoded information, `decodedInfo`.

`[ ___ ,pktValidStatus,decodedCRC] = bluetoothIdealReceiver( ___ )` returns a flag, `pktValidStatus`, to indicate the validity of the received Bluetooth BR/EDR packet. The function also returns the decoded cyclic redundancy check (CRC), `decodedCRC`, of the received Bluetooth packet.

Examples

Demodulate and Decode Time-Domain Bluetooth BR/EDR Waveform

Demodulate and decode time-domain Bluetooth BR/EDR waveform using two different ways to get physical layer (PHY) information. The `bluetoothIdealReceiver` function uses `bluetoothPhyConfig` object or `bluetoothWaveformConfig` object to extract PHY information and decode the Bluetooth BR/EDR waveform.

Use the `bluetoothPhyConfig` object to get PHY information

Create a Bluetooth BR/EDR waveform configuration object with default settings.

```
txconfig = bluetoothWaveformConfig;
```

Create a random input bit vector to generate the payload. Generate the time-domain Bluetooth BR/EDR waveform using the payload.

```
dataBits = randi([0 1],getPayloadLength(txconfig)*8,1);
waveform = bluetoothWaveformGenerator(dataBits,txconfig);
```

Create a configuration object for Bluetooth BR/EDR PHY with default settings. This object sends the PHY information to the ideal receiver.

```
rxConfig = bluetoothPhyConfig;
```

Demodulate and decode the Bluetooth BR/EDR waveform. The generated output displays the decoded bits and a structure containing the decoded information.

```
[bits,decodedInfo] = bluetoothIdealReceiver(waveform,rxConfig)
```

```
bits = 144×1

     1
     1
     0
     1
     1
     0
     0
     1
     1
     1
       ⋮


decodedInfo = struct with fields:
                       LAP: [24x1 double]
                PacketType: 'FHS'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
              PayloadLength: 18
                       LLID: [2x1 double]
              FlowIndicator: 0
```

Use the bluetoothWaveformConfig object to get PHY information

Create a Bluetooth BR/EDR waveform configuration object with default settings. Set the packet type to 'DM1'and payload length to 10.

```
cfg = bluetoothWaveformConfig;
cfg.PacketType = 'DM1';
cfg.PayloadLength = 10;
```

Create a random input bit vector to generate the payload.

```
numBits = getPayloadLength(cfg)*8;
dataBits = randi([0 1],numBits,1);
```

Generate the time-domain Bluetooth BR/EDR waveform using the payload.

```
waveform = bluetoothWaveformGenerator(dataBits,cfg);
```

Get PHY information from the `bluetoothWaveformConfig` object function, `getPhyConfigProperties`.

```
rxConfig = getPhyConfigProperties(cfg);
```

Demodulate and decode the Bluetooth BR/EDR waveform. The generated output displays the decoded bits, a structure containing the decoded information, the packet status, and the decoded CRC.

```
[bits,decodedInfo,pktStatus,crc] = bluetoothIdealReceiver(waveform,rxConfig)
```

```
bits = 80×1
```

```
     0
     0
     0
     0
     0
     0
     0
     0
     1
     1
      ⋮


decodedInfo = struct with fields:
                        LAP: [24x1 double]
                 PacketType: 'DM1'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
              PayloadLength: 10
                       LLID: [2x1 double]
              FlowIndicator: 1


pktStatus = logical
   1


crc = 16×1

     1
     1
     1
     1
     1
     1
     0
     0
     0
     0
      ⋮
```

Input Arguments

`waveform` — Synchronized time-domain Bluetooth waveform
complex-valued column vector

Synchronized time-domain Bluetooth waveform, specified as a complex-valued column vector.

Data Types: `double`
Complex Number Support: Yes

`rxConfig` — System configuration object
`bluetoothPHYConfig` object

System configuration object, specified as a `bluetoothPhyConfig` object. The function uses this value to set its configuration parameters.

Output Arguments

bits — Decoded payload bits

binary-valued column vector

Decoded payload bits, returned as a binary-valued column vector.

Data Types: double

decodedInfo — Decoded information

structure

Decoded information, returned as a structure containing these fields:

Field	Value	Description
PacketType	'ID', 'NULL', 'POLL', 'FHS', 'HV1', 'HV2', 'HV3', 'DV', 'EV3', 'EV4', 'EV5', 'AUX1', 'DM3', 'DM1', 'DH1', 'DM5', 'DH3', 'DH5', '2-DH1', '2-DH3', '2-DH5', '2-DH1', '2-DH3', '2-DH5', '2-EV3', '2-EV5', '3-EV3', or '3-EV5'	Type of received Bluetooth BR/EDR packet
LAP	24-bit column vector of type double.	Decoded lower address part (LAP) of the Bluetooth device address
PayloadLength	Scalar of type double	Number of payload bytes in the received Bluetooth BR/EDR packet
LogicalTransportAddress	3-bit vector of type double	Active destination slave for a Bluetooth BR/EDR packet in a master-to-slave transmission slot
HeaderControlBits	3-bit vector of type double	Link control information containing flow control information (FLOW), acknowledgement for successfully receiving a Bluetooth BR/EDR packet payload (ARQN), and sequencing scheme for received packets (SEQN) bits
LLID	2-bit binary vector of type double	Logical link identifier. This field is applicable only if the value of PacketType field is one of these: 'DM1', 'DH1', 'DM3', 'DH3', 'DM5', 'DH5', 'AUX1', 'DV', '2-DH1', '2-DH3', '2-DH5', '3-DH1', '3-DH3', or '3-DH5'.

Field	Value	Description
FlowIndicator	Scalar of type double	Control data flow indicator over logical channels. This field is applicable only if the value of PacketType field is one of these: 'DM1', 'DH1', 'DM3', 'DH3', 'DM5', 'DH5', 'AUX1', 'DV', '2-DH1', '2-DH3', '2-DH5', '3-DH1', '3-DH3', or '3-DH5'

Data Types: struct

pktValidStatus — Flag indicating validity of received Bluetooth BR/EDR packet
1 | 0

Flag indicating validity of received Bluetooth BR/EDR packet, returned as 1 or 0 of data type logical. Validity is based on the Bluetooth BR/EDR packet header error check (HEC) and cyclic redundancy check (CRC). The value of this argument is true only when both, HEC and CRC are true.

Dependencies

To enable this output argument, set the PacketType field value of the “decodedInfo” on page 2-0 output argument to any one of these: 'NULL', 'POLL', 'FHS', 'HV1', 'HV2', 'HV3', 'DV', 'EV3', 'EV4', 'EV5', 'AUX1', 'DM3', 'DM1', 'DH1', 'DM5', 'DH3', 'DH5', '2-DH1', '2-DH3', '2-DH5', '2-DH1', '2-DH3', '2-DH5', '2-EV3', '2-EV5', '3-EV3', or '3-EV5'.

Data Types: logical

decodedCRC — Decoded CRC
16-bit binary-valued column vector

Decoded CRC, specified as the CRC of the received Bluetooth BR/EDR packet.

Dependencies

To enable this output argument, set the packet type field value of the decoded information output argument to any one of these: 'FHS', 'DV', 'EV3', 'EV4', 'EV5', 'DM3', 'DM1', 'DH1', 'DM5', 'DH3', 'DH5', '2-DH1', '2-DH3', '2-DH5', '2-DH1', '2-DH3', '2-DH5', '2-EV3', '2-EV5', '3-EV3', or '3-EV5'.

Data Types: double

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Properties must be given as: coder. Constant().

See Also

Functions
`bleIdealReceiver` | `bleWaveformGenerator` | `bluetoothWaveformGenerator`

Objects
`bluetoothPhyConfig` | `bluetoothWaveformConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2020a

bluetoothWaveformGenerator

Waveform generator for Bluetooth BR/EDR PHY

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
waveform = bluetoothWaveformGenerator(data,cfgFormat)
```

Description

`waveform = bluetoothWaveformGenerator(data,cfgFormat)` generates `waveform`, a multipacket time-domain Bluetooth BR/EDR waveform, for input information bits `data` and a given format configuration `cfgFormat`.

Examples

Generate Multipacket Bluetooth BR/EDR Waveform with HV1 Packets

Specify the number of HV1 packets.

```
numPackets = 10;
```

Create a Bluetooth BR/EDR waveform configuration object. Specify the packet type as HV1.

```
cfg = bluetoothWaveformConfig;
cfg.PacketType = 'HV1';
```

Create a random input bit vector containing concatenated payloads.

```
numBits = getPayloadLength(cfg)*8*numPackets; % Byte to bit conversion
dataBits = randi([0 1],numBits,1);
```

Set the symbol rate.

```
symbolRate = 1e6; % In MHz
```

Generate the Bluetooth waveform.

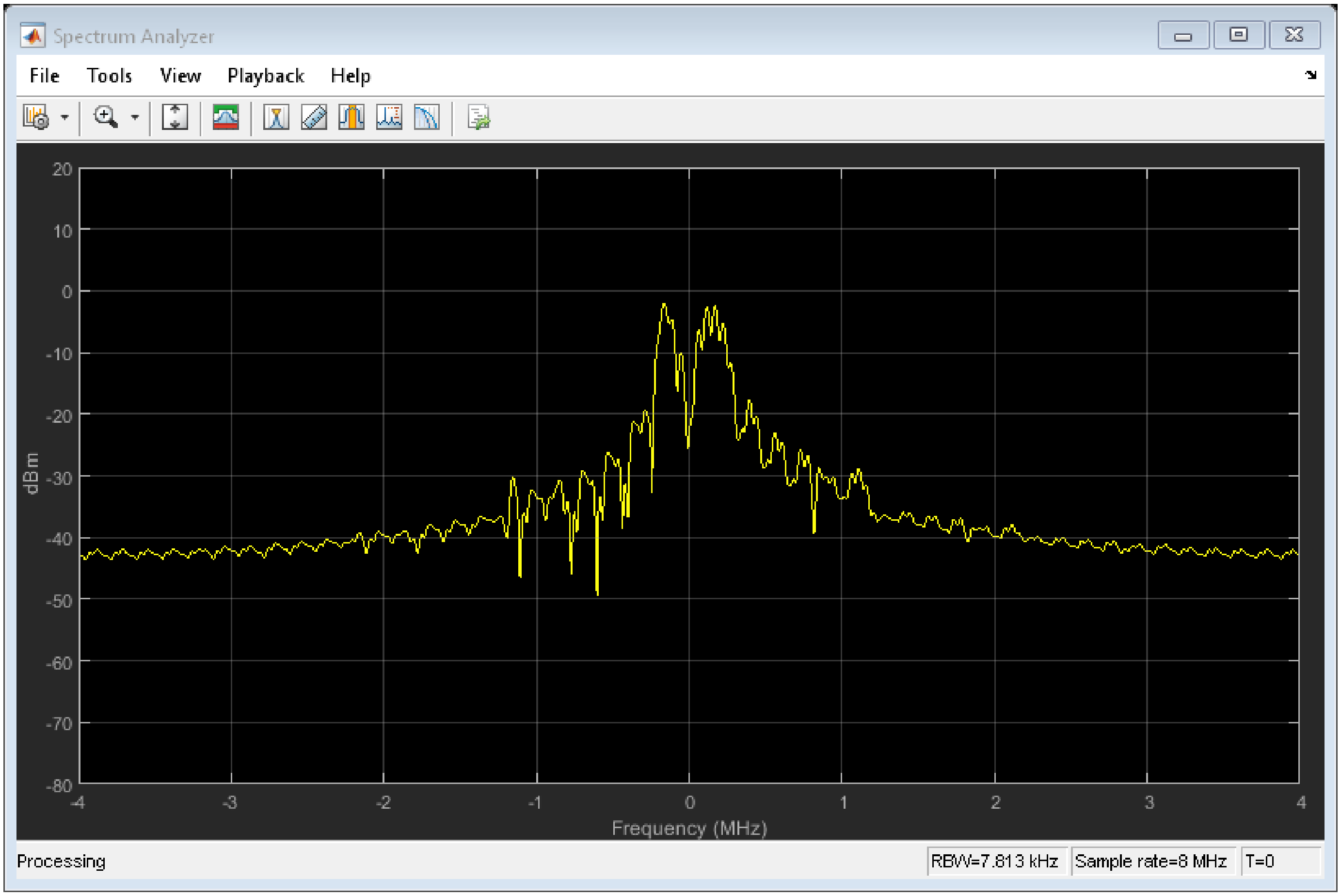
```
waveform = bluetoothWaveformGenerator(dataBits,cfg);
```

Create a `dsp.SpectrumAnalyzer` System object to display the frequency spectrum of the generated Bluetooth BR/EDR waveform. Set the sample rate of the frequency spectrum.

```
scope = dsp.SpectrumAnalyzer;
scope.SampleRate = cfg.SamplesPerSymbol*symbolRate;
```

Plot the Bluetooth BR/EDR waveform.

```
scope(waveform);
```



Generate Bluetooth BR/EDR Waveform for Single Enhanced Data Rate Packet

Create a Bluetooth BR/EDR waveform configuration object.

```
cfg = bluetoothWaveformConfig;
```

To generate enhanced data rate packet 2-EV3, specify the packet type as EV3 and the transmission mode as EDRM2.

```
cfg.PacketType = 'EV3';
cfg.Mode = 'EDR2M';
```

Create a random input bit vector to generate the payload for a single packet.

```
numBits = getPayloadLength(cfg)*8; % Byte to bit conversion
dataBits = randi([0 1],numBits,1);
```

Generate the Bluetooth BR/EDR waveform.

```
txWaveform = bluetoothWaveformGenerator(dataBits,cfg);
```

Input Arguments

`data` — Input information bits

binary-valued column vector

Input information bits, specified as a binary-valued column vector with data type `double`. This value represents multiple concatenated payloads. The length of `data` must be an exact multiple of the payload length derived from the `getPayloadLength` object function for the `bluetoothWaveformConfig` object.

Data Types: `double`

`cfgFormat` — Format configuration object

`bluetoothWaveformConfig` object

Format configuration object, specified as a `bluetoothWaveformConfig` object. The function uses this value to set its configuration parameters.

Output Arguments

`waveform` — Generated time-domain Bluetooth BR/EDR waveform

complex-valued column vector

Generated time-domain Bluetooth BR/EDR waveform, returned as a complex-valued column vector containing the generated Bluetooth BR/EDR waveform. The function appends this value with zero samples to accommodate a packet-specific slot duration.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Properties must be given as: coder.Constant().

See Also

Functions

`bleIdealReceiver` | `bleWaveformGenerator` | `bluetoothIdealReceiver`

Objects

`bluetoothPhyConfig` | `bluetoothWaveformConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2020a

bleWaveformGenerator

Waveform generator for BLE PHY

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
waveform = bleWaveformGenerator(message)
waveform = bleWaveformGenerator(message,Name,Value)
```

Description

`waveform = bleWaveformGenerator(message)` generates `waveform`, a time-domain Bluetooth low energy (BLE) physical layer (PHY) waveform by using the input information bits `message`.

`waveform = bleWaveformGenerator(message,Name,Value)` also specifies options using one or more name-value pair arguments. For example, `'Mode','LE2M'` specifies the generating mode value of the desired BLE waveform.

Examples

Generate BLE Waveform Using Default Settings

Create an input message column vector of length 2056 containing random binary values. Set the symbol rate to default value.

```
message = randi([0 1],2056,1);
symbolRate = 1e6;
```

Generate the BLE waveform.

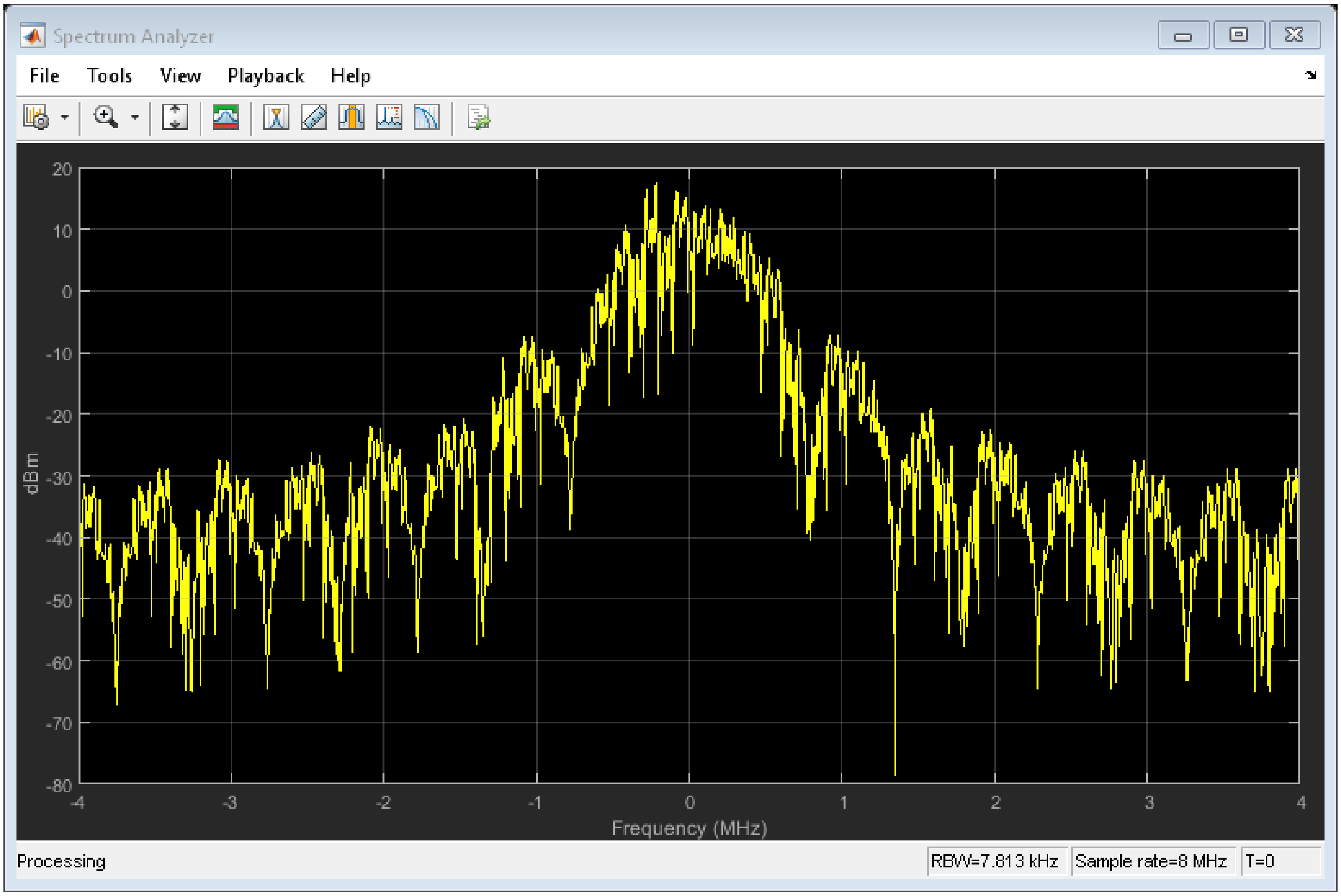
```
waveform = bleWaveformGenerator(message);
```

Create a `dsp.SpectrumAnalyzer` System object to display the frequency spectrum of the generated BLE waveform. Set the sample rate of the frequency spectrum.

```
scope = dsp.SpectrumAnalyzer;
scope.SampleRate = 8*symbolRate;
```

Plot the BLE waveform.

```
scope(waveform);
```



Generate BLE Waveform Using Specified Name-Value Pairs

Create an input message column vector of length 640 containing random binary values.

```
message = randi([0 1],640,1);
```

Specify the values of generating mode, channel index, samples per symbol and access address. Set symbol rate to default value.

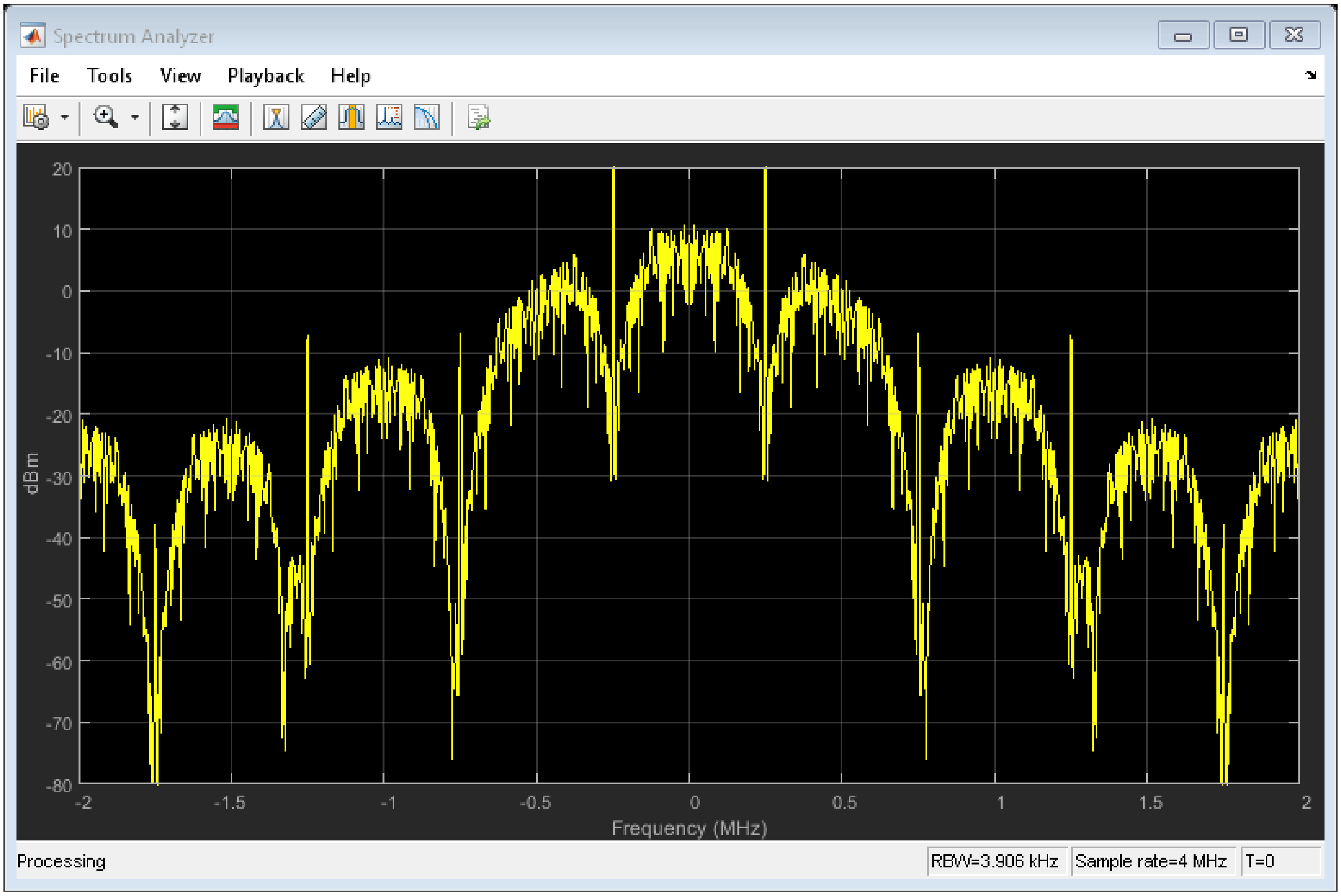
```
phyMode = 'LE125K';
chanIdx = 2;
sps = 4;
accAdd = [1 1 1 1 0 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 ...
    0 1 0 1 1 0 0].';
symbolRate = 1e6;
```

Create a `dsp.SpectrumAnalyzer` System object to display the frequency spectrum of the generated BLE waveform. Set the sample rate of the frequency spectrum.

```
scope = dsp.SpectrumAnalyzer;
scope.SampleRate = sps*symbolRate;
```

Generate the BLE waveform using specified name-value pair arguments. Plot the BLE waveform.

```
waveform = bleWaveformGenerator(message,'Mode',phyMode, ...
    'SamplesPerSymbol',sps,'ChannelIndex',chanIdx,'AccessAddress',accAdd);
scope(waveform);
```



Generate BLE Waveform With Constant Tone Extension For Connection-Oriented Scenario

Specify a connection data channel protocol data unit (PDU) for angle of departure (AoD) constant tone extension (CTE).

```
pduHex = '1B820127'; % Valid PDU in hexadecimal
pdu = de2bi(hex2dec(pduHex), 32)';
```

Generate and append cyclic redundancy check (CRC) to the PDU.

```
crcGen = comm.CRCGenerator('z^24+z^10+z^9+z^6+z^4+z^3+z+1',...
    'InitialConditions',de2bi(hex2dec('555551'),'left-msb',24),...
    'DirectMethod',true);
pduCRC = crcGen(pdu);
```

Generate the BLE waveform using specified name-value pair arguments.

```
waveform = bleWaveformGenerator(pduCRC,'ChannelIndex',36,...
    'DFPacketType','ConnectionCTE')
```

```
waveform = 896×1 complex

   1.0000 + 0.0000i
   0.9892 + 0.1469i
   0.9441 + 0.3297i
   0.8507 + 0.5257i
   0.7071 + 0.7071i
   0.5257 + 0.8507i
   0.3297 + 0.9441i
   0.1469 + 0.9892i
   0.0000 + 1.0000i
   0.1469 + 0.9892i
      ⋮
```

Input Arguments

`message` — Input message bits

binary-valued column vector

Input message bits, specified as a binary-valued column vector of numerical or logical values. This message contains the protocol data unit (PDU) and cyclic redundancy check (CRC) data. The maximum length of this value is 2080 bits.

Data Types: `double` | `logical`

Name-Value Pair Arguments

Note For information about connection CTE and connectionless CTE direction finding packet generation, see “Bluetooth Packet Structure”.

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `bleWaveformGenerator(message,'Mode','LE2M','ChannelIndex',36)`

`Mode` — Generating mode

`'LE1M'` (default) | `'LE2M'` | `'LE500K'` | `'LE125K'`

Generating mode, specified as the comma-separated pair consisting of `'Mode'` and `'LE1M'`, `'LE2M'`, `'LE500K'`, or `'LE125K'`. This value indicates the type of BLE waveform.

Data Types: `string` | `char`

`ChannelIndex` — Channel Index

`37` (default) | integer in the range [0, 39]

Channel index, specified as the comma-separated pair consisting of `'ChannelIndex'`and an integer in the range [0, 39]. For data channels, this value must be in the range [0, 36]. This value is used by the data-whitening block to randomize the bits.

Data Types: `single` | `double`

`SamplesPerSymbol` — Samples per symbol

8 (default) | positive integer

Samples per symbol, specified as the comma-separated pair consisting of `'SamplesPerSymbol'` and a positive integer. This value is used for the Gaussian frequency-shift keying (GFSK) modulation.

Data Types: `single` | `double`

`AccessAddress` — Access address

`[0 1 1 0 1 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1]'` (default) | 32-bit column vector

Access address, specified as the comma-separated pair consisting of `'AccessAddress'` and a 32-bit column vector of numerical or logical values.

Data Types: `logical` | `single` | `double`

`DFPacketType` — Type of direction finding packet

`'Disabled'` (default) | `'ConnectionlessCTE'` | `'ConnectionCTE'`

Type of direction finding packet, specified as the comma-separated pair consisting of `'DFPacketType'` and `'ConnectionlessCTE'`, `'ConnectionCTE'`, or `'Disabled'`.

Data Types: `string` | `char`

Output Arguments

`waveform` — Output time-domain waveform

complex-valued column vector

Output time-domain waveform, returned as a complex-valued column vector of size *Ns*-by-1, where *Ns* represents the number of time-domain samples. The `waveform` is generated in the form of complex in-phase quadrature (IQ) samples.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleIdealReceiver` | `bluetoothIdealReceiver` | `bluetoothWaveformGenerator` | `bleAngleEstimate`

Objects
`bleAngleEstimateConfig`

Apps
Wireless Waveform Generator

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”
“Bluetooth Location and Direction Finding”

Introduced in R2019b

getPayloadLength

Payload length in bytes for Bluetooth BR/EDR format configuration

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
payloadLength = getPayloadLength(cfg)
```

Description

`payloadLength = getPayloadLength(cfg)` returns the payload length, in bytes, for the Bluetooth BR/EDR format configuration, `cfg`.

Examples

Get Payload Length for Bluetooth BR/EDR Format Configuration

Create a Bluetooth BR/EDR waveform configuration object with default properties.

```
cfg = bluetoothWaveformConfig
```

```
cfg = 
  bluetoothWaveformConfig with properties:

                      Mode: 'BR'
                PacketType: 'FHS'
             DeviceAddress: '0123456789AB'
   LogicalTransportAddress: [3x1 double]
         HeaderControlBits: [3x1 double]
           ModulationIndex: 0.3200
          SamplesPerSymbol: 8
              WhitenStatus: 'On'
      WhitenInitialization: [7x1 double]
```

Get the payload length of the default `'FHS'` packet.

```
payloadLength = getPayloadLength(cfg)
```

```
payloadLength = 18
```

Create another Bluetooth BR/EDR waveform configuration object. Set the packet type as `'HV1'`.

```
cfgHV1 = bluetoothWaveformConfig('PacketType','HV1');
```

Get the payload length of the specified `'HV1'` packet.

```
payloadLength = getPayloadLength(cfgHV1)
```

```
payloadLength = 10
```

Input Arguments

`cfg` — Bluetooth BR/EDR format configuration
`bluetoothWaveformConfig` object

Bluetooth BR/EDR format configuration, specified as `bluetoothWaveformConfig` object.

Output Arguments

`payloadLength` — Payload length of packet
18 (default) | nonnegative integer

Payload length of packet, returned as a nonnegative integer. This value indicates the number of bytes to be processed in a packet.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bluetoothIdealReceiver` | `bluetoothWaveformGenerator`

Objects
`bluetoothPhyConfig` | `bluetoothWaveformConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2020a

getPhyConfigProperties

Updated configuration properties of Bluetooth BR/EDR PHY configuration object

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
cfgPHY = getPhyConfigProperties(cfg)
```

Description

`cfgPHY = getPhyConfigProperties(cfg)` returns updated Bluetooth BR/EDR PHY configuration properties, `cfgPHY` for the Bluetooth BR/EDR waveform format configuration object, `cfg`.

Examples

Get PHY Configuration Properties of Bluetooth BR/EDR PHY Configuration Object

Create a Bluetooth BR/EDR waveform configuration object with default properties.

```
cfg = bluetoothWaveformConfig
```

```
cfg = 
  bluetoothWaveformConfig with properties:

                     Mode: 'BR'
               PacketType: 'FHS'
            DeviceAddress: '0123456789AB'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
            ModulationIndex: 0.3200
           SamplesPerSymbol: 8
               WhitenStatus: 'On'
       WhitenInitialization: [7x1 double]
```

Get the PHY configuration properties for the created Bluetooth BR/EDR waveform configuration object.

```
cfgPHY = getPhyConfigProperties (cfg)
```

```
cfgPHY = 
  bluetoothPhyConfig with properties:

                 Mode: 'BR'
        DeviceAddress: '0123456789AB'
      ModulationIndex: 0.3200
     SamplesPerSymbol: 8
```

```
            WhitenStatus: 'On'
    WhitenInitialization: [7x1 double]
```

Create another Bluetooth BR/EDR waveform configuration object and specify the PHY transmission mode as `'EDR2M'`.

```
cfgEDR = bluetoothWaveformConfig('Mode','EDR2M');
```

Get the PHY configuration properties for this Bluetooth BR/EDR waveform configuration object.

```
cfgPHY = getPhyConfigProperties (cfgEDR)
```

```
cfgPHY = 
  bluetoothPhyConfig with properties:

                    Mode: 'EDR2M'
           DeviceAddress: '0123456789AB'
         ModulationIndex: 0.3200
        SamplesPerSymbol: 8
            WhitenStatus: 'On'
    WhitenInitialization: [7x1 double]
```

Input Arguments

`cfg` — Bluetooth BR/EDR format configuration

`bluetoothWaveformConfig` object

Bluetooth BR/EDR Format configuration, specified as `bluetoothWaveformConfig` object.

Output Arguments

`cfgPHY` — Configuration object for Bluetooth BR/EDR PHY

`bluetoothPHYConfig` object

Configuration object for Bluetooth BR/EDR PHY, returned as a `bluetoothPhyConfig` object.

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bluetoothIdealReceiver` | `bluetoothWaveformGenerator`

Objects
`bluetoothPhyConfig` | `bluetoothWaveformConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2020a

addCustomBasemap

Add custom basemap

Syntax

```
addCustomBasemap(basemapName,URL)
addCustomBasemap( ___ ,Name,Value)
```

Description

`addCustomBasemap(basemapName,URL)` adds the custom basemap specified by `URL` to the list of basemaps available for use with mapping functions. `basemapName` is the name you choose to call the custom basemap. Added basemaps remain available for use in future MATLAB sessions.

`addCustomBasemap( ___ ,Name,Value)` specifies name-value pairs that set additional parameters of the basemap.

Examples

Add and Remove a Custom Basemap

Add a custom basemap to view locations on an OpenTopoMap® basemap, then remove the custom basemap from `siteviewer`.

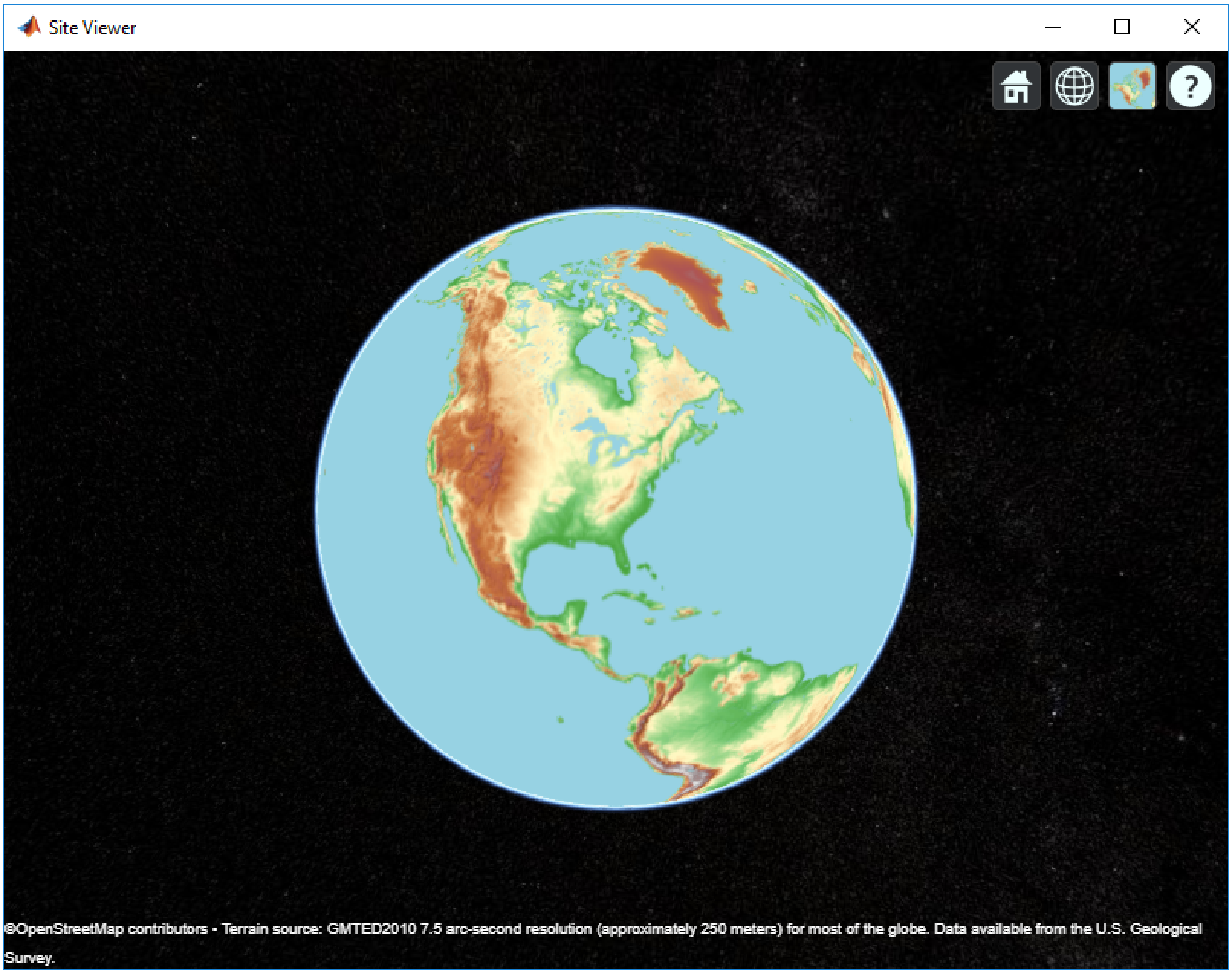
Initialize simulation variables to:

- Define the name that you will use to specify your custom basemap.
- Specify the website that provides the map data. The first character of the URL indicates which server to use to get the data. For load balancing, the provider has three servers that you can use: a, b, or c.
- Create an attribution to display on the map that gives credit to the provider of the map data. Web map providers might define specific requirements for the attribution.
- Define a display name for the custom map.

```
name = 'opentopomap';
url = 'a.tile.opentopomap.org';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
displayName = 'Open Topo Map';
```

Use `addCustomBasemap` to load the custom basemap, and then create a `siteviewer` object that loads the custom basemap.

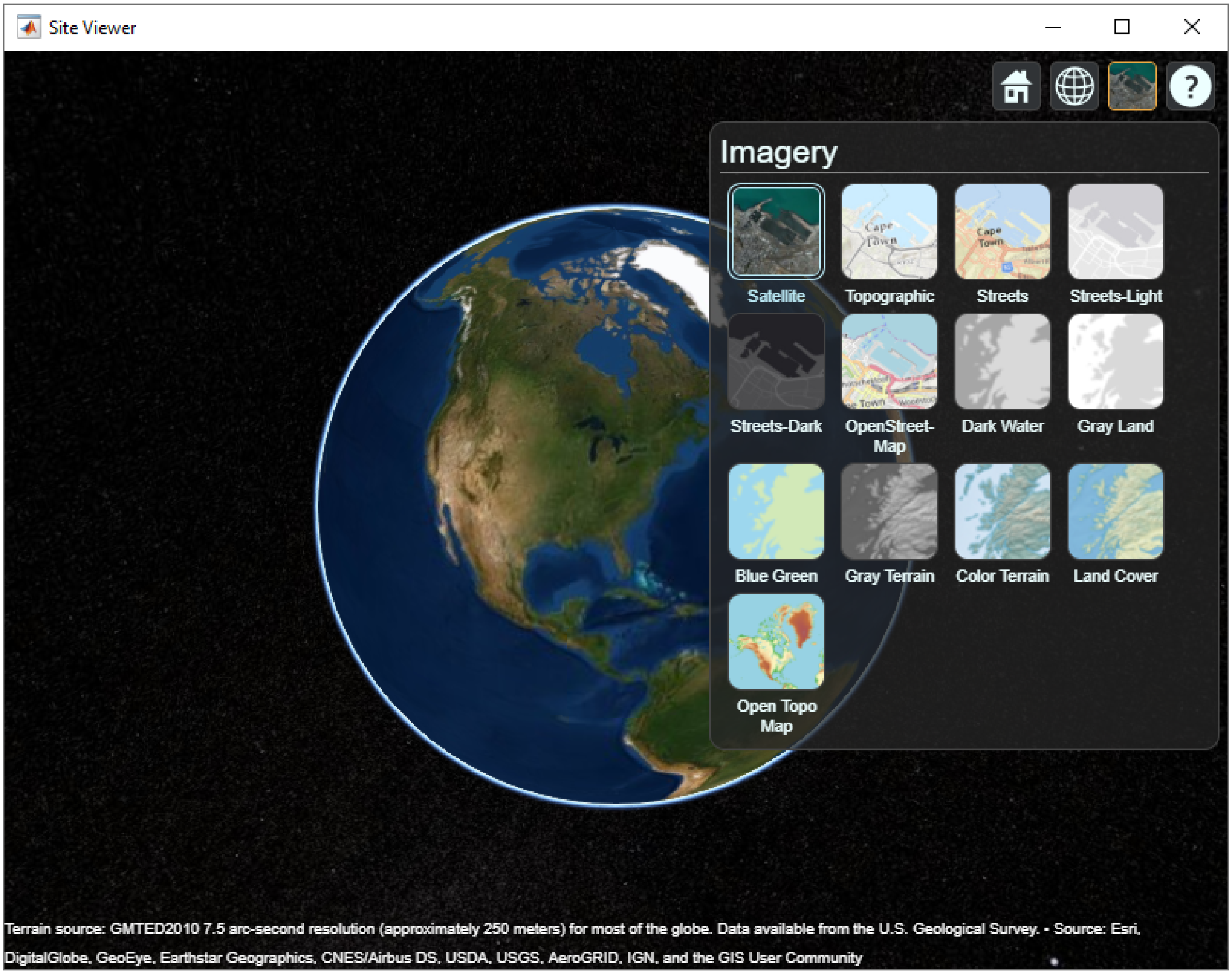
```
addCustomBasemap(name,url,'Attribution',attribution','DisplayName',displayName)
viewer = siteviewer('Basemap',name);
```



©OpenStreetMap contributors • Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey.

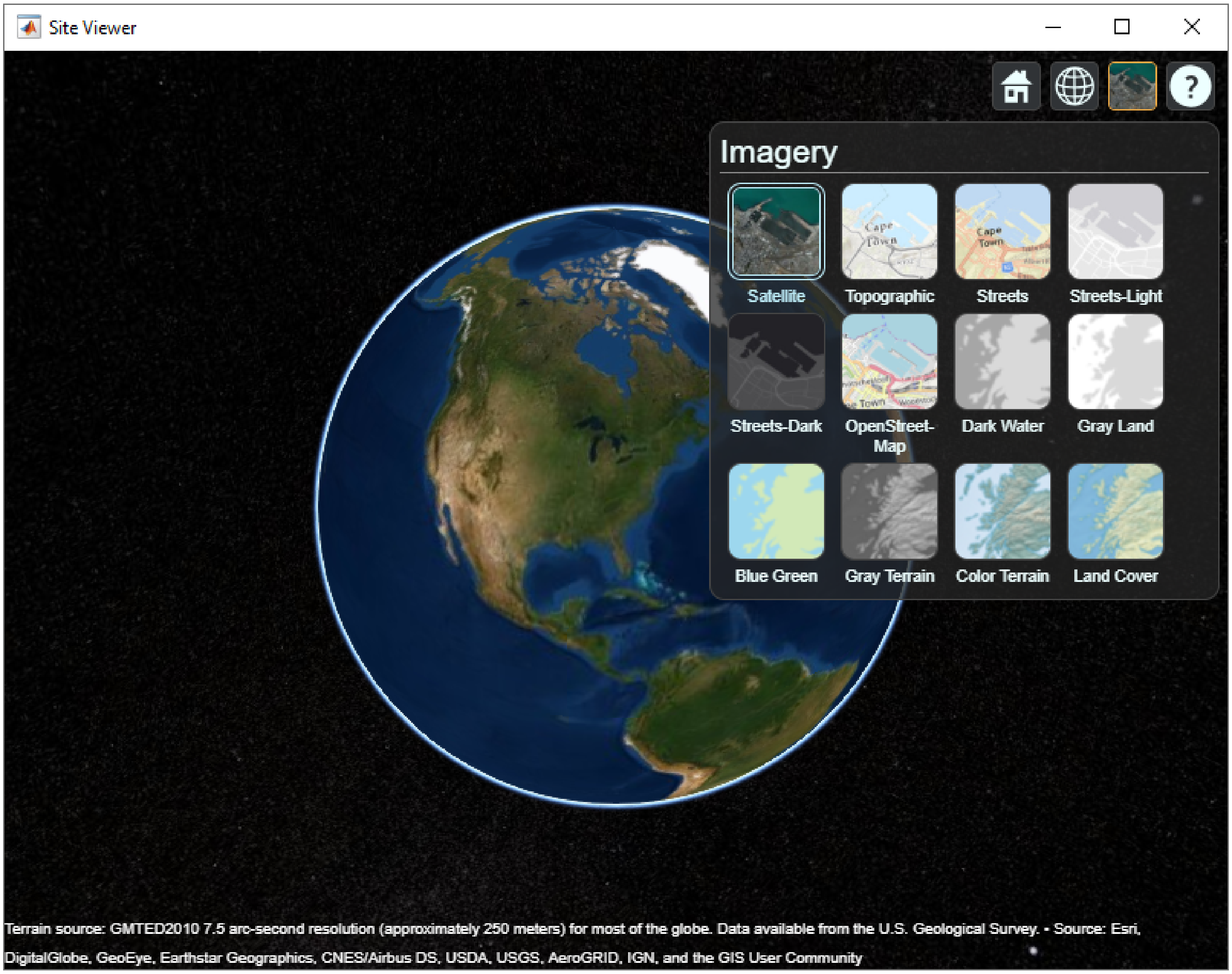
After a custom basemap is added to `siteviewer`, the custom map is available for future calls to `siteviewer`. Note the `'Open Topo Map'` icon in the `Imagery` tab.

```
siteviewer;
```


Site Viewer
Imagery
Satellite
Topographic
Streets
Streets-Light
Streets-Dark
OpenStreet-Map
Dark Water
Gray Land
Blue Green
Gray Terrain
Color Terrain
Land Cover
Open Topo Map
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Use `removeCustomBasemap` to remove the custom basemap from future calls to `siteviewer`. Note the `'Open Topo Map'` icon is no longer available in the `Imagery` tab.

```
removeCustomBasemap(name)
siteviewer;
```


Site Viewer
Imagery
Satellite
Topographic
Streets
Streets-Light
Streets-Dark
OpenStreet-Map
Dark Water
Gray Land
Blue Green
Gray Terrain
Color Terrain
Land Cover
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

basemapName — Name used to identify basemap programmatically

string scalar | character vector

Name used to identify basemap programmatically, specified as a string scalar or character vector.

Example: `'openstreetmap'`

Data Types: `string` | `char`

URL — Parameterized map URL

string scalar | character vector

Parameterized map URL, specified as a string scalar or character vector. A parameterized URL is an index of the map tiles, formatted as `${z}/${x}/${y}.png` or `{z}/{x}/{y}.png`, where:

- `${z}` or `{z}` is the tile zoom level.
- `${x}` or `{x}` is the tile column index.
- `${y}` or `{y}` is the tile row index.

Example: `'https://hostname/${z}/${x}/${y}.png'`

Data Types: `string` | `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `addCustomBasemap(basemapName,URL,'Attribution',attribution)`

`Attribution` — Attribution of custom basemap

`'Tiles courtesy of` *`DOMAIN_NAME_OF_URL`*`'` (default) | string scalar | string array | character vector | cell array of character vectors

Attribution of custom basemap, specified as the comma-separated pair consisting of `'Attribution'` and a string scalar, string array, character vector, or cell array of character vectors. If the host is `'localhost'`, or if `URL` contains only IP numbers, specify an empty value (`''`). To create a multiline attribution, specify a string array or nonscalar cell array of character vectors.

If you do not specify an attribution, the default attribution is `'Tiles courtesy of` *`DOMAIN_NAME_OF_URL`*`'`, where the `addCustomBasemap` function obtains the domain name from the `URL` input argument.

Example: `'Credit: U.S. Geological Survey'`

Data Types: `string` | `char` | `cell`

`DisplayName` — Display name of custom basemap

string scalar | character vector

Display name of the custom basemap, specified as the comma-separated pair consisting of `'DisplayName'` and a string scalar or character vector.

Example: `'OpenStreetMap'`

Data Types: `string` | `char`

`MaxZoomLevel` — Maximum zoom level of basemap

`18` (default) | integer in the range [0, 25]

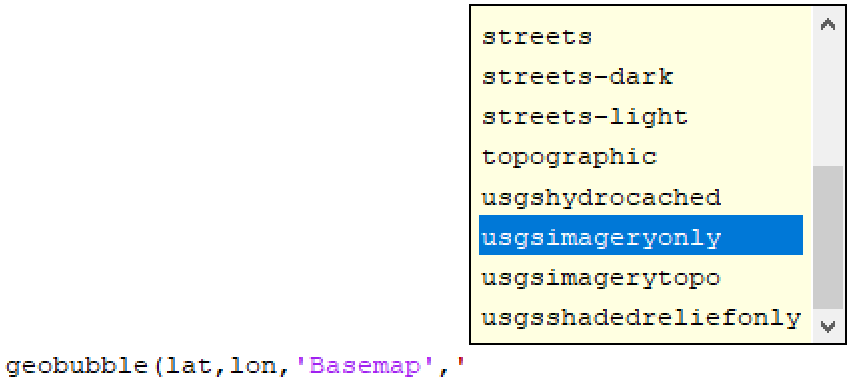
Maximum zoom level of the basemap, specified as the comma-separated pair consisting of `'MaxZoomLevel'` and an integer in the range [0, 25].

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

Tips

- You can find tiled web maps from various vendors, such as OpenStreetMap®, the USGS National Map, Mapbox, DigitalGlobe, Esri® ArcGIS Online, the Geospatial Information Authority of Japan (GSI), and HERE Technologies. Abide by the map vendors terms-of-service agreement and include accurate attribution with the maps you use.

- To access a list of available basemaps, press **Tab** before specifying the basemap in your plotting function.



```
geobubble(lat,lon,'Basemap','
```

See Also

geoaxes | geobasemap | geobubble | removeCustomBasemap

removeCustomBasemap

Remove custom basemap

Syntax

```
removeCustomBasemap(basemapName)
```

Description

`removeCustomBasemap(basemapName)` removes the custom basemap specified by `basemapName` from the list of available basemaps.

Examples

Add and Remove a Custom Basemap

Add a custom basemap to view locations on an OpenTopoMap® basemap, then remove the custom basemap from `siteviewer`.

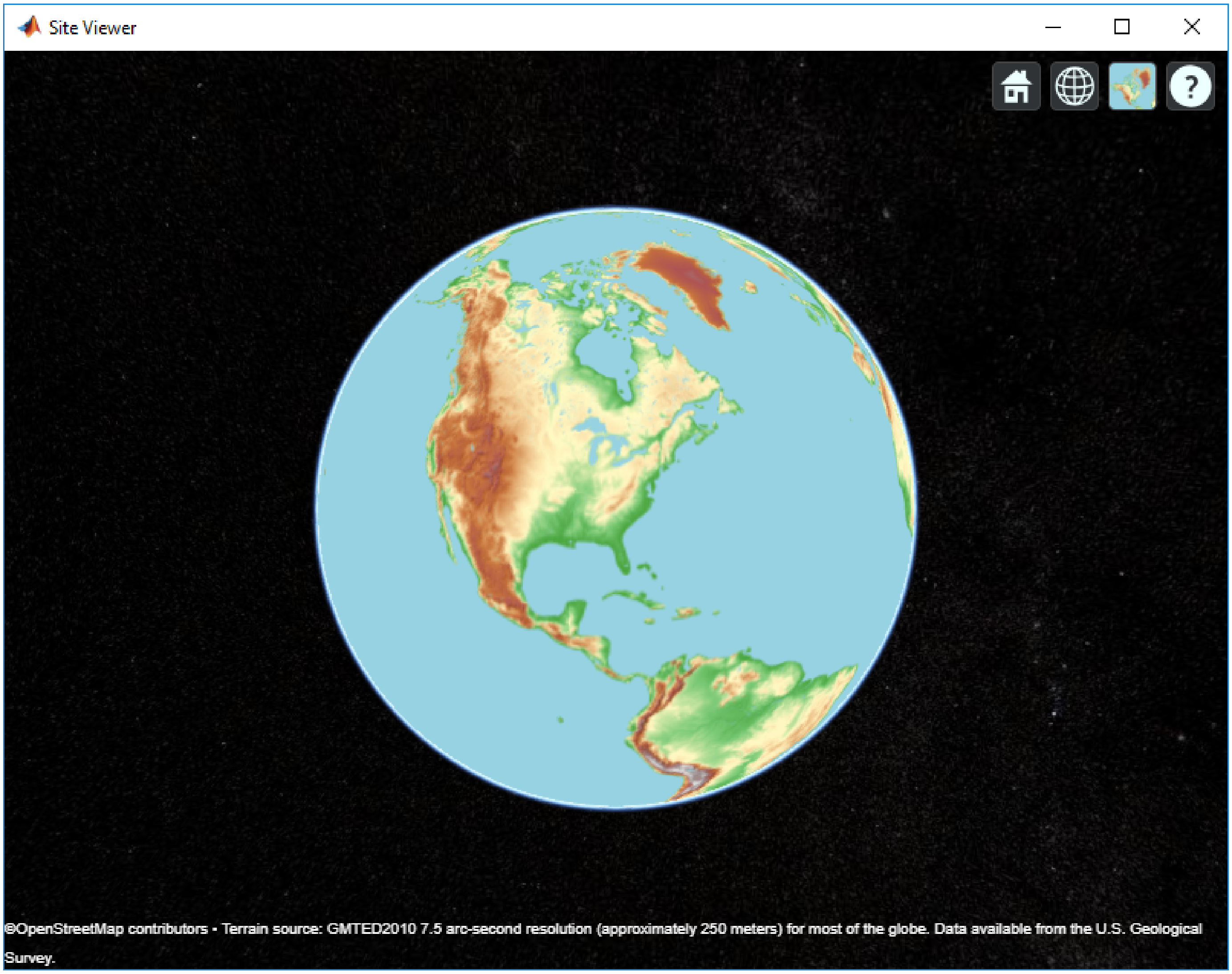
Initialize simulation variables to:

- Define the name that you will use to specify your custom basemap.
- Specify the website that provides the map data. The first character of the URL indicates which server to use to get the data. For load balancing, the provider has three servers that you can use: a, b, or c.
- Create an attribution to display on the map that gives credit to the provider of the map data. Web map providers might define specific requirements for the attribution.
- Define a display name for the custom map.

```
name = 'opentopomap';
url = 'a.tile.opentopomap.org';
copyright = char(uint8(169));
attribution = copyright + "OpenStreetMap contributors";
displayName = 'Open Topo Map';
```

Use `addCustomBasemap` to load the custom basemap, and then create a `siteviewer` object that loads the custom basemap.

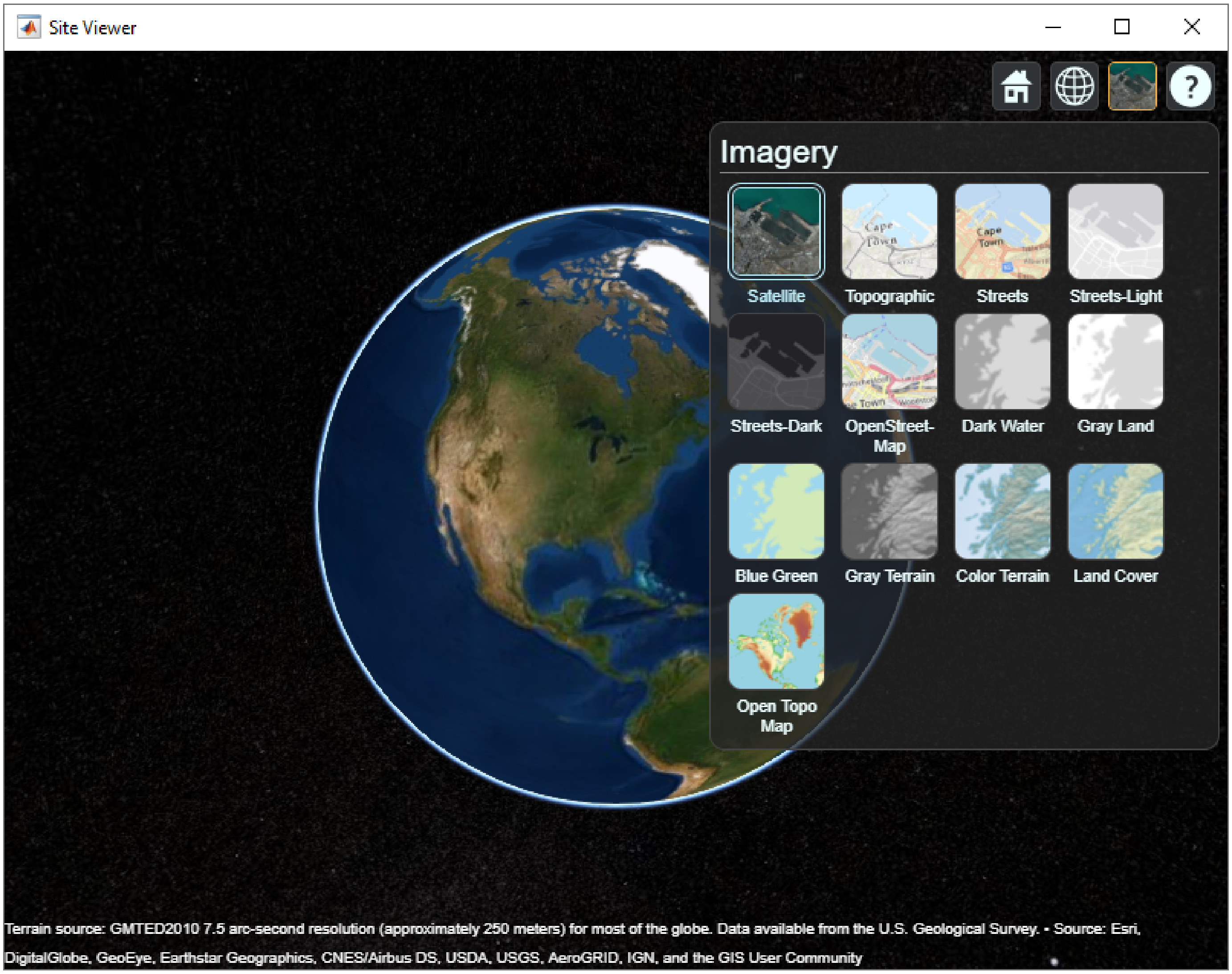
```
addCustomBasemap(name,url,'Attribution',attribution','DisplayName',displayName)
viewer = siteviewer('Basemap',name);
```



©OpenStreetMap contributors • Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey.

After a custom basemap is added to `siteviewer`, the custom map is available for future calls to `siteviewer`. Note the `'Open Topo Map'` icon in the `Imagery` tab.

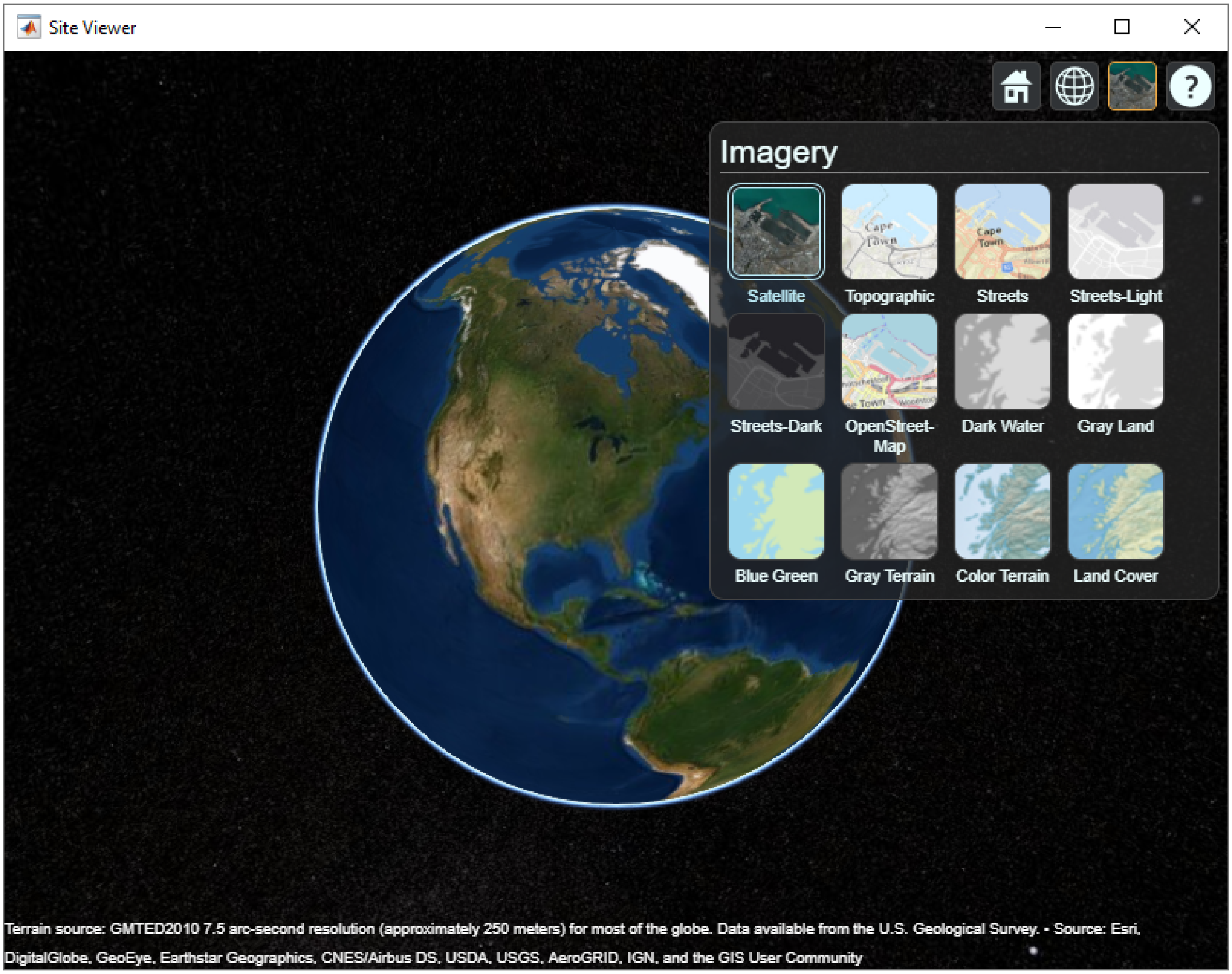
```
siteviewer;
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Use `removeCustomBasemap` to remove the custom basemap from future calls to `siteviewer`. Note the `'Open Topo Map'` icon is no longer available in the `Imagery` tab.

```
removeCustomBasemap(name)
siteviewer;
```


Site Viewer
Imagery
Satellite
Topographic
Streets
Streets-Light
Streets-Dark
OpenStreet-Map
Dark Water
Gray Land
Blue Green
Gray Terrain
Color Terrain
Land Cover
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

basemapName — Name of custom basemap
string scalar | character vector

Name of the custom basemap to remove, specified as a string scalar or character vector. You define the basemap name when you add the basemap using the `addCustomBasemap` function.

Data Types: `string` | `char`

See Also

`addCustomBasemap` | `geoaxes` | `geobasemap` | `geobubble` | `geodensityplot` | `geoplot` | `geoscatter`

buildingMaterialPermittivity

Permittivity and conductivity of building materials

Syntax

```
[epsilon,sigma,complexepsilon] = buildingMaterialPermittivity(material,fc)
```

Description

`[epsilon,sigma,complexepsilon] = buildingMaterialPermittivity(material,fc)` calculates the relative permittivity, conductivity, and complex relative permittivity for the specified material at the specified frequency. The methods and equations modeled in the `buildingMaterialPermittivity` function are presented in Recommendation ITU-R P.2040 [1].

Examples

Calculate Permittivity of Various Building Materials

Calculate relative permittivity and conductivity at 9 GHz for various building materials as defined by textual classifications in ITU-R P.2040, Table 3.

```
material = ["vacuum";"concrete";"brick";"plasterboard";"wood"; ...
    "glass";"ceiling-board";"chipboard";"floorboard";"metal"];
fc = repmat(9e9,size(material)); % Frequency in Hz
[permittivity,conductivity] = ...
    arrayfun(@(x,y)buildingMaterialPermittivity(x,y),material,fc);
```

Display the results in a table.

```
varNames = ["Material";"Permittivity";"Conductivity"];
table(material,permittivity,conductivity,'VariableNames',varNames)
```

```
ans=10×3 table
       Material        Permittivity    Conductivity
    _______________    ____________    ____________

    "vacuum"                  1                   0
    "concrete"             5.31             0.19305
    "brick"                3.75               0.038
    "plasterboard"         2.94            0.054914
    "wood"                 1.99            0.049528
    "glass"                6.27            0.059075
    "ceiling-board"         1.5           0.0064437
    "chipboard"            2.58             0.12044
    "floorboard"           3.66            0.085726
    "metal"                   1               1e+07
```

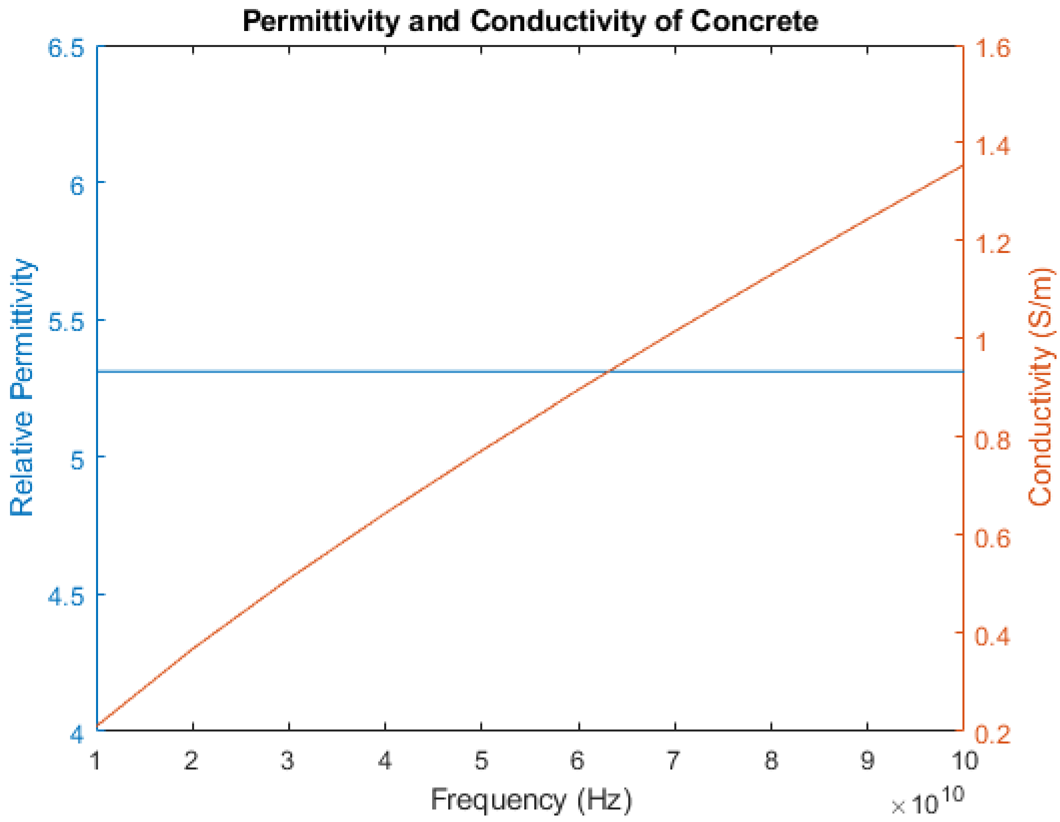
Plot Permittivity and Conductivity of Concrete at Various Frequencies

Calculate the relative permittivity and conductivity for concrete at frequencies specified.

```
fc = ((1:1:10)*10e9); % Frequency in Hz
[permittivity,conductivity] = ...
    arrayfun(@(y)buildingMaterialPermittivity("concrete",y),fc);
```

Plot the relative permittivity and conductivity of concrete across the range of frequencies.

```
figure
yyaxis left
plot(fc,permittivity)
ylabel('Relative Permittivity')
yyaxis right
plot(fc,conductivity)
ylabel('Conductivity (S/m)')
xlabel('Frequency (Hz)')
title('Permittivity and Conductivity of Concrete')
```



Input Arguments

`material` — Building material
`"vacuum"` | `"concrete"` | `"brick"` | `"plasterboard"` | ...

Building material, specified as vector of strings including one or more of these options:

`"vacuum"`	`"glass"`	`"very-dry-ground"`
`"concrete"`	`"ceiling-board"`	`"medium-dry-ground"`
`"brick"`	`"floorboard"`	`"wet-ground"`
`"plasterboard"`	`"chipboard"`	
`"wood"`	`"metal"`	

Example: `["vacuum" "brick"]`

Data Types: `char` | `string`

`fc` — Carrier frequency
positive scalar

Carrier frequency in Hz, specified as a positive scalar.

Note `fc` must be in the range [1e6, 10e6] when the `material` is `"very-dry-ground"`, `"medium-dry-ground"` or `"wet-ground"`.

Data Types: `double`

Output Arguments

`epsilon` — Relative permittivity
nonnegative scalar | nonnegative row vector

Relative permittivity of the building material, returned as a nonnegative scalar or row vector. The output dimension of `epsilon` matches that of the input argument `material`. For more information about the computation for the relative permittivity, see “ITU Building Materials” on page 2-967.

`sigma` — Conductivity
nonnegative scalar | nonnegative row vector

Conductivity, in Siemens/m, of the building material, returned as a nonnegative scalar or row vector. The output dimension of `sigma` matches that of the input argument `material`. For more information about the computation for the conductivity, see “ITU Building Materials” on page 2-967.

`complexepsilon` — Complex relative permittivity
complex scalar | row vector of complex values

Complex relative permittivity of the building material, returned as a complex scalar or row vector of complex values.

The output dimension of `complexepsilon` matches that of the input argument `material`. For more information about the computation for the complex relative permittivity, see “ITU Building Materials” on page 2-967.

More About

ITU Building Materials

Section 3 of ITU-R P.2040-1 [1] presents methods, equations, and values used to calculate real relative permittivity, conductivity, and complex relative permittivity at carrier frequencies up to 100 GHz for common building materials.

The `buildingMaterialPermittivity` function uses equations from ITU-R P.2040-1 to compute these values.

- The real part of the relative permittivity is calculated as
  `epsilon` = af^b.
  The computation of `epsilon` is based on equation (58). f is the frequency in GHz. Values for a and b are specified in Table 3 from ITU-R P.2040-1.
- The conductivity in Siemens/m is calculated as
  `sigma` = cf^d.
  The computation of `sigma` is based on equation (59). f is the frequency in GHz. Values for c and d are specified in Table 3 from ITU-R P.2040-1.
- The complex permittivity is calculated as
  `complexepsilon` = `epsilon` – 1*i* `sigma` / $(2\pi f c \varepsilon_0)$.
  The computation of `complexepsilon` is based on Equations (59) and (9b). f is the frequency in GHz. c is the velocity of light in free space. ε_0 = 8.854187817e-12 Farads/m, where ε_0 is the electric constant for the permittivity of free space.

For cases where the value of b or d is zero, the corresponding value of `epsilon` or `sigma` is a or c, respectively and independent of frequency.

The contents of Table 3 from ITU-R P.2040-1 are repeated in this table. The values a, b, c, and d are used to calculate relative permittivity and conductivity. Except as noted for the three ground types, the frequency ranges given in the table are not hard limits but are indicative of the measurements used to derive the models. The `buildingMaterialPermittivity` function interpolates or extrapolates relative permittivity and conductivity values for frequencies that fall outside of the noted limits. To compute relative permittivity and conductivity for different types of ground as a function carrier frequencies up to 1000 GHz, see the `earthSurfacePermittivity` function.

Material Class	**Real Part of Relative Permittivity**		**Conductivity (S/m)**		**Frequency Range (GHz)**
	a	***b***	***c***	***d***	
Vacuum (~ air)	1	0	0	0	[0.001, 100]
Concrete	5.31	0	0.0326	0.8095	[1, 100]
Brick	3.75	0	0.038	0	[1, 10]
Plasterboard	2.94	0	0.0116	0.7076	[1, 100]
Wood	1.99	0	0.0047	1.0718	[0.001, 100]
Glass	6.27	0	0.0043	1.1925	[0.1, 100]
Ceiling board	1.50	0	0.0005	1.1634	[1, 100]
Chipboard	2.58	0	0.0217	0.78	[1, 100]
Floorboard	3.66	0	0.0044	1.3515	[50, 100]

Material Class	**Real Part of Relative Permittivity**		**Conductivity (S/m)**		**Frequency Range (GHz)**
	a	***b***	***c***	***d***	
Metal	1	0	10^7	0	[1, 100]
Very dry ground	3	0	0.00015	2.52	[1, 10] only[(a)]
Medium dry ground	15	– 0.1	0.035	1.63	[1, 10] only[(a)]
Wet ground	30	– 0.4	0.15	1.30	[1, 10] only[(a)]
Note (a): For the three ground types (very dry, medium dry, and wet), the noted frequency limits cannot be exceeded.					

References

[1] ITU-R P.2040-1. "Effects of Building Materials and Structures on Radiowave Propagation Above 100MHz." *International Telecommunications Union - Radiocommunications Sector (ITU-R)*. July 2015.

See Also

Functions

`earthSurfacePermittivity` | `propagationModel` | `raypl` | `raytrace`

Objects

`comm.Ray`

Introduced in R2020a

earthSurfacePermittivity

Permittivity and conductivity of earth surface materials

Syntax

```
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('pure-water',fc,
temp)
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('dry-ice',fc,temp)
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('sea-water',fc,
temp,salinity)
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('wet-ice',fc,
liqfrac)
```

```
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('soil',fc,temp,
sandpercent,claypercent,specificgravity,vwc)
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('soil',___,
bulkdensity)
```

```
[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('vegetation',fc,
temp,gwc)
```

Description

The `earthSurfacePermittivity` function computes electrical characteristics (relative permittivity, conductivity, and complex relative permittivity) of earth surface materials based on the methods and equations presented in ITU-R P.527 [1]. The `earthSurfacePermittivity` function provides various syntaxes to account for characteristics germane to the specified surface material.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('pure-water',fc,temp)` calculates the electrical characteristics for pure water at the specified frequency and temperature. For pure-water, the temperature setting must be greater than 0 ℃.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('dry-ice',fc,temp)` calculates the electrical characteristics for dry-ice at the specified frequency and temperature. For dry-ice, the temperature must be less than or equal to 0 ℃.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('sea-water',fc,temp,salinity)` calculates the electrical characteristics for sea water at the specified frequency, temperature, and salinity. For sea-water, the temperature must be greater than –2 ℃.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('wet-ice',fc,liqfrac)` calculates the electrical characteristics for wet ice at the specified frequency, and liquid water volume fraction. For wet-ice, the temperature is 0 ℃.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('soil',fc,temp,sandpercent,claypercent,specificgravity,vwc)` calculates the electrical characteristics for soil at the specified frequency, temperature, sand percentage, clay percentage, specific gravity, and volumetric water content.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('soil',___,bulkdensity)` sets the soil bulk density in addition to input arguments from the previous syntax.

`[epsilon,sigma,complexepsilon] = earthSurfacePermittivity('vegetation',fc,temp,gwc)` calculates the electrical characteristics for vegetation at the specified frequency, temperature, and gravimetric water content. For vegetation, the temperature must be greater than or equal to –20 ℃.

Examples

Compare Permittivity and Conductivity of Salt-free Sea Water to Pure Water

Compare the relative permittivity and conductivity for salt-free (zero-salinity) sea water to pure water.

Specify a carrier frequency of 9 GHz, temperature of 30℃, and salinity of zero.

```
fc = 9e9; % Carrier frequency in Hz.
temp = 30;
salinity = 0;
```

Compute the relative permittivity and conductivity.

```
[epsilon_pure_water,sigma_pure_water] = earthSurfacePermittivity('pure-water',fc,temp);
[epsilon_sea_water,sigma_sea_water] = earthSurfacePermittivity('sea-water',fc,temp,salinity);
```

Confirm that salt-free sea water and pure water have equal relative permittivity and conductivity.

```
isequal(epsilon_pure_water,epsilon_sea_water)
```

```
ans = logical
   1
```

```
isequal(sigma_pure_water,sigma_sea_water)
```

```
ans = logical
   1
```

Compare Permittivity and Conductivity of Wet Ice to Dry Ice

Compare the relative permittivity and conductivity for wet ice with no liquid water to dry ice at 0℃. Confirm the results differ by a negligible amount.

Specify a carrier frequency of 12 GHz.

```
fc = 12e9; % Carrier frequency in Hz.
```

Calculate the relative permittivity and conductivity for wet ice with zero liquid water by volume.

```
liqfrac = 0;
[epsilon_wet_ice_0,sigma_wet_ice_0] = earthSurfacePermittivity('wet-ice',fc,liqfrac); % Set liqu
```

Calculate the relative permittivity and conductivity for dry ice at 0 ℃.

```
temp = 0;
[epsilon_dry_ice_0,sigma_dry_ice_0] = earthSurfacePermittivity('dry-ice',fc,temp); % Set tempera
```

Compare the relative permittivity and conductivity for wet ice with no liquid to dry ice at 0℃. Confirm that wet ice with no liquid and dry ice at 0℃ have essentially equal relative permittivity and conductivity.

```
epsilon_wet_ice_0-epsilon_dry_ice_0
```

```
ans = 8.8818e-16
```

```
sigma_wet_ice_0-sigma_dry_ice_0
```

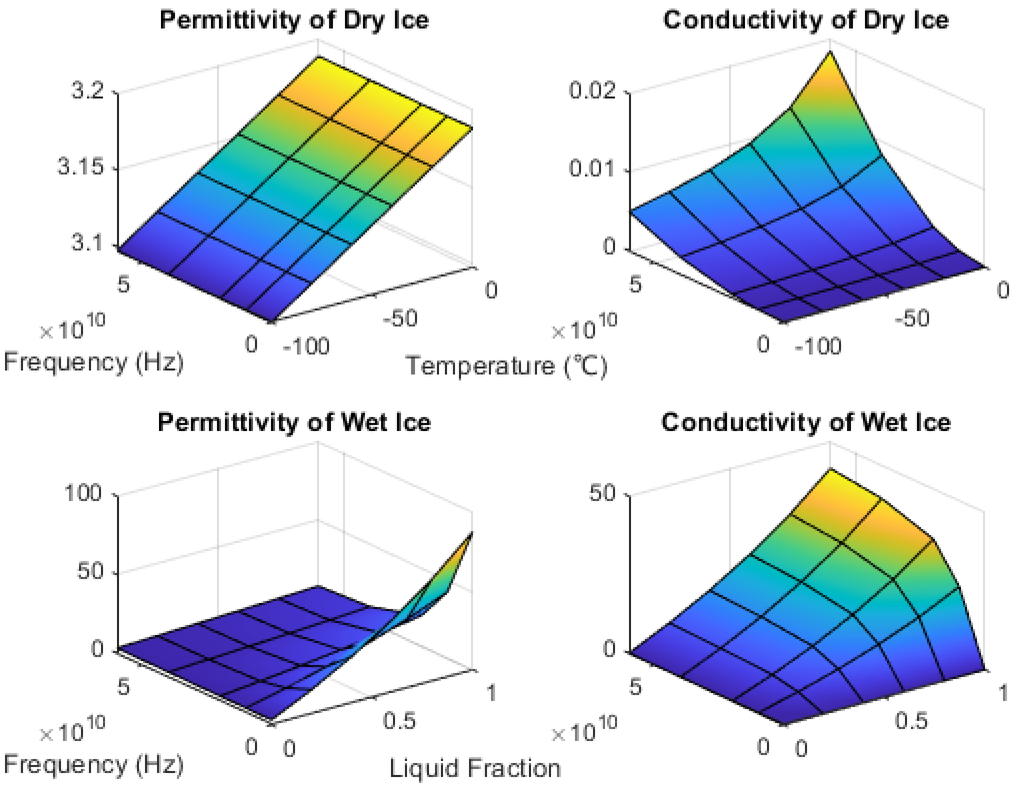
```
ans = -9.2179e-16
```

Plot permittivity and conductivity versus frequency for dry ice and for wet ice. For dry ice, vary the temperature. For wet ice, vary the liquid water volume fraction. Calculate the permittivity and conductivity values by using `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs.

```
freq = repmat([0.1,10,20,40,60]*1e9,6,1);
temp = repmat((-100:20:0)',1,5);
liqfrac = repmat((0:0.2:1)',1,5);
[epsilon_dry_ice, sigma_dry_ice] = arrayfun(@(x,y)earthSurfacePermittivity('dry-ice',x,y),freq,te
[epsilon_wet_ice, sigma_wet_ice] = arrayfun(@(x,y)earthSurfacePermittivity('wet-ice',x,y),freq,li
```

Display tiled surface plots across specified ranges.

```
figure
tiledlayout(2,2)
nexttile
surf(temp,freq,epsilon_dry_ice,'FaceColor','interp')
title('Permittivity of Dry Ice')
xlabel('Temperature (℃)')
ylabel('Frequency (Hz)')
nexttile
surf(temp,freq,sigma_dry_ice,'FaceColor','interp')
title('Conductivity of Dry Ice')
nexttile
surf(liqfrac,freq,epsilon_wet_ice,'FaceColor','interp')
title('Permittivity of Wet Ice')
xlabel('Liquid Fraction')
ylabel('Frequency (Hz)')
nexttile
surf(liqfrac,freq,sigma_wet_ice,'FaceColor','interp')
title('Conductivity of Wet Ice')
```



Calculate Permittivity and Conductivity of Various Soil Mixtures

Calculate relative permittivity and conductivity for various soil mixtures as defined by textual classifications in ITU-R P.527, Table 1.

Initialize computation variables for constant values and arrayed values.

```
fc = 28e9; % Frequency in Hz
temp = 23; % Temperature in °C
vwc = 0.5; % Volumetric water content
pSand = [51.52; 41.96; 30.63; 5.02]; % Sand percentage
pClay = [13.42; 8.53; 13.48; 47.38]; % Clay percentage
sg = [2.66; 2.70; 2.59; 2.56]; % Specific gravity
bd = [1.6006; 1.5781; 1.5750; 1.4758]; % Bulk density (g/cm^3)
```

Calculate the relative permittivity and conductivity for these textual classifications: sandy loam, loam, silty loam, and silty clay. Use `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs. Tabulate the results.

```
[Permittivity,Conductivity] = arrayfun(@(w,x,y,z)earthSurfacePermittivity( ...
    'soil',fc,temp,w,x,y,vwc,z),pSand,pClay,sg,bd);

pSilt = 100 - (pSand + pClay); % Silt percentage
soilType = ["Sandy Loam";"Loam";"Silty Loam";"Silty Clay"];
```

```
varNames1 = ["Soil Textual Classification";"Sand";"Clay";"Silt";"Specific Gravity";"Bulk Density"
varNames2 = ["Soil Textual Classification";"Permittivity";"Conductivity"];
```

ITU-R P.527, Table 1 specifies the sand percentage, clay percentage, specific gravity, and bulk density for soil mixtures with these soil textual classifications.

```
table(soilType,pSand,pClay,pSilt,sg,bd,'VariableNames',varNames1)
```

```
ans=4×6 table
    Soil Textual Classification    Sand     Clay     Silt     Specific Gravity    Bulk Density
    ___________________________    _____    _____    _____    ________________    ____________

           "Sandy Loam"            51.52    13.42    35.06          2.66             1.6006
           "Loam"                  41.96     8.53    49.51           2.7             1.5781
           "Silty Loam"            30.63    13.48    55.89          2.59              1.575
           "Silty Clay"             5.02    47.38     47.6          2.56             1.4758
```

The relative permittivity and conductivity for these soil textual classifications are included in this table.

```
table(soilType,Permittivity,Conductivity,'VariableNames',varNames2)
```

```
ans=4×3 table
    Soil Textual Classification    Permittivity    Conductivity
    ___________________________    ____________    ____________

           "Sandy Loam"               15.281            18.2
           "Loam"                     14.563          16.998
           "Silty Loam"               13.965          16.011
           "Silty Clay"               12.861          14.647
```

Calculate Permittivity and Conductivity of Vegetation

Calculate relative permittivity and conductivity versus frequency for vegetation, varying gravimetric water content and temperature.

Calculate relative permittivity and conductivity for vegetation at specified settings.

```
fc = 10e9; % Frequency in Hz
temp  = 23; % Temperature in °C
gwc = 0.68; % Gravimetric water content
[epsilon_veg,sigma_veg] = ...
    earthSurfacePermittivity('vegetation',fc,temp,gwc)
```

```
epsilon_veg = 20.5757

sigma_veg = 4.9320
```

Calculate values necessary to plot permittivity and conductivity by using `arrayfun` to apply the `earthSurfacePermittivity` function to the elements of the arrayed inputs.

For a range of temperatures, calculate values to plot permittivity and conductivity versus frequency for vegetation at a 0.68 gravimetric water content.

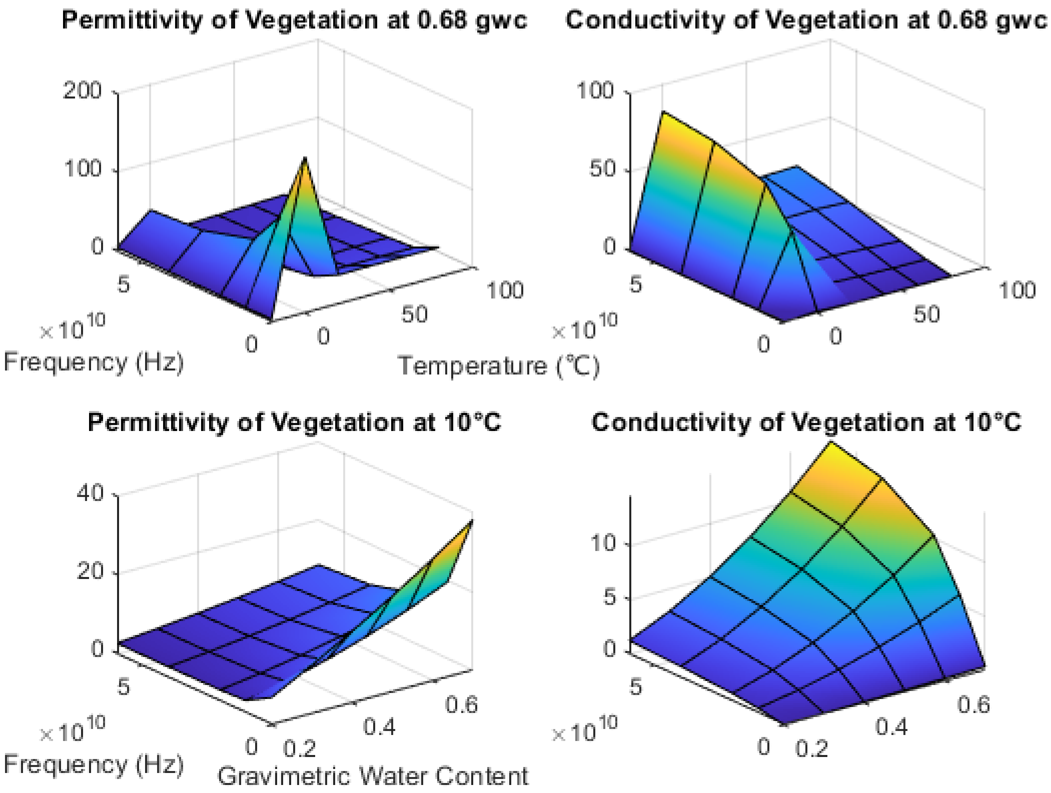
```
fc = repmat([0.1,10,20,40,60]*1e9,6,1);
gwc1 = 0.68;
temp1 = repmat((-20:20:80)',1,5);
[epsilon_veg_gwc,sigma_veg_gwc] = ...
    arrayfun(@(x,y)earthSurfacePermittivity('vegetation',x,y,gwc1),fc,temp1);
```

For a range of gravimetric water contents, calculate values to plot permittivity and conductivity versus frequency for vegetation at 10°C.

```
temp2 = 10;
gwc2 = repmat((0.2:0.1:0.7)',1,5);
[epsilon_veg_tmp, sigma_veg_tmp] = ...
    arrayfun(@(x,z)earthSurfacePermittivity('vegetation',x,temp2,z),fc,gwc2);
```

Display tiled surface plots across specified ranges.

```
figure
tiledlayout(2,2)
nexttile
surf(temp1,fc,epsilon_veg_gwc,'FaceColor','interp')
title('Permittivity of Vegetation at 0.68 gwc')
xlabel('Temperature (℃)')
ylabel('Frequency (Hz)')
nexttile
surf(temp1,fc,sigma_veg_gwc,'FaceColor','interp')
title('Conductivity of Vegetation at 0.68 gwc')
nexttile
surf(gwc2,fc,epsilon_veg_tmp,'FaceColor','interp')
title('Permittivity of Vegetation at 10°C')
xlabel('Gravimetric Water Content')
ylabel('Frequency (Hz)')
nexttile
surf(gwc2,fc,sigma_veg_tmp,'FaceColor','interp')
title('Conductivity of Vegetation at 10°C')
```



Input Arguments

`fc` — Carrier frequency

scalar in the range (0, 1e12]

Carrier frequency in Hz, specified as a scalar in the range (0, 1e12].

Data Types: `double`

`temp` — Temperature

numeric scalar

Temperature in °C, specified as a numeric scalar. Valid surfaces and associated temperature limits are indicated in this table.

Surface	Valid Temperature (°C)
pure-water	greater than 0
dry-ice	less than or equal to 0
sea-water	greater than or equal to –2
soil	any numeric
vegetation	≥ –20

Note When the surface is wet-ice, the temperature is 0 ℃.

Data Types: `double`

`salinity` — Salinity of sea water
nonnegative scalar

Salinity of the sea water in g/Kg, specified as a nonnegative scalar.

Data Types: `double`

`liqfrac` — Liquid water volume fraction of wet ice
numeric scalar in the range [0, 1]

Liquid water volume fraction of the wet ice, specified as a numeric scalar in the range [0, 1].

Data Types: `double`

`sandpercent` — Sand percentage of soil
numeric scalar in the range [0, 100]

Sand percentage of the soil, specified as a numeric scalar in the range [0, 100]. The sum of `sandpercent` and `claypercent` must be less than or equal to 100.

Data Types: `double`

`claypercent` — Clay percentage of soil
numeric scalar in the range [0, 100]

Clay percentage of the soil, specified as a numeric scalar in the range [0, 100]. The sum of `sandpercent` and `claypercent` must be less than or equal to 100.

Data Types: `double`

`specificgravity` — Specific gravity of soil
nonnegative scalar

Specific gravity of the soil, specified as a nonnegative scalar. The specific gravity is the mass density of the soil sample divided by the mass density of the amount of water in the soil sample.

Data Types: `double`

`vwc` — Volumetric water content of soil
numeric scalar in the range [0, 1]

Volumetric water content of the soil, specified as a numeric scalar in the range [0, 1]. For more information, see “Soil Water Content” on page 2-978.

Data Types: `double`

`bulkdensity` — Bulk density of soil
nonnegative scalar

Bulk density, in g/cm^3, of the soil, specified as a nonnegative scalar. For more information, see “Soil Water Content” on page 2-978.

Data Types: `double`

`gwc` — Gravimetric water content of vegetation
numeric scalar in the range [0, 0.7]

Gravimetric water content of the vegetation, specified as a numeric scalar in the range [0, 0.7]. For more information, see "Soil Water Content" on page 2-978.

Data Types: `double`

Output Arguments

`epsilon` — Relative permittivity
nonnegative scalar

Relative permittivity of the earth surface, returned as a nonnegative scalar.

`sigma` — Conductivity
nonnegative scalar

Conductivity of the earth surface in Siemens per meter (S/m), returned as a nonnegative scalar.

`complexepsilon` — Complex relative permittivity
complex scalar

Complex relative permittivity of the earth surface, returned as a complex scalar calculated as
`complexepsilon` = `epsilon` – 1*i* `sigma` / ($2\pi fc\varepsilon_0$).
The computation of `complexepsilon` is based on Equations (59) and (9b) in ITU-R P.527 [1]. *f* is the frequency in GHz. *c* is the velocity of light in free space. ε_0 = 8.854187817e-12 Farads/m, where ε_0 is the electric constant for the permittivity of free space.

More About

ITU Terrain Materials

ITU-R P.527 [1] presents methods and equations to calculate complex relative permittivity at carrier frequencies up to 1,000 GHz for these common earth surface materials.

- Water
- Sea Water
- Dry or Wet Ice
- Dry or Wet Soil (combination of sand, clay, and silt)
- Vegetation (above and below freezing)

As described in ITU-R P.527, specific textural classification applies to these mixtures of sand, clay, and silt in soil with associated specific gravities and bulk densities.

Soil Designation Textural Class	Sandy Loam	Loam	Silty Loam	Silty Clay
% Sand	51.52	41.96	30.63	5.02
% Clay	13.42	8.53	13.48	47.38
% Silt	35.06	49.51	55.89	47.60

Soil Designation Textural Class	**Sandy Loam**	**Loam**	**Silty Loam**	**Silty Clay**
Specific gravity (ρ_s)	2.66	2.70	2.59	2.56
Bulk Density (ρ_b) in g/cm^3	1.6006	1.5781	1.5750	1.4758

Soil Water Content

Soil water content is expressed on a gravimetric or volumetric basis. Gravimetric water content, `gwc`, is the mass of water per mass of dry soil. Volumetric water content, `vwc`, is the volume of liquid water per volume of soil. The bulk density, `bulkdensity`, is the ratio of the dry soil weight to the volume of the soil sample. The relationship between `gwc` and `vwc` is `vwc` = `gwc` □ `bulkdensity`. When bulk density is not specified, the value of `bulkdensity` is computed by using ITU-R P.527, Equation 36:

`bulkdensity` = 1.07256 + 0.078886 ln(*pSand*) + 0.038753 ln(*pClay*) + 0.032732 ln(*pSilt*),

where

- *pSand* = `sandpercent`
- *pClay* = `claypercent`
- *pSilt* = 100 – (`sandpercent` + `claypercent`)

References

[1] ITU-R P.527-5. "Electrical characteristics of the surface of the Earth." *International Telecommunications Union - Radiocommunications Sector (ITU-R)*. August 2019.

See Also

Functions
`buildingMaterialPermittivity` | `propagationModel` | `raypl` | `raytrace`

Objects
`comm.Ray`

Introduced in R2020a

raypl

Calculate path loss and phase shift for ray

Syntax

```
[pl,phase] = raypl (ray)
[pl,phase] = raypl (ray,Name,Value)
```

Description

`[pl,phase] = raypl (ray)` returns the path loss in dB and phase shift in radians based on the properties specified by `ray`. The path loss and path shift computations consider the free space loss and reflection loss derived from the propagation path, reflection materials, and polarizations. The function accounts for geometric coupling between horizontal and vertical polarizations only when both transmit and receive antennas are polarized. For more information, see “Path Loss Computation” on page 2-984.

`[pl,phase] = raypl (ray,Name,Value)` calculates the path loss and phase shift with additional options specified by one or more name-value pair arguments.

Examples

Reevaluate Path Loss Changing Reflection Materials and Frequency

Change the reflection materials and frequency for a ray and reevaluate the path loss and phase shift.

Launch Site Viewer with buildings in Hong Kong. For more information about the osm file, see [1] on page 2-0 . Specify transmitter and receiver sites.

```
viewer = siteviewer("Buildings","hongkong.osm");

tx = txsite("Latitude",22.2789,"Longitude",114.1625, ...
    "AntennaHeight",10,"TransmitterPower",5, ...
    "TransmitterFrequency",28e9);
rx = rxsite("Latitude",22.2799,"Longitude",114.1617, ...
    "AntennaHeight",1);
```

Perform ray tracing between the sites.

```
rays = raytrace(tx,rx,"NumReflections",0:2);
```

Find the first ray with 2-order reflections from the result. Display the ray characteristics. Plot the ray to see the ray reflect off two buildings.

```
ray = rays{1}(find([rays{1}.NumReflections] == 2,1))
```

```
ray = 
  Ray with properties:

      PathSpecification: 'Locations'
```

```
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×2 double]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 122.1825
             PhaseShift: 4.5977

   Read-only properties:
       PropagationDelay: 8.3060e-07
    PropagationDistance: 249.0069
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
          NumReflections: 2
```

```
plot(ray);
```

By default, all buildings have concrete building material electrical characteristics. Change the material to metal for the second reflection and re-evaluate path loss. Use the `raypl` function to reevaluate the pathloss for the ray. Display the ray path to compare the change in path loss. Replot to show the slight change in color due to the path loss change of the ray.

```
[ray.PathLoss,ray.PhaseShift] = raypl(ray, ...
    "ReflectionMaterials",["concrete","metal"])
```

```
ray = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×2 double]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 117.4814
             PhaseShift: 4.5977

   Read-only properties:
       PropagationDelay: 8.3060e-07
    PropagationDistance: 249.0069
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 2


ray = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×2 double]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 117.4814
             PhaseShift: 4.5977

   Read-only properties:
       PropagationDelay: 8.3060e-07
    PropagationDistance: 249.0069
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 2
```

```
plot(ray);
```

Change the frequency and reevaluate the path loss and phase shift. Plot the ray again and observe the obvious color change.

```
ray.Frequency = 2e9;
[ray.PathLoss,ray.PhaseShift] = raypl(ray, ...
    "ReflectionMaterials",["concrete","metal"]);
plot(ray);
```

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

ray — Ray configuration

`comm.Ray` object

Ray configuration, specified as one `comm.Ray` object. The object must have the `PathSpecification` property set to `"Locations"`.

Data Types: `comm.Ray`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `raypl(ray,'TransmitterPolarization','H','ReceiverPolarization','H')`, specifies the horizontal polarizations for the transmit and receive antennas for `ray`.

ReflectionMaterials — Reflection materials

`"concrete"` (default) | string scalar | 1-by-*NR* string vector | 2-by-1 numeric vector | 2-by-*NR* numeric matrix

Reflection materials for a non-line-of-sight (NLOS) ray, specified as a string scalar, 1-by-*NR* string vector, 2-by-1 numeric vector, or 2-by-*NR* numeric matrix. *NR* represents the number of reflections as specified by the `comm.Ray.NumReflections` property.

- When `ReflectionMaterials` is specified as a string scalar or string vector, the reflection material must be one of `"concrete"`, `"brick"`, `"wood"`, `"glass"`, `"metal"`, `"water"`, `"vegetation"`, `"loam"`, or `"perfect-reflector"`. When specified as a string scalar, the setting applies to all the reflections.
- When `ReflectionMaterials` is specified as a 2-by-1 numeric vector, the [relative permittivity; conductivity] value pair applies to all the reflections.
- When `ReflectionMaterials` is specified as a 2-by-*NR* numeric matrix, the [relative permittivity; conductivity] value pair in each column applies for each of the *NR* reflection points, respectively.

Example: `"ReflectionMaterials",["concrete","water"]`, specifies that a ray with two reflections will use electrical characteristics of concrete at the first reflection point and water at the second reflection point.

Data Types: `string` | `char` | `double`

`TransmitterPolarization` — Transmit antenna polarization type
`"none"` (default) | `"H"` | `"V"` | `"RHCP"` | `"LHCP"` | normalized 2-by-1 Jones vector

Transmit antenna polarization type, specified as `"none"`, `"H"`, `"V"`, `"RHCP"`, `"LHCP"`, or a normalized [H; V] Jones vector. For more information, see “Jones Vector Notation” on page 2-985.

Example: `'TransmitterPolarization','RHCP'`, specifies right-hand circular polarization for the transmit antenna.

Data Types: `double` | `char` | `string`

`ReceiverPolarization` — Receive antenna polarization type
`"none"` (default) | `"H"` | `"V"` | `"RHCP"` | `"LHCP"` | normalized 2-by-1 Jones vector

Receive antenna polarization type, specified as `"none"`, `"H"`, `"V"`, `"RHCP"`, `"LHCP"`, or a normalized [H; V] Jones vector. For more information, see “Jones Vector Notation” on page 2-985.

Example: `'ReceiverPolarization',[1;0]`, specifies horizontal polarization for the receive antenna by using Jones vector notation.

Data Types: `double` | `char` | `string`

`TransmitterAxes` — Orientation of transmit antenna axes
3-by-3 identity matrix (default) | 3-by-3 unitary matrix

Orientation of the transmit antenna axes, specified as a 3-by-3 unitary matrix indicating the rotation from the transmitter local coordinate system (LCS) into the global coordinate system (GCS). When the `CoordinateSystem` property of the `comm.Ray` is set to `"Geographic"`, the GCS orientation is the local East-North-Up (ENU) coordinate system at transmitter. For more information, see “Coordinate System Orientation” on page 2-983.

Example: `'TransmitterAxes',eye(3)`, specifies that the local coordinate system for the transmitter axes is aligned with the global coordinate system. This is the default orientation.

Data Types: `double`

`ReceiverAxes` — Orientation of receive antenna axes
3-by-3 identity matrix (default) | 3-by-3 unitary matrix

Orientation of the receive antenna axes, specified as a 3-by-3 unitary matrix indicating the rotation from the receiver local coordinate system (LCS) into the global coordinate system (GCS). The GCS orientation is the local East-North-Up (ENU) coordinate system at receiver when the .`CoordinateSystem` property of the `comm.Ray` is set to `"Geographic"`. For more information, see “Coordinate System Orientation” on page 2-983.

Example: `'ReceiverAxes',[0 -1 0; 1 0 0; 0 0 1]`, specifies a 90° rotation around the z-axis of the local receiver coordinate system with respect to the global coordinate system.

Data Types: `double`

Output Arguments

`pl` — Path loss

scalar

Path loss in dB, returns the path loss calculated for the input ray object, accounting for any modifications specified by `Name,Value` pairs.

`phase` — Phase shift

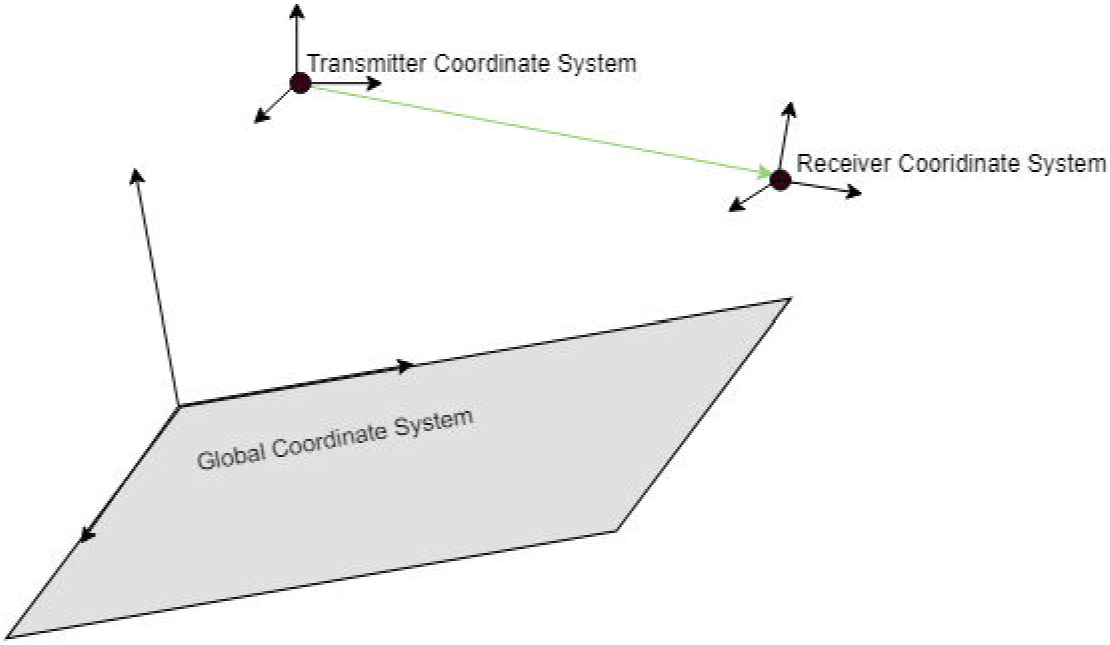
scalar

Phase shift in radians, returns the phase shift calculated for the input ray object, accounting for any modifications specified by `Name,Value` pairs.

More About

Coordinate System Orientation

This image shows the orientation of the electromagnetic fields in the global coordinate system (GCS) and the local coordinate systems of the transmitter and receiver.



When the `CoordinateSystem` property of the `comm.Ray` is set to `"Geographic"`, the GCS orientation is the local East-North-Up (ENU) coordinate system at observer. The path loss computation accounts for the round-earth differences between ENU coordinates at the transmitter and receiver.

Path Loss Computation

The path loss computations in raypl follow the path loss and reflection matrix computations as described in IEEE Document 802.11-09/0334r8 [1]. The function accounts for geometric coupling between horizontal and vertical polarizations only when both transmit and receive antennas are polarized.

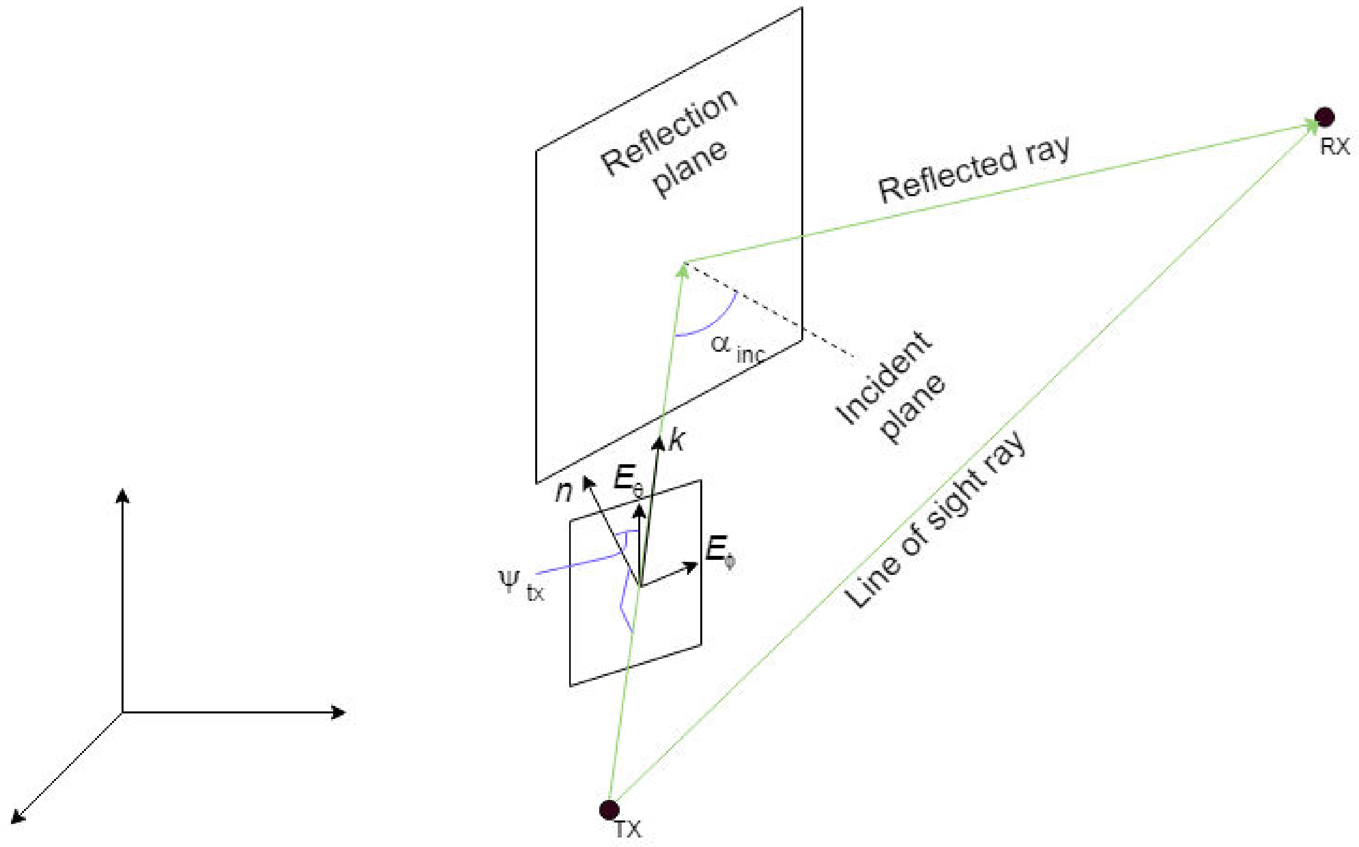
For a first order signal reflection, the reflection matrix, H_{ref1}, is computed as

$$H_{ref1} = \begin{bmatrix} \cos(\psi_{rx}) & \sin(\psi_{rx}) \\ -\sin(\psi_{rx}) & \cos(\psi_{rx}) \end{bmatrix} \times \begin{bmatrix} R_{\perp}(\alpha_{inc}) & 0 \\ 0 & R_{\parallel}(\alpha_{inc}) \end{bmatrix} \times \begin{bmatrix} \cos(\psi_{tx}) & \sin(\psi_{tx}) \\ -\sin(\psi_{tx}) & \cos(\psi_{tx}) \end{bmatrix}$$

The terms in the channel propagation matrix computation represent

- RX geometric coupling matrix — Recalculation of the polarization vector from the plane of incidence basis to RX coordinates.
- Polarization matrix — Matrix includes the reflection coefficients *R*▯ and *R*‖ for the perpendicular and parallel components of the electric field *E* ▯ and *E* ‖ respectively.
- TX geometric coupling matrix — Recalculation of the polarization vector from the TX coordinates basis to the plane of incidence.

This figure illustrates a first order reflected signal path.



Where

- The reflection plane is offset from the global coordinate system origin.
- k represents the waveform propagation vector.
- n represents the vector normal to the incident plane.
- E_θ and E_φ represent the vertical and horizontal electromagnetic field vectors.
- α_{inc} represents the incident angle of k.
- ψ_{tx} represents the angle between E_θ and a normal to the incident plane.
- TX represents the transmit antenna.
- RX represents the receive antenna.

The reflection matrix computations for second order signal reflections extend from the first order signal reflection computations. For more information, see IEEE Document 802.11-09/0334r8 [1].

Jones Vector Notation

For Jones vector notation, the raypl function describes signal polarization using Jones calculus.

The orthogonal components of Jones vectors are defined for E_θ and E_φ. This table shows the Jones vector corresponding to various antenna polarizations.

Antenna Polarization Type	Corresponding Jones Vector
Linear polarized in the θ direction	$\begin{pmatrix} H \\ V \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$
Linear polarized in the φ direction	$\begin{pmatrix} H \\ V \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
Left-hand circular polarized (LHCP)	$\begin{pmatrix} H \\ V \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} j \\ 1 \end{pmatrix}$
Right-hand circular polarized (RHCP)	$\begin{pmatrix} H \\ V \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} -j \\ 1 \end{pmatrix}$

References

[1] Maltsev, A., et al. "Channel models for 60 GHz WLAN systems." IEEE Document 802.11-09/0334r8, May 2010.

See Also

Functions
`buildingMaterialPermittivity` | `earthSurfacePermittivity` | `propagationModel` | `raytrace`

Objects
`comm.Ray` | `siteviewer`

Introduced in R2020a

blkdiagbfweights

MIMO channel block diagonalized weights

Syntax

```
[wp,wc] = blkdiagbfweights(chanmat,ns)
[wp,wc] = blkdiagbfweights(chanmat,ns,pt)
```

Description

`[wp,wc] = blkdiagbfweights(chanmat,ns)` returns precoding weights, `wp`, and combining weights, `wc`, derived from the channel response matrices contained in a MATLAB cell array `chanmat`.

- You can specify multiple user channels by putting each channel in a `chanmat` cell. `chanmat{k}` represents the k^{th}channel from the transmitter to the user.
- - For a single frequency, specify the channel cell as a matrix.
  - For multiply frequencies, specify the channel cell as a three-dimensional array where the rows represent different subcarriers.
- Specify multiple subchannels per channel using the `ns` argument. Subchannels represent different data streams. `ns` specifies the number of subchannels for each user channel. Multiply the data streams by the precoding weights, `wp`.

The precoding and combining weights diagonalize the channel into independent subchannels so that for the k^{th}user, the matrix `wp*chanmat{k}*wc{k}` is diagonal for each subcarrier.

`[wp,wc] = blkdiagbfweights(chanmat,ns,pt)` also specifies the total transmitted power, `pt`, per subcarrier.

Examples

Spatial Multiplexing with Block Diagonal Weights

Start with a base station consisting of a uniform linear array (ULA) with 16 antennas, and two users having receiver ULA arrays with 8 and 4 antennas, respectively. Show that using block diagonalization-based precoding and combining weights achieves spatial multiplexing, where the received signal at each user can be decoded without interference from the other user. Specify two data streams for each user.

Specify the transmitter location in `txpos` and two user receiver locations in `rxpos1` and `rxpos2`. Array elements are spaced one-half wavelength apart.

```
txpos = (0:15)*0.5;
rxpos1 = (0:7)*0.5;
rxpos2 = (0:3)*0.5;
```

Create the channel matrix cell array using `scatteringchanmtx` and then compute the beamforming weights `wp` and `wc`. Each channel corresponds to a user. Assume that the channels have 10 scatterers. Each channel has two subchannels specified by the vector `ns`.

```
chanmat = {scatteringchanmtx(txpos,rxpos1,10), ...
    scatteringchanmtx(txpos,rxpos2,10)};
ns = [2 2];
[wp,wc] = blkdiagbfweights(chanmat,ns);
```

The weights diagonalize the channel matrices for each user.

For channel 1:

```
disp(wp*chanmat{1}*wc{1})
```

```
   8.2269 - 0.0000i   0.0000 - 0.0000i
   0.0000 + 0.0000i   6.1371 - 0.0000i
   0.0000 - 0.0000i  -0.0000 + 0.0000i
   0.0000 - 0.0000i   0.0000 + 0.0000i
```

For channel 2:

```
disp(wp*chanmat{2}*wc{2})
```

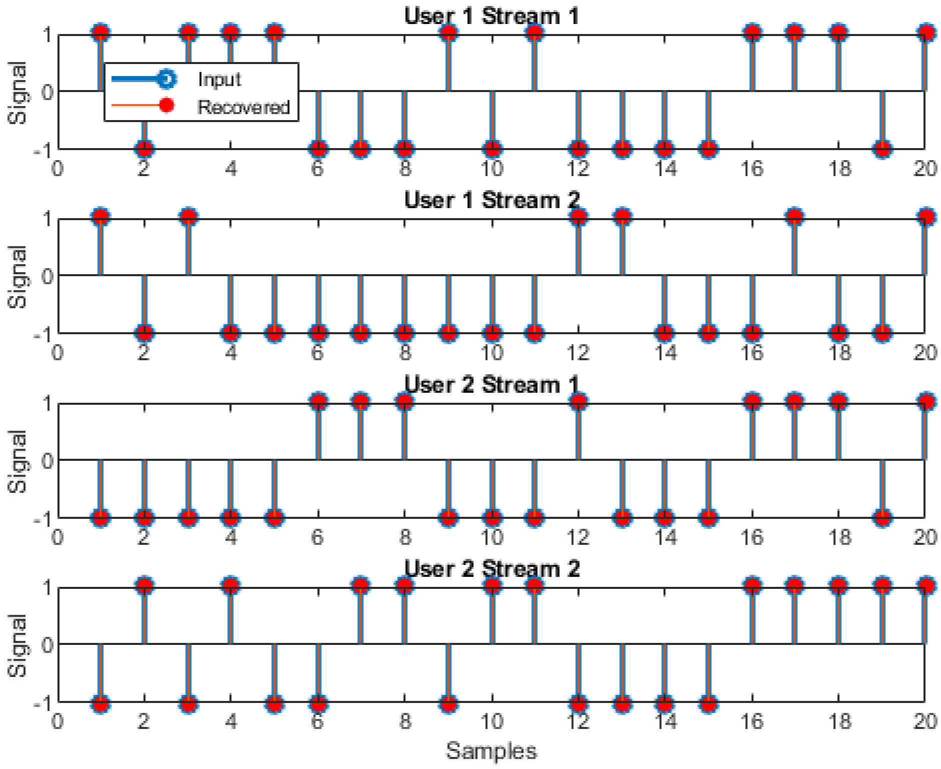
```
  -0.0000 + 0.0000i  -0.0000 + 0.0000i
  -0.0000 + 0.0000i  -0.0000 + 0.0000i
   8.7543 - 0.0000i   0.0000 - 0.0000i
   0.0000 + 0.0000i   4.4372 + 0.0000i
```

First create four subchannels to carry the data streams: two subchannels per channel. Each data stream contains 20 samples of ±1. Precode the input streams and combine the streams to produce the recovered signals.

```
x = 2*round(rand([20,4])) - 1;
xp = x*wp;
y1 = xp*chanmat{1} + 0.1*randn(20,8);
y2 = xp*chanmat{2} + 0.1*randn(20,4);
y = [y1*wc{1},y2*wc{2}];
```

Overlay stem plots of the input and recovered signals to show that the received user signals are the same as the transmitted signals.

```
for m = 1:4
    subplot(4,1,m)
    s = stem([x(:,m) 2*((real(y(:,m)) > 0) - 0.5)]);
    s(1).LineWidth = 2;
    s(2).MarkerEdgeColor = 'none';
    s(2).MarkerFaceColor = 'r';
    ylabel('Signal')
    title(sprintf('User %d Stream %d',ceil(m/2),rem(m-1,2) + 1))
    if m==1
        legend('Input','Recovered','Location','best')
    end
end
xlabel('Samples')
```



Spatial Multiplexing with Specified Power

Start with a base station consisting of a uniform linear array (ULA) with 16 antennas, and two users having receiver ULA arrays with 8 and 5 antennas, respectively. Show how to use three-dimensional arrays of channel matrices to handle two subcarriers. Then, the channel matrix for the first user takes the form 2-by-16-by-8 and the channel matrix for the second users takes the form 2-by-16-by-5. Also assume that there are two data streams for each user.

Specify the transmitter location in `txpos` and two user receiver locations in `rxpos1` and `rxpos2`. Array elements are spaced one-half wavelength apart.

```
nr1 = 8;
nr2 = 5;
txpos = (0:15)*0.5;
rxpos1 = (0:(nr1-1))*0.5;
rxpos2 = (0:(nr2-1))*0.5;
```

Create the channel matrices using `scatteringchanmtx` and put them in a cell array. To create a second subchannel for each receiver, duplicate each channel matrix. Assume 10 point scatterers in computing the channel matrix.

```
smtmp1 = scatteringchanmtx(txpos,rxpos1,10);
smtmp2 = scatteringchanmtx(txpos,rxpos2,10);
sm1 = zeros(2,16,8);
```

```
sm2 = zeros(2,16,5);
sm1(1,:,:) = smtmp1;
sm1(2,:,:) = smtmp1;
sm2(1,:,:) = smtmp2;
sm2(2,:,:) = smtmp2;
chanmat = {sm1,sm2};
```

Specify that there are two data streams for each user.

```
ns = [2 2];
```

Specify the transmitted powers for each subcarrier.

```
pt = [1.0 1.5];
```

Compute the beamforming weights.

```
[wp,wc] = blkdiagbfweights(chanmat,ns,pt);
```

Show that the channels are diagonalized for the first subcarrier.

```
ksubcr = 1;
wpx = squeeze(wp(ksubcr,:,:));
chanmat1 = squeeze(chanmat{1}(ksubcr,:,:));
chanmat2 = squeeze(chanmat{2}(ksubcr,:,:));
wc1 = squeeze(wc{1}(ksubcr,:,:));
wc2 = squeeze(wc{2}(ksubcr,:,:));
wpx*chanmat1*wc1
```

```
ans = 4×2 complex

   8.2104 + 0.0000i  -0.0000 - 0.0000i
   0.0000 - 0.0000i   5.9732 - 0.0000i
   0.0000 - 0.0000i  -0.0000 - 0.0000i
   0.0000 - 0.0000i   0.0000 - 0.0000i
```

```
wpx*chanmat2*wc2
```

```
ans = 4×2 complex

  -0.0000 + 0.0000i   0.0000 + 0.0000i
  -0.0000 + 0.0000i   0.0000 + 0.0000i
   8.8122 + 0.0000i  -0.0000 + 0.0000i
   0.0000 + 0.0000i   4.8186 - 0.0000i
```

Propagate the signals to each user and then decode. Generate four streams of random data containing -1's and +1's and having two columns for each user. Each stream is a subchannel.

```
x = 2*(round(rand([20 4]))) - 1;
```

Precode the data streams.

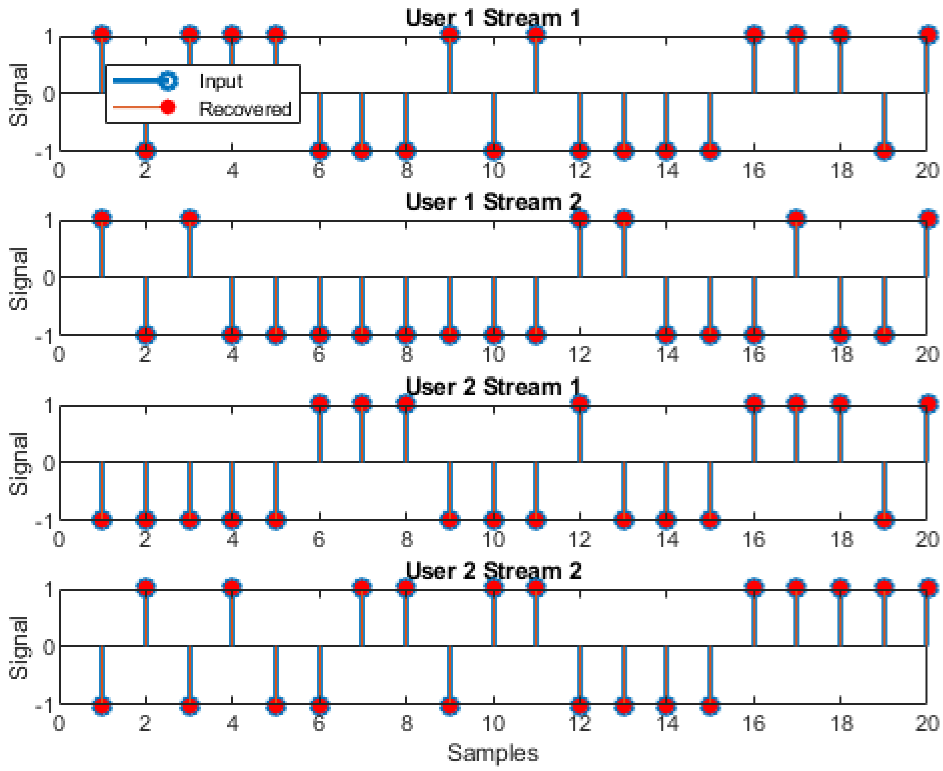
```
xp = x*wpx;
y1 = xp*chanmat1 + 0.1*randn(20,8);
y2 = xp*chanmat2 + 0.1*randn(20,5);
```

Decode the data streams.

```
y = [y1*wc1,y2*wc2];
```

Overlay stem plots of the input and recovered signals to show that the received user signals are the same as the transmitted signals.

```
for m = 1:4
    subplot(4,1,m)
    s = stem([x(:,m) 2*((real(y(:,m)) > 0) - 0.5)]);
    s(1).LineWidth = 2;
    s(2).MarkerEdgeColor = 'none';
    s(2).MarkerFaceColor = 'r';
    ylabel('Signal')
    title(sprintf('User %d Stream %d',ceil(m/2),rem(m-1,2) + 1))
    if m==1
        legend('Input','Recovered','Location','best')
    end
end
xlabel('Samples')
```



Input Arguments

chanmat — Channel response matrices

N_u-element cell array

Channel response matrices, specified as an N_u-element cell array. N_u is the number of receive arrays. Each cell corresponds to a different channel and contains a channel response matrix or a three dimensional MATLAB array. The cell array must contain either all matrices or all arrays. For matrices, the number of rows for all matrices must be the same. For three-dimensional arrays, the number of rows and columns must be the same.

- If the k^{th} cell is a matrix, the matrix has the size N_t-by-$N_r(k)$. N_t is the number of elements in the transmitting array and $N_r(k)$ is the number of elements in the k^{th} receiving array.
- If the k^{th} cell is an array, the array has the size L-by-N_t-by-$N_r(k)$. L is the number of subcarriers. N_t is the number of elements in the transmit array and $N_r(k)$ is the number of elements in the k^{th} receive array.

Data Types: `double`
Complex Number Support: Yes

`ns` — Number of data streams per receive array
N_u-element row vector of positive integers

Number of data streams per receive array, specified as an N_u-element row vector of positive integers. N_u is the number of receive arrays.

Data Types: `double`

`pt` — Total transmitted power per subcarrier
`1` (default) | positive scalar | L-element vector of positive values

Total transmitted power per subcarrier, specified as a positive scalar or an L-element vector of positive values. L is the number of subcarriers. If `pt` is a scalar, all subcarriers have the same transmitted power. If `pt` is a vector, each vector element specifies the transmitted power for the corresponding subcarrier. Power is in linear units.

Data Types: `double`

Output Arguments

`wp` — Precoding weights
complex-valued N_{st}-by-N_t matrix | complex-valued L-by-N_{st}-by-N_t MATLAB array

Precoding weights, returned as a complex-valued N_{st}-by-N_t matrix or a complex-valued L-by-N_{st}-by-N_t MATLAB array.

- If `chanmat` contains matrices, `wp` is a complex-valued N_{st}-by-N_t matrix where N_{st} is the total number of data channels (`sum(ns)`).
- If `chanmat` contains three-dimensional MATLAB arrays, `wp` is a complex-valued L-by-N_{st}-by-N_t MATLAB array where N_{st} is the total number of data channels (`sum(ns)`).

Units are dimensionless.

Data Types: `double`

`wc` — Combining weights
N_u-element cell array

Combining weights, returned as an N_u-element cell array. Units are dimensionless.

- If `chanmat` contains matrices, the k^{th} cell in `wc` contains a complex valued $N_r(k)$-by-$N_s(k)$ matrix. $N_s(k)$ is the value of the argument `ns` for the k^{th} receive array.

- If chanmat contains three-dimensional MATLAB arrays, the k^{th} cell of wc contains a complex-valued L-by-$N_r(k)$-by-$N_s(k)$ MATLAB array. $N_s(k)$ is the value of the k^{th} entry of the ns vector.

Data Types: double

References

[1] Heath, Robert W., et al. “An Overview of Signal Processing Techniques for Millimeter Wave MIMO Systems.” *IEEE Journal of Selected Topics in Signal Processing*, vol. 10, no. 3, Apr. 2016, pp. 436–53. DOI.org (Crossref), doi:10.1109/JSTSP.2016.2523924. Bibliography

[2] Tse, D. and P. Viswanath, *Fundamentals of Wireless Communications*, Cambridge: Cambridge University Press, 2005.

[3] Paulraj, A. *Introduction to Space-Time Wireless Communications*, Cambridge: Cambridge University Press, 2003.

[4] Spencer, Q.H., et al. "Zero-Forcing Methods for Downlink Spatial Multiplexing in Multiuser MIMO Channels." *IEEE Transactions on Signal Processing*, Vol. 52, No. 2, February 2004, pp. 461-471. DOI.org (Crossref), doi:10.1109/TSP.2003.821107.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Does not support variable-size inputs.

See Also

Objects

comm.MIMOChannel

Introduced in R2020a

cart2sphvec

Convert vector from Cartesian components to spherical representation

Syntax

```
vs = cart2sphvec(vr,az,el)
```

Description

`vs = cart2sphvec(vr,az,el)` converts the components of a vector or set of vectors, `vr`, from their representation in a local Cartesian coordinate system to a spherical basis representation contained in `vs`. A spherical basis representation is the set of components of a vector projected into a basis given by $(\hat{\mathbf{e}}_{az}, \hat{\mathbf{e}}_{el}, \hat{\mathbf{e}}_{R})$. The orientation of a spherical basis depends upon its location on the sphere as determined by azimuth, `az`, and elevation, `el`.

Examples

Spherical Representation of Unit Z-Vector

Start with a vector in Cartesian coordinates pointing along the *z*-direction and located at 45° azimuth, 45° elevation. Compute its components with respect to the spherical basis at that point.

```
vr = [0;0;1];
vs = cart2sphvec(vr,45,45)
```

```
vs = 3×1

         0
    0.7071
    0.7071
```

Input Arguments

`vr` — Vector in Cartesian basis representation
3-by-1 column vector | 3-by-N matrix

Vector in Cartesian basis representation specified as a 3-by-1 column vector or 3-by-N matrix. Each column of `vr` contains the three components of a vector in the right-handed Cartesian basis *x,y,x*.

Example: `[4.0; -3.5; 6.3]`

Data Types: `double`
Complex Number Support: Yes

az — Azimuth angle
scalar in range [–180,180]

Azimuth angle specified as a scalar in the closed range [–180,180]. Angle units are in degrees. To define the azimuth angle of a point on a sphere, construct a vector from the origin to the point. The azimuth angle is the angle in the *xy*-plane from the positive *x*-axis to the vector's orthogonal projection into the *xy*-plane. As examples, zero azimuth angle and zero elevation angle specify a point on the *x*-axis while an azimuth angle of 90° and an elevation angle of zero specify a point on the *y*-axis.

Example: `45`

Data Types: `double`

`el` — Elevation angle
scalar in range [–90,90]

Elevation angle specified as a scalar in the closed range [–90,90]. Angle units are in degrees. To define the elevation of a point on the sphere, construct a vector from the origin to the point. The elevation angle is the angle from its orthogonal projection into the *xy*-plane to the vector itself. As examples, zero elevation angle defines the equator of the sphere and ±90° elevation define the north and south poles, respectively.

Example: `30`

Data Types: `double`

Output Arguments

`vs` — Vector in spherical basis
3-by-1 column vector | 3-by-N matrix

Spherical representation of a vector returned as a 3-by-1 column vector or 3-by-N matrix having the same dimensions as `vs`. Each column of `vs` contains the three components of the vector in the right-handed ($\hat{\mathbf{e}}_{az}$, $\hat{\mathbf{e}}_{el}$, $\hat{\mathbf{e}}_{R}$) basis.

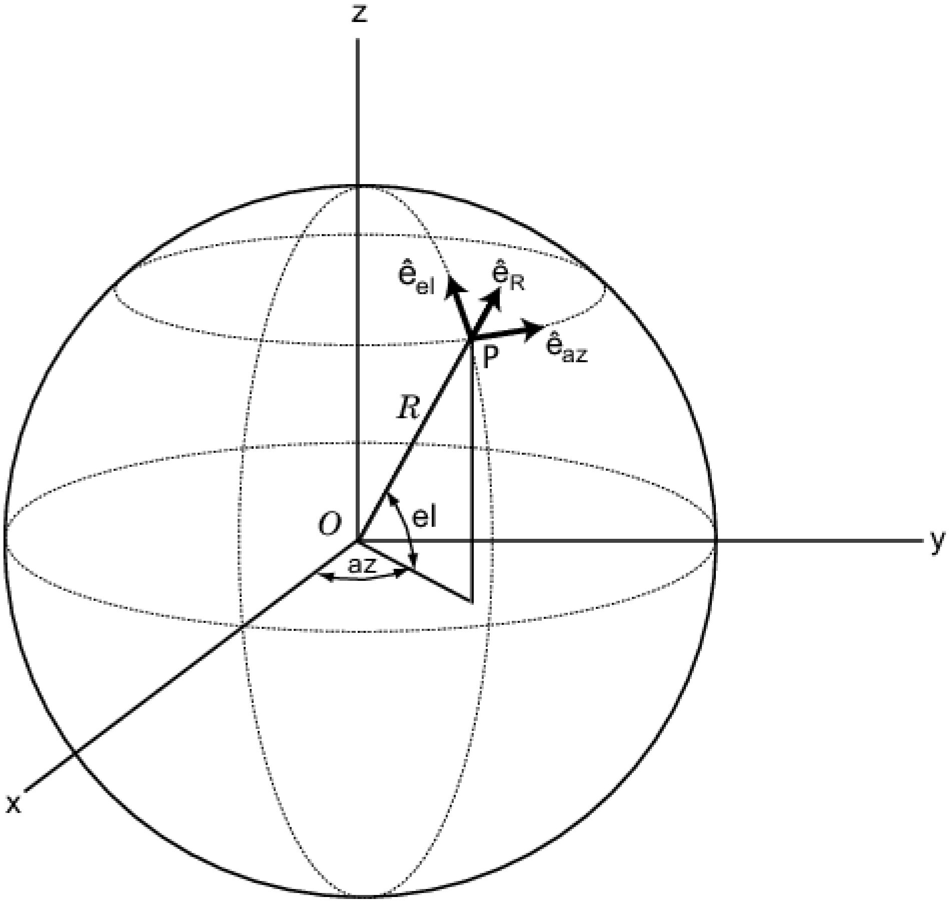
More About

Spherical basis representation of vectors

Spherical basis vectors are a local set of basis vectors which point along the radial and angular directions at any point in space.

The spherical basis is a set of three mutually orthogonal unit vectors ($\hat{\mathbf{e}}_{az}$, $\hat{\mathbf{e}}_{el}$, $\hat{\mathbf{e}}_{R}$) defined at a point on the sphere. The first unit vector points along lines of azimuth at constant radius and elevation. The second points along the lines of elevation at constant azimuth and radius. Both are tangent to the surface of the sphere. The third unit vector points radially outward.

The orientation of the basis changes from point to point on the sphere but is independent of *R* so as you move out along the radius, the basis orientation stays the same. The following figure illustrates the orientation of the spherical basis vectors as a function of azimuth and elevation:



For any point on the sphere specified by *az* and *el*, the basis vectors are given by:

$$\begin{aligned}
\widehat{\mathbf{e}}_{\mathbf{az}} &= -\sin(az)\widehat{\mathbf{i}} + \cos(az)\widehat{\mathbf{j}} \\
\widehat{\mathbf{e}}_{\mathbf{el}} &= -\sin(el)\cos(az)\widehat{\mathbf{i}} - \sin(el)\sin(az)\widehat{\mathbf{j}} + \cos(el)\widehat{\mathbf{k}} \\
\widehat{\mathbf{e}}_{\mathbf{R}} &= \cos(el)\cos(az)\widehat{\mathbf{i}} + \cos(el)\sin(az)\widehat{\mathbf{j}} + \sin(el)\widehat{\mathbf{k}} \quad .
\end{aligned}$$

Any vector can be written in terms of components in this basis as $\mathbf{v} = v_{az}\widehat{\mathbf{e}}_{\mathbf{az}} + v_{el}\widehat{\mathbf{e}}_{\mathbf{el}} + v_R\widehat{\mathbf{e}}_{\mathbf{R}}$. The transformations between spherical basis components and Cartesian components take the form

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} -\sin(az) & -\sin(el)\cos(az) & \cos(el)\cos(az) \\ \cos(az) & -\sin(el)\sin(az) & \cos(el)\sin(az) \\ 0 & \cos(el) & \sin(el) \end{bmatrix} \begin{bmatrix} v_{az} \\ v_{el} \\ v_R \end{bmatrix}$$

.

and

$$\begin{bmatrix} v_{az} \\ v_{el} \\ v_R \end{bmatrix} = \begin{bmatrix} -\sin(az) & \cos(az) & 0 \\ -\sin(el)\cos(az) & -\sin(el)\sin(az) & \cos(el) \\ \cos(el)\cos(az) & \cos(el)\sin(az) & \sin(el) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}.$$

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also
`sph2cartvec`

Introduced in R2020a

cranerainpl

RF signal attenuation due to rainfall using Crane model

Syntax

```
L = cranerainpl(range,freq,rainrate)
L = cranerainpl(range,freq,rainrate,elev)
L = cranerainpl(range,freq,rainrate,elev,tau)
```

Description

`L = cranerainpl(range,freq,rainrate)` returns the signal attenuation, `L`, due to rain based on the Crane rain model [1]. Signal attenuation is a function of the signal path length, `range`, the signal frequency, `freq`, and the rain rate, `rainrate`. The rain rate is defined as the long-term statistical rain rate. The attenuation model applies only for frequencies from 1 GHz to 1000 GHz and is valid for ranges up to 22.5 km. The Crane model accounts for the cellular nature of rainstorms.

`L = cranerainpl(range,freq,rainrate,elev)` also specifies the elevation angle, `elev`, of the signal path.

`L = cranerainpl(range,freq,rainrate,elev,tau)` also specifies the polarization tilt angle, `tau`, of the signal.

Examples

Compare Attenuation for Two Rain Rates Using Crane Model

Use the Crane rain model to compute the signal attenuation caused by rain for a 20 GHz signal sent over a distance of 10 km. Use rain rates of 10.0 and 100.0 mm/hr.

First, set the rain rate to 10 mm/hr.

```
rr = 10.0;
L = cranerainpl(10e3,20.0e9,rr)
```

```
L = 12.5988
```

Repeat the computation using a rain rate of 100.0 mm/hr.

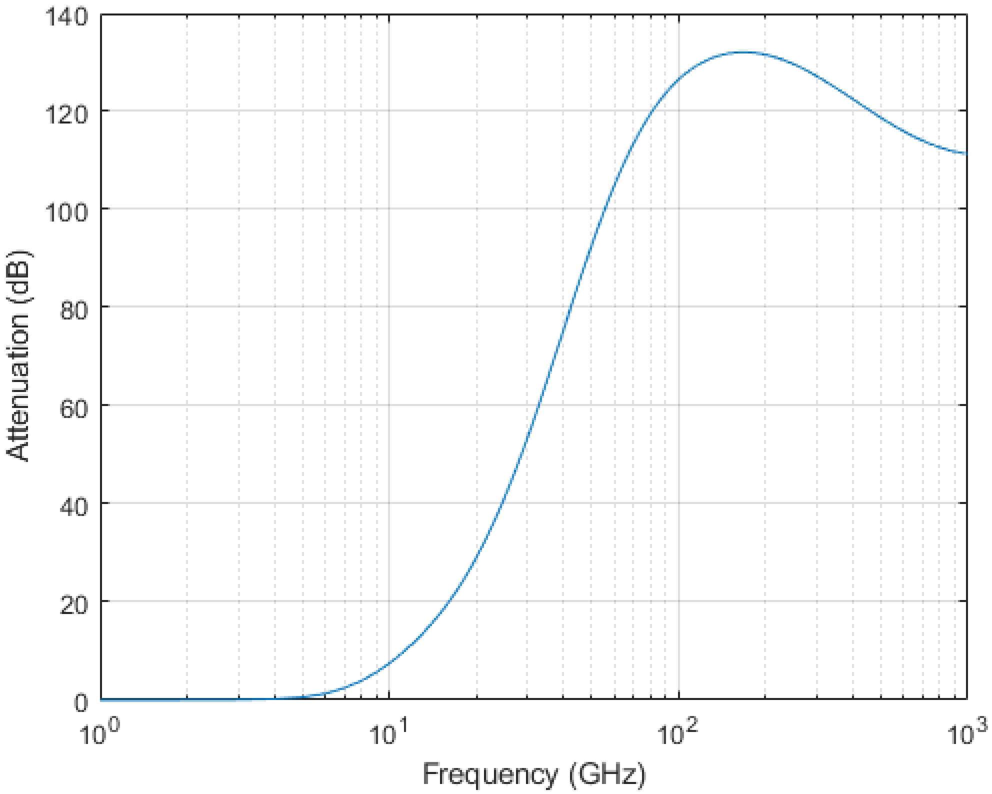
```
rr = 100.0;
L = cranerainpl(10e3,20.0e9,rr)
```

```
L = 73.1912
```

Rain Attenuation as a Function of Frequency Using Crane Model

Plot the signal attenuation due to rain for signals in the frequency range from 1 to 1000 GHz. Use the Crane model to compute the attenuation for a rain rate of 30.0 mm/hr and a signal path distance of 10 km.

```
rr = 30.0;
freq = [1:1000]*1e9;
L = cranerainpl(10e3,freq,rr);
semilogx(freq/1e9,L)
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
```



Rain Attenuation as a Function of Elevation Using Crane Model

Plot the signal attenuation due to rain as a function of elevation angle. Elevation angles vary from 0 to 90 degrees. Assume a path distance of 10 km and a signal frequency of 10 GHz. The rain rate is 100 mm/hr.

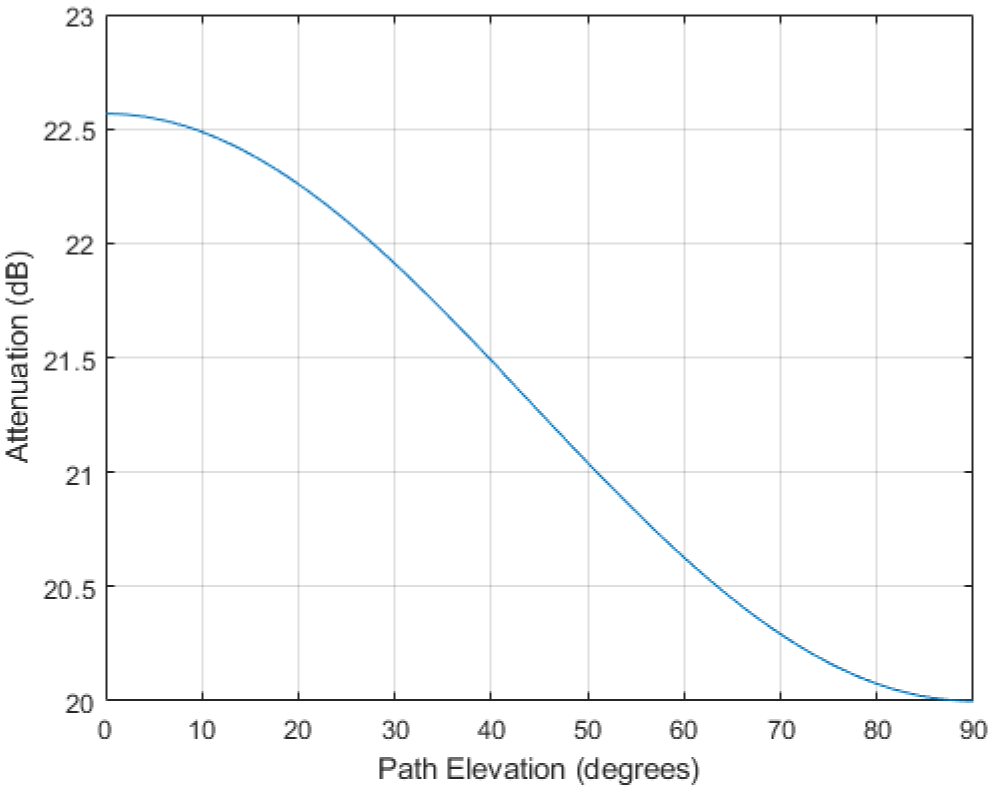
```
rr = 100.0;
```

Set the elevation angles, frequency, and path length.

```
elev = [0:1:90];
freq = 10.0e9;
rng = 10e3*ones(size(elev));
```

Compute and plot the loss.

```
L = cranerainpl(rng,freq,rr,elev);
plot(elev,L)
grid
xlabel('Path Elevation (degrees)')
ylabel('Attenuation (dB)')
```



Rain Attenuation as a Function of Polarization Using Crane Model

Plot the signal attenuation due to rainfall as a function of the polarization tilt angle. Assume a path distance of 10 km, a signal frequency of 10 GHz, and a path elevation angle of 0 degrees. Set the rainfall rate to 70 mm/hour. Plot the signal attenuation against polarization tilt angle.

Set the polarization tilt angle to vary from -90 to 90 degrees.

```
tau = -90:90;
```

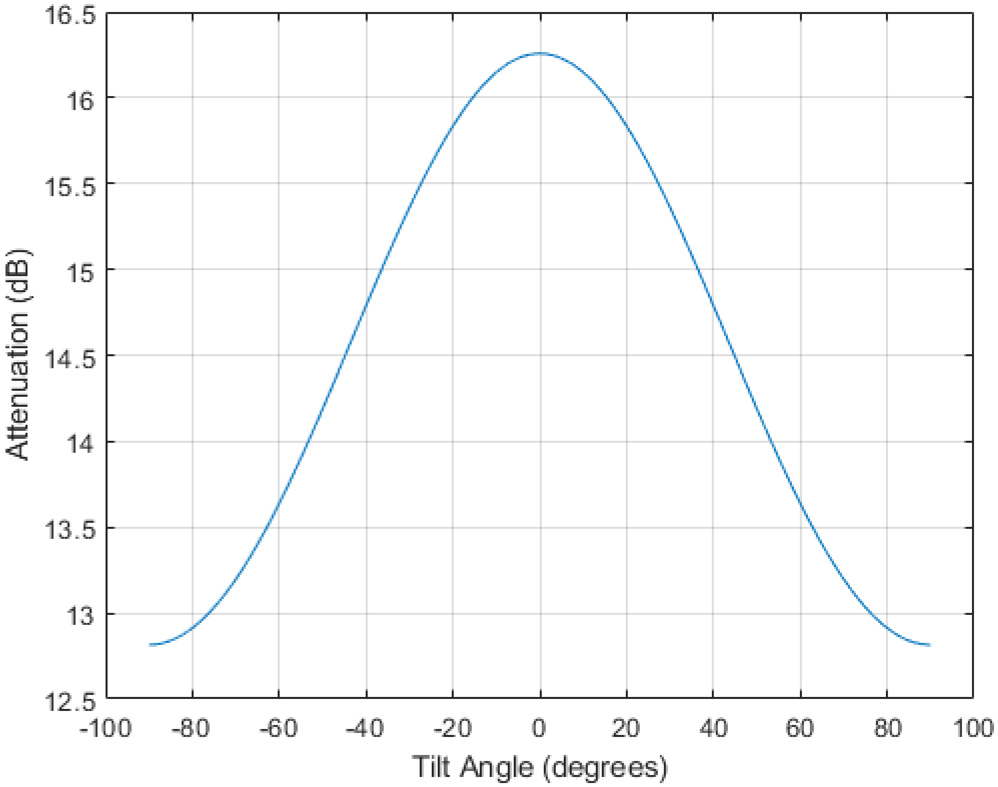
Set the elevation angle, frequency, path distance, and rain rate.

```
elev = 0;
freq = 10.0e9;
```

```
rng = 10e3*ones(size(tau));
rr = 70.0;
```

Compute and plot the attenuation.

```
L = cranerainpl(rng,freq,rr,elev,tau);
plot(tau,L)
grid
xlabel('Tilt Angle (degrees)')
ylabel('Attenuation (dB)')
```



Input Arguments

range — Signal path length

positive scalar | real-valued 1-by-*M* vector of positive values | real-valued *M*-by-1 vector of positive values

Signal path length, specified as a positive scalar, a real-valued 1-by-*M* vector of positive values, or real-valued *M*-by-1 vector of positive values. Units are in meters.

Example: `[13000.0,14000.0]`

freq — Signal frequency

positive scalar | real-valued 1-by-*N* vector of positive values | real-valued *N*-by-1 vector of positive values

Signal frequency, specified as a positive scalar, a real-valued 1-by-*N* vector of positive values, or a real-valued *N*-by-1 vector of positive values. Units are in Hz. Frequencies must lie in the range 1–1000 GHz.

Example: `[2.0:2:10.0]*1e9]`

`rainrate` — Rain rate

nonnegative scalar

Rain rate, specified as a nonnegative scalar. Rain rate represents the long-term statistical rainfall rate provided by Crane (see [1]). Units are in mm/hr.

Example: `100.5`

`elev` — Signal path elevation angle

0.0 (default) | scalar | real-valued 1-by-*M* vector | real-valued *M*-by-1 vector

Signal path elevation angle, specified as a real-valued scalar, or real-valued *M*-by-1 or real-valued 1-by-*M* vector. Units are in degrees between –90° and 90°.

- If `elev` is a scalar, all propagation paths have the same elevation angle.
- If `elev` is a vector, its length must match the length of `range` and each element in `elev` corresponds to a propagation range.

Example: `[0,45]`

`tau` — Tilt angle of signal polarization ellipse

0.0 (default) | scalar | real-valued 1-by-*M* vector | real-valued *M*-by-1 vector

Tilt angle of the signal polarization ellipse, specified as a scalar, a real-valued 1-by-*M* vector, or a real-valued *M*-by-1 vector. Tilt angle values are in the range –90° and 90°, inclusive. Units are in degrees.

- If `tau` is a scalar, all signals have the same tilt angle.
- If `tau` is a vector, its length must match the length of `range`. In that case, each element in `tau` corresponds to a propagation path in `range`.

The tilt angle is defined as the angle between the semimajor axis of the polarization ellipse and the *x*-axis. Because the ellipse is symmetrical, a tilt angle of 10° corresponds to the same polarization state as a tilt angle of -80°. Thus, the tilt angle need only be specified between ±90°.

Example: `[45,30]`

Output Arguments

`L` — Signal attenuation

real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

More About

Crane Rainfall Attenuation Model

The Crane model calculates the attenuation of signals that propagate through regions of rainfall. The model was developed for use on Earth–space or terrestrial propagation paths and is a commonly-used method for the calculation of rain attenuation. The model is based on observations of rain rate, rain structure, and the vertical variation of temperature in the atmosphere. The Crane model (see *Electromagnetic Wave Propagation through Rain*) is primarily applicable to North America. The Crane model generally predicts losses greater than those of the ITU rain attenuation model used in the `rainpl` function. However, the uncertainty of both models and the short-term variation of fade can be large.

The ITU and Crane models are very similar but have some differences. The ITU and Crane rain attenuation models both require statistical annual rainfall rates and utilize an effective path length reduction factor to account for the cellular nature of storms. The 0.01% rainfall rate tables provided by Crane and the ITU are different. The Crane rainfall zones are similar to the ITU zones but more zones are defined in the US than in the ITU model. The ITU rainfall zones are discussed in *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The Crane model is more complex consisting of a piecewise combination of path profiles composed of exponential functions.

The Crane model utilizes two exponential functions to span the distance from 0 to 22.5 km.

- For $\delta < D < 22.5$,

  $$L = \gamma\left(\frac{e^{y\delta} - 1}{y} - \frac{b^{\alpha}e^{z\delta}}{z} + \frac{b^{\alpha}e^{zD}}{z}\right)$$

- For $0 < D < \delta$,

  $$L = \gamma\left(\frac{e^{yD} - 1}{y}\right)$$

where

- L = path attenuation (dB)
- □□= propagation distance (km)
- R = statistical 0.01% rain rate (mm/hr)
- γ = specific attenuation identical to that calculated in `rainpl`.

  $$\gamma_R = kR^{\alpha},$$

  The parameters k and α depend on the frequency, the polarization state, and the elevation angle of the signal path. These coefficients, given by both Crane *Electromagnetic Wave Propagation through Rain* and the *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*, are identical and are valid from 1 GHz to 1000 GHz. The specific attenuation model is valid for frequencies from 1–1000 GHz. Rainfall specific attenuation is computed according to the ITU rainfall model in *ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*.

The remaining parameters are empirical constants defined as:

- $b = 2.3R^{-0.17}$

- c = 0.026 - 0.03ln R
- δ = 3.8 - 0.6 ln R
- u = ln $(be^{c\delta})/\delta$
- $y = \alpha u$
- $z = \alpha c$

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the propagation distance.

You can also apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Crane, Robert K. *Electromagnetic Wave Propagation through Rain*. Wiley, 1996.

[2] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. P Series, Radiowave Propagation 2005.

[3] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.530-17: Propagation data and prediction methods required for the design of terrestrial line-of-sight systems*. 2017.

[4] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.837-7: Characteristics of precipitation for propagation modelling*. 6/2017

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fogpl` | `fspl` | `gaspl` | `rainpl`

Introduced in R2020a

fogpl

RF signal attenuation due to fog and clouds

Syntax

```
L = fogpl(R,freq,T,den)
```

Description

`L = fogpl(R,freq,T,den)` returns attenuation, `L`, when signals propagate in fog or clouds. `R` represents the signal path length. `freq` represents the signal carrier frequency, `T` is the ambient temperature, and `den` specifies the liquid water density in the fog or cloud.

The `fogpl` function applies the International Telecommunication Union (ITU) cloud and fog attenuation model to calculate path loss of signals propagating through clouds and fog. See [1] (Phased Array System Toolbox). Fog and clouds are the same atmospheric phenomenon, differing only by height above ground. Both environments are parametrized by their liquid water density. Other model parameters include signal frequency and temperature. This function applies to cases when the signal path is contained entirely in a uniform fog or cloud environment. The liquid water density does not vary along the signal path. The attenuation model applies only for frequencies at 10–1000 GHz.

Examples

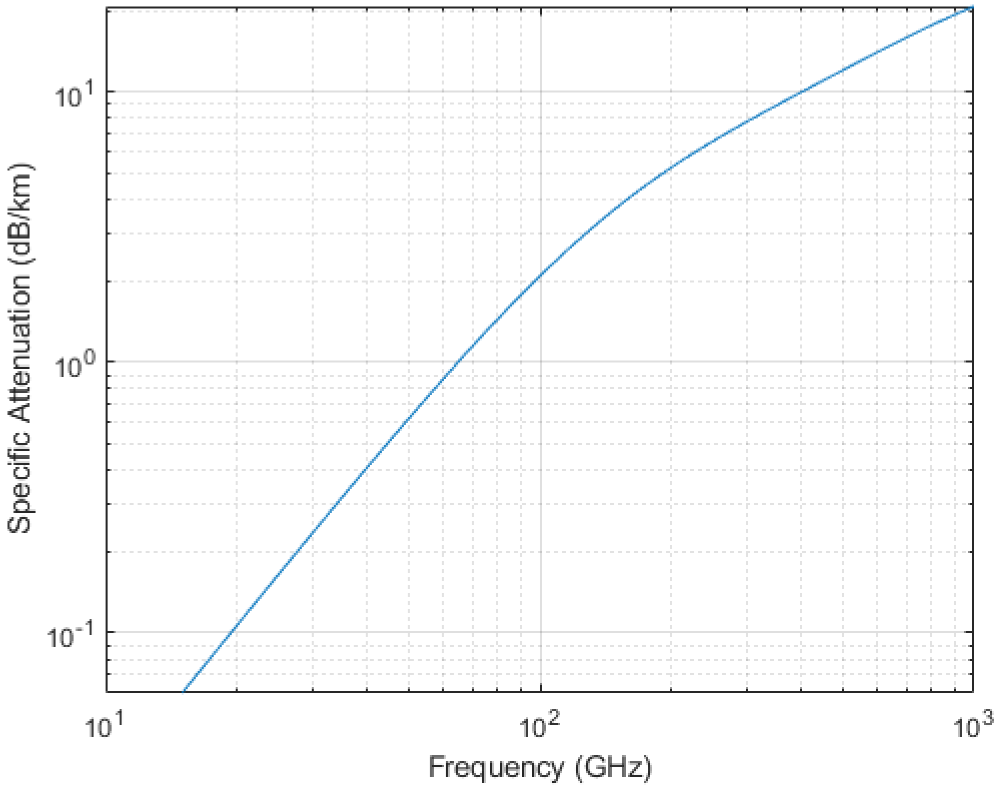
Attenuation in Cumulus Clouds

Compute the attenuation of signals propagating through a cloud that is 1 km long at 1000 meters altitude. Compute the attenuation for frequencies from 15 to 1000 GHz. A typical value for the cloud liquid water density is 0.5 g/m^3. Assume the atmospheric temperature at 1000 meters is 20˚C.

```
R = 1000.0;
freq = [15:5:1000]*1e9;
T = 20.0;
lwd = 0.5;
L = fogpl(R,freq,T,lwd);
```

Plot the specific attenuation as a function of frequency. Specific attenuation is the attenuation or loss per kilometer.

```
loglog(freq/1e9,L)
grid
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB/km)')
```



Input Arguments

R — Signal path length

positive real-valued scalar | *M*-by-1 nonnegative real-valued vector | 1-by-*M* nonnegative real-valued vector

Signal path length, specified as a scalar or as an *M*-by-1 or 1-by-*M* vector of nonnegative real-values. Total attenuation is the specific attenuation multiplied by the path length. Units are meters.

Example: `[1300.0,1400.0]`

`freq` — Signal frequency

positive real-valued scalar | *N*-by-1 nonnegative real-valued column vector | 1-by-*N* nonnegative real-valued row vector

Signal frequency, specified as a positive real-valued scalar or as an *N*-by-1 nonnegative real-valued vector or 1-by-*N* nonnegative real-valued vector. Frequencies must lie in the range 10–1000 GHz.

Example: `[14.0e9,15.0e9]`

T — Ambient temperature

real-valued scalar

Ambient temperature in fog or cloud, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `-10.0`

den — Liquid water density
nonnegative real-valued scalar

Liquid water density, specified as a nonnegative real-valued scalar. Units are g/m^3. Typical values for liquid water density in fog range from approximately 0.05 g/m^3 for medium fog to approximately 0.5 g/m^3 for thick fog. For medium fog, visibility is about 300 meters. For heavy fog, visibility is about 50 meters. Cumulus cloud liquid water density is typically 0.5 g/m^3.

Example: `0.01`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

More About

Fog and Cloud Attenuation Model

This model calculates the attenuation of signals that propagate through fog or clouds.

Fog and cloud attenuation are the same atmospheric phenomenon. The ITU model, *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog* is used. The model computes the specific attenuation (attenuation per kilometer), of a signal as a function of liquid water density, signal frequency, and temperature. The model applies to polarized and nonpolarized fields. The formula for specific attenuation at each frequency is

$$\gamma_c = K_l(f)M,$$

where *M* is the liquid water density in gm/m^3. The quantity $K_l(f)$ is the specific attenuation coefficient and depends on frequency. The cloud and fog attenuation model is valid for frequencies 10–1000 GHz. Units for the specific attenuation coefficient are (dB/km)/(g/m^3).

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length *R*. Total attenuation is $L_c = R\gamma_c$.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply narrowband attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.840-6: Attenuation due to clouds and fog*. 2013.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fspl` | `gaspl` | `rainpl`

Introduced in R2017b

fspl

Free space path loss

Syntax

```
L = fspl(R,lambda)
```

Description

`L = fspl(R,lambda)` returns the free space path loss in decibels for a waveform with wavelength `lambda` propagated over a distance of `R` meters. The minimum value of `L` is zero, indicating no path loss.

Examples

Calculate Free-Space Path Loss

Calculate the free-space path loss (in dB) of a 10 GHz radar signal over a distance of 10 km.

```
fc = 10.0e9;
lambda = physconst('LightSpeed')/fc;
R = 10e3;
L = fspl(R,lambda)
```

```
L = 132.4478
```

Input Arguments

R — Propagation distance of signal
real-valued 1-by-*M* or *M*-by-1 vector

Units are in meters.

lambda — Speed of propagation divided by the signal frequency
real-valued 1-by-*N* or *N*-by-1 vector

Wavelength units are meters.

Output Arguments

L — Path loss in decibels
M-by-*N* non-negative matrix. A value of zero signifies no path loss.

When `lambda` is a scalar, `L` has the same dimensions as `R`.

More About

Free Space Path Loss

The free-space path loss, *L*, in decibels is:

$$L = 20\log_{10}(\frac{4\pi R}{\lambda})$$

This formula assumes that the target is in the far-field of the transmitting element or array. In the near-field, the free-space path loss formula is not valid and can result in a loss smaller than 0 dB, equivalent to a signal gain. For this reason, the loss is set to 0 dB for range values $R \le \lambda/4\pi$.

References

[1] Proakis, J. *Digital Communications*. New York: McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fogpl` | `gaspl` | `rainpl`

Introduced in R2017b

gaspl

RF signal attenuation due to atmospheric gases

Syntax

```
L = gaspl(range,freq,T,P,den)
```

Description

`L = gaspl(range,freq,T,P,den)` returns the attenuation, `L`, when signals propagate through the atmosphere. `range` represents the signal path length, and `freq` represents the signal carrier frequency. `T` represents the ambient temperature, `P` represents the atmospheric pressure, and `den` represents the atmospheric water vapor density.

The `gaspl` function applies the International Telecommunication Union (ITU) atmospheric gas attenuation model [1] to calculate path loss for signals primarily due to oxygen and water vapor. The model computes attenuation as a function of ambient temperature, pressure, water vapor density, and signal frequency. The function requires that the signal path is contained entirely in a uniform environment. Atmospheric parameters do not vary along the signal path. The attenuation model applies only for frequencies at 1–1000 GHz.

Examples

Atmospheric Gas Attenuation Spectrum

Compute the attenuation spectrum from 1 to 1000 GHz for an atmospheric pressure of 101.300 kPa and a temperature of 15˚C. Plot the spectrum for a water vapor density of 7.5 g/m^3 and then plot the spectrum for dry air (zero water vapor density).

Set the attenuation frequencies.

```
freq = [1:1000]*1e9;
```

Assume a 1 km path distance.

```
R = 1000.0;
```

Compute the attenuation for air containing water vapor.

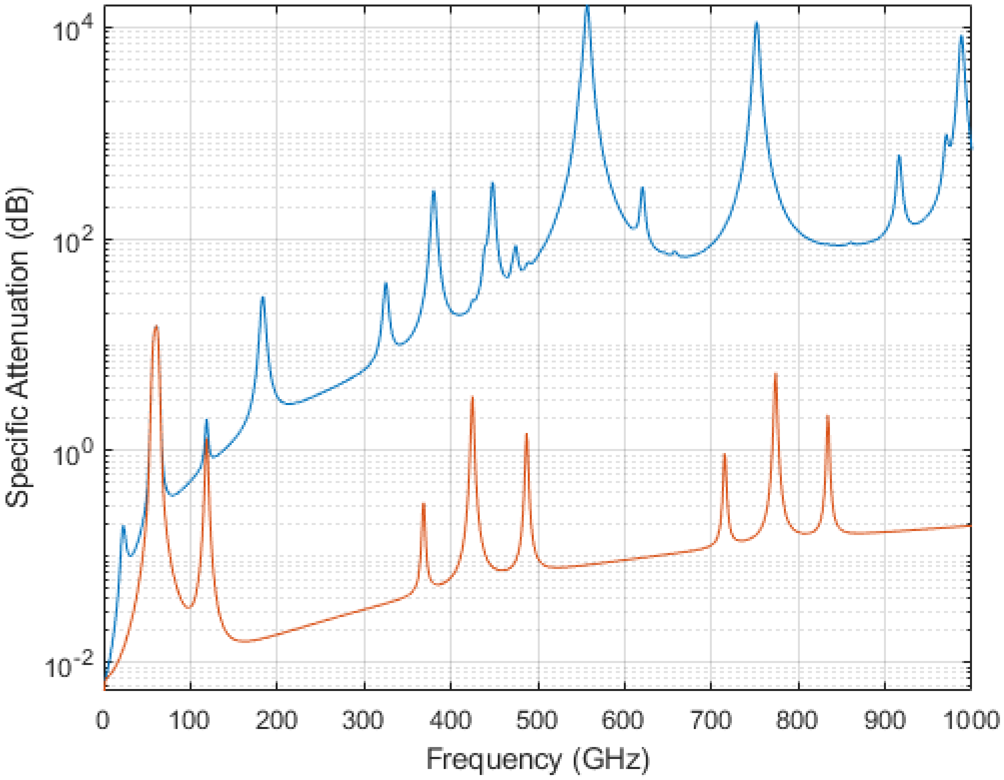
```
T = 15;
P = 101300.0;
W = 7.5;
L = gaspl(R,freq,T,P,W);
```

Compute the attenuation for dry air.

```
L0 = gaspl(R,freq,T,P,0.0);
```

Plot the attenuations.

```
semilogy(freq/1e9,L)
hold on
semilogy(freq/1e9,L0)
grid
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB)')
hold off
```



Plot Attenuation Due to Atmospheric Gases and Free Space

First, plot the specific attenuation of atmospheric gases for frequencies from 1 GHz to 1000 GHz. Assume a sea-level dry air pressure of 101.325e5 kPa and a water vapor density of 7.5 g/m^3. The air temperature is 20˚C. Specific attenuation is defined as dB loss per kilometer. Then, plot the actual attenuation at 10 GHz for a span of ranges.

Plot Specific Atmospheric Gas Attenuation

Set the atmosphere temperature, pressure, water vapor density.

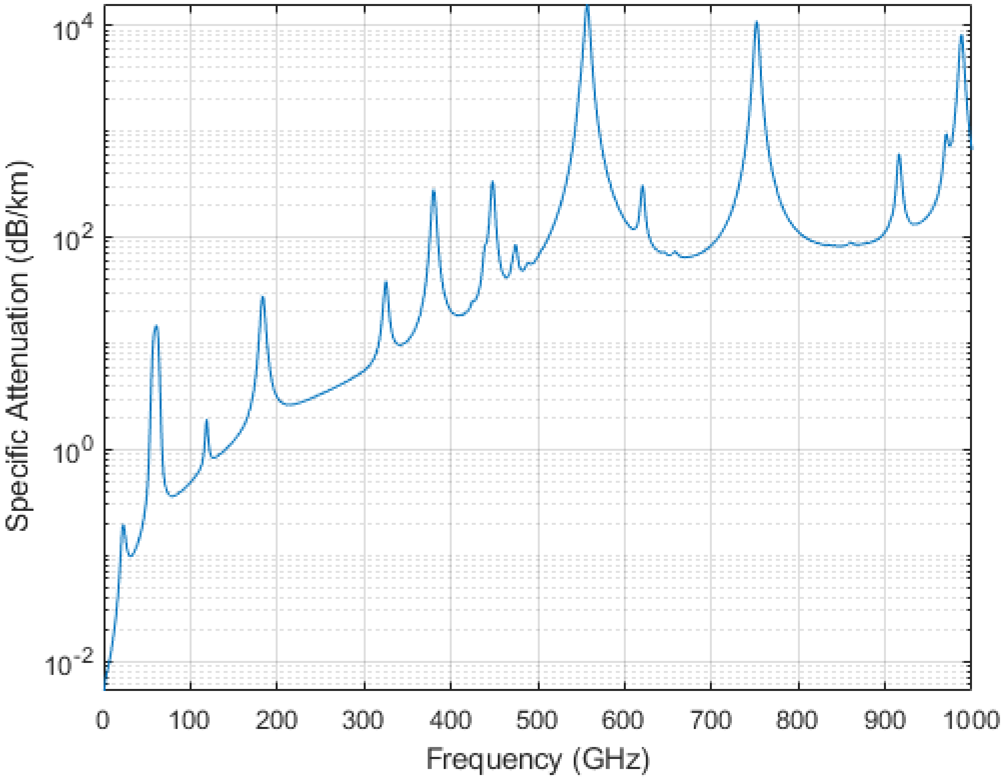
```
T = 20.0;
Patm = 101.325e3;
rho_wv = 7.5;
```

Set the propagation distance, speed of light, and frequencies.

```
km = 1000.0;
c = physconst('LightSpeed');
freqs = [1:1000]*1e9;
```

Compute and plot the atmospheric gas loss.

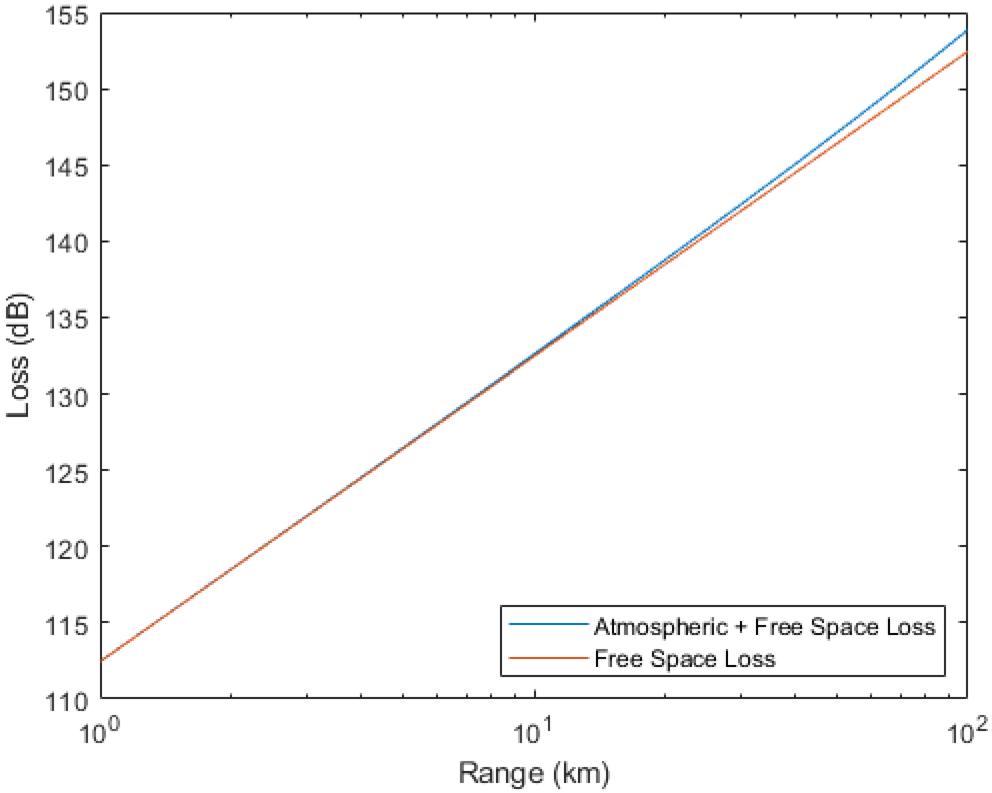
```
loss = gaspl(km,freqs,T,Patm,rho_wv);
semilogy(freqs/1e9,loss)
grid on
xlabel('Frequency (GHz)')
ylabel('Specific Attenuation (dB/km)')
```



Plot Actual Atmospheric and Free Space Attenuation

Compute both free space loss and atmospheric gas loss at 10 GHz for ranges from 1 to 100 km. The frequency corresponds to an *X*-band radar. Then, plot the free space loss and the total (atmospheric + free space) loss.

```
ranges = [1:100]*1000;
freq_xband = 10e9;
loss_gas = gaspl(ranges,freq_xband,T,Patm,rho_wv);
lambda = c/freq_xband;
loss_fsp = fspl(ranges,lambda);
semilogx(ranges/1000,loss_gas + loss_fsp.',ranges/1000,loss_fsp)
legend('Atmospheric + Free Space Loss','Free Space Loss','Location','SouthEast')
xlabel('Range (km)')
ylabel('Loss (dB)')
```



Input Arguments

`range` — Signal path length

nonnegative real-valued scalar | *M*-by-1 nonnegative real-valued column vector | 1-by-*M* nonnegative real-valued row vector

Signal path length used to compute attenuation, specified as a nonnegative real-valued scalar or vector. You can specify multiple path lengths simultaneously. Units are in meters.

Example: `[13000.0,14000.0]`

`freq` — Signal frequency

positive real-valued scalar | *N*-by-1 nonnegative real-valued column vector | 1-by-*N* nonnegative real-valued row vector

Signal frequency, specified as a positive real-valued scalar, or as an *N*-by-1 nonnegative real-valued vector or 1-by-*N* nonnegative real-valued vector. You can specify multiple frequencies simultaneously. Frequencies must lie in the range 1–1000 GHz. Units are in hertz.

Example: `[1.4e9,2.0e9]`

`T` — Ambient temperature

real-valued scalar

Ambient temperature, specified as a real-valued scalar. Units are in degrees Celsius.

Example: `-10.0`

P — Dry air pressure
positive real-valued scalar

Dry air pressure, specified as a positive real-valued scalar. Units are in Pa. One standard atmosphere at sea level is 101325 Pa.

Example: `101300.0`

den — Water vapor density
nonnegative real-valued scalar

Water vapor density or absolute humidity, specified as a nonnegative real-valued scalar. Units are g/m^3. The maximum water vapor density of air at 30° C is approximately 30.0 g/m^3. The maximum water vapor density of air at 0°C is approximately 5.0 g/m^3.

Example: `4.0`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

More About

Atmospheric Gas Attenuation Model

This model calculates the attenuation of signals that propagate through atmospheric gases.

Electromagnetic signals attenuate when they propagate through the atmosphere. This effect is due primarily to the absorption resonance lines of oxygen and water vapor, with smaller contributions coming from nitrogen gas. The model also includes a continuous absorption spectrum below 10 GHz. The ITU model *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* is used. The model computes the specific attenuation (attenuation per kilometer) as a function of temperature, pressure, water vapor density, and signal frequency. The atmospheric gas model is valid for frequencies from 1–1000 GHz and applies to polarized and nonpolarized fields.

The formula for specific attenuation at each frequency is

$$\gamma = \gamma_o(f) + \gamma_w(f) = 0.1820 f N''(f)\,.$$

The quantity *N"()* is the imaginary part of the complex atmospheric refractivity and consists of a spectral line component and a continuous component:

$$N''(f) = \sum_i S_i F_i + N''_D(f)$$

The spectral component consists of a sum of discrete spectrum terms composed of a localized frequency bandwidth function, $F(f)_{\mathrm{i}}$, multiplied by a spectral line strength, S_{i}. For atmospheric oxygen, each spectral line strength is

$$S_i = a_1 \times 10^{-7}\left(\frac{300}{T}\right)^3 \exp\left[a_2(1 - \left(\frac{300}{T}\right)\right]P\,.$$

For atmospheric water vapor, each spectral line strength is

$$S_i = b_1 \times 10^{-1}\left(\frac{300}{T}\right)^{3.5}\exp\left[b_2(1 - \left(\frac{300}{T}\right)\right]W\,.$$

P is the dry air pressure, W is the water vapor partial pressure, and T is the ambient temperature. Pressure units are in hectoPascals (hPa) and temperature is in degrees Kelvin. The water vapor partial pressure, W, is related to the water vapor density, ρ, by

$$W = \frac{\rho T}{216.7}\,.$$

The total atmospheric pressure is $P + W$.

For each oxygen line, S_i depends on two parameters, a_1 and a_2. Similarly, each water vapor line depends on two parameters, b_1 and b_2. The ITU documentation cited at the end of this section contains tabulations of these parameters as functions of frequency.

The localized frequency bandwidth functions $F_i(f)$ are complicated functions of frequency described in the ITU references cited below. The functions depend on empirical model parameters that are also tabulated in the reference.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the path length, R. Then, the total attenuation is $L_g= R(\gamma_o + \gamma_w)$.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands, and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.676-10: Attenuation by atmospheric gases* 2013.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`fogpl` | `fspl` | `rainpl`

Introduced in R2017b

global2localcoord

Convert global to local coordinates

Syntax

```
lclCoord = global2localcoord(gCoord, OPTION)
gCoord = global2localcoord( ___ ,localOrigin)
gCoord = global2localcoord( ___ ,localAxes)
```

Description

`lclCoord = global2localcoord(gCoord, OPTION)` converts global coordinates `gCoord` to local coordinates `lclCoord`. `OPTION` determines the type of global-to-local coordinate transformation. In this syntax, the global coordinate origin is located at (0,0,0) and the coordinate axes are the unit vectors in the *x*, *y*, and *z* directions.

`gCoord = global2localcoord( ___ ,localOrigin)` specifies the origin of the local coordinate system, `localOrigin`.

`gCoord = global2localcoord( ___ ,localAxes)` specifies the axes of the local coordinate system, `localAxes`.

Input Arguments

gCoord

Global coordinates in rectangular or spherical coordinate, specified as a 3-by-*N* matrix. Each column represents one set of global coordinates.

If the coordinates are in rectangular form, each column contains the (*x*,*y*,*z*) components. Units are in meters.

If the coordinates are in spherical form, each column contains (*az*,*el*,*r*) components. *az* is the azimuth angle on page 2-1018 in degrees, *el* is the elevation angle on page 2-1018 in degrees, and *r* is the radius in meters.

The origin of the global coordinate system is assumed to be located at (0, 0, 0). The global system axes are the standard unit basis vectors in three-dimensional space, (1, 0, 0), (0, 1, 0), and (0, 0, 1).

OPTION

Type of coordinate transformation, specified as a character vector. Valid types are

OPTION	Transformation
`'rr'`	Global rectangular to local rectangular
`'rs'`	Global rectangular to local spherical
`'sr'`	Global spherical to local rectangular

OPTION	Transformation
'ss'	Global spherical to local spherical

localOrigin

Origin of local coordinate system, specified as a 3-by-*N* matrix containing the rectangular coordinates of the local coordinate system origin with respect to the global coordinate system. *N* must match the number of columns of `gCoord`. Each column represents a separate origin. However, you can specify `localOrigin` as a 3-by-*1* vector. In this case, `localOrigin` is expanded into a 3-by-*N* matrix with identical columns.

Default: `[0;0;0]`

localAxes

Axes of local coordinate system, specified as a 3-by-3-by-*N* array. Each page contains a 3-by-3 matrix representing a different local coordinate system axes. The columns of the 3-by-3 matrices specify the local *x*, *y*, and *z* axes in rectangular form with respect to the global coordinate system. However, you can specify `localAxes` as a single 3-by-3 matrix. In this case, `localAxes` is expanded into a 3-by-3-by-*N* array with identical 3-by-3 matrices. The default is the identity matrix.

Default: `[1 0 0;0 1 0;0 0 1]`

Output Arguments

lclCoord

Local coordinates in rectangular or spherical coordinate form, returned as a 3-by-*N* matrix. The dimensions of `lclCoord` match the dimensions of `gCoord`.

Examples

Convert Global Coordinates to Local Coordinates

Convert global rectangular coordinates, *(0,1,0)*, to local rectangular coordinates. The local coordinate origin is *(1,1,1)*.

```
lclCoord = global2localcoord([0;1;0],'rr',[1;1;1])
```

```
lclCoord = 3×1

    -1
     0
    -1
```

Convert global spherical coordinates to local rectangular coordinates.

```
lclCoord = global2localcoord([45;45;50],'sr',[50;50;50])
```

```
lclCoord = 3×1

  -25.0000
  -25.0000
  -14.6447
```

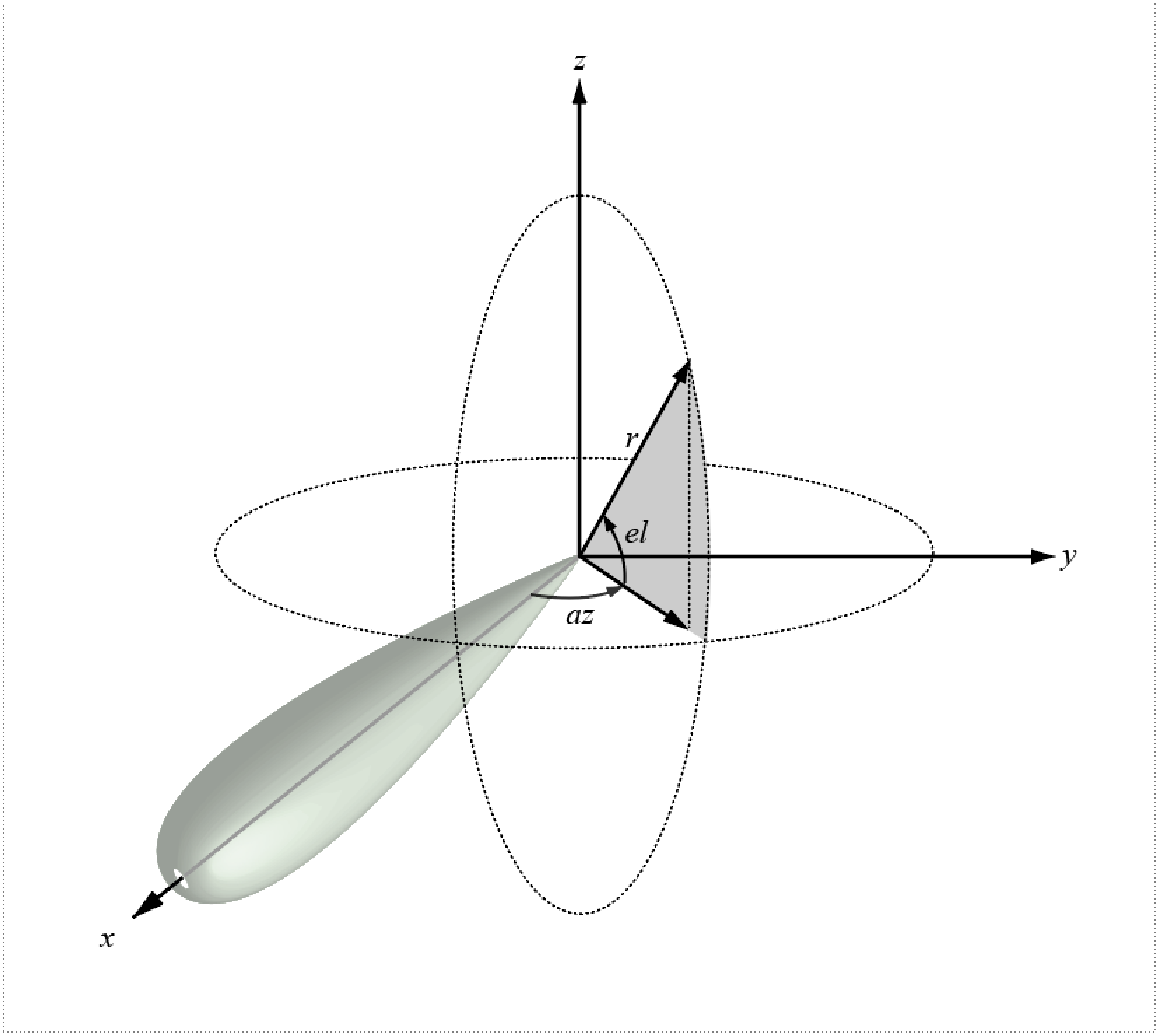
More About

Azimuth and Elevation Angles

The azimuth angle of a vector is the angle between the *x*-axis and the orthogonal projection of the vector onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane. By default, the boresight direction of an element or array is aligned with the positive *x*-axis. The boresight direction is the direction of the main lobe of an element or array.

Note The elevation angle is sometimes defined in the literature as the angle a vector makes with the positive *z*-axis. The MATLAB and Communications Toolbox products do not use this definition.

This figure illustrates the azimuth and elevation angles of a direction vector.



References

[1] Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C*, 2nd Ed. Reading, MA: Addison-Wesley, 1995.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`local2globalcoord` | `rangeangle`

Introduced in R2020a

local2globalcoord

Convert local to global coordinates

Syntax

```
gCoord = local2globalcoord(lclCoord,OPTION)
gCoord = local2globalcoord( ___ ,localOrigin)
gCoord = local2globalcoord( ___ ,localAxes)
```

Description

`gCoord = local2globalcoord(lclCoord,OPTION)` converts local coordinates `lclCoord` to global coordinates `gCoord`. `OPTION` determines the type of local-to-global coordinate transformation.

`gCoord = local2globalcoord( ___ ,localOrigin)` specifies the origin of the local coordinate system, `localOrigin`.

`gCoord = local2globalcoord( ___ ,localAxes)` specifies the axes of the local coordinate system, `localAxes`.

Input Arguments

`lclCoord`

Local coordinates in rectangular or spherical coordinate form, specified as a 3-by-*N* matrix. Each column represents one set of local coordinates.

If the coordinates are in rectangular form, each column contains the (*x*,*y*,*z*) components. Units are in meters.

If the coordinates are in spherical form, each column contains (*az*,*el*,*r*) components. *az* is the azimuth angle on page 2-1018 in degrees, *el* is the elevation angle on page 2-1018 in degrees, and *r* is the radius in meters.

`OPTION`

Types of coordinate transformations, specified as a character vector. Valid values are

OPTION	Transformation
`'rr'`	Local rectangular to global rectangular
`'rs'`	Local rectangular to global spherical
`'sr'`	Local spherical to global rectangular
`'ss'`	Local spherical to global spherical

`localOrigin`

Origin of local coordinate system, specified as a 3-by-*N* matrix containing the rectangular coordinates of the local coordinate system origin with respect to the global coordinate system. *N* must match the number of columns of `gCoord`. Each column represents a separate origin. However, you can specify `localOrigin` as a 3-by-*1* vector. In this case, `localOrigin` is expanded into a 3-by-*N* matrix with identical columns.

Default: `[0;0;0]`

`localAxes`

Axes of local coordinate system, specified as a 3-by-3-by-*N* array. Each page contains a 3-by-3 matrix representing a different local coordinate system axes. The columns of the 3-by-3 matrices specify the local *x*, *y*, and *z* axes in rectangular form with respect to the global coordinate system. However, you can specify `localAxes` as a single 3-by-3 matrix. In this case, `localAxes` is expanded into a 3-by-3-by-*N* array with identical 3-by-3 matrices. The default is the identity matrix.

Default: `[1 0 0;0 1 0;0 0 1]`

Output Arguments

`gCoord`

Glabal coordinates in rectangular or spherical coordinate form, returned as a 3-by-*N* matrix. The dimensions of `gCoord` match the dimensions of `lclCoord`. The origin of the global coordinate system is assumed to be located at (0, 0, 0). The global system axes are the standard unit basis vectors in three-dimensional space, (1, 0, 0), (0, 1, 0), and (0, 0, 1).

Examples

Convert Local Rectangular Coordinates to Global Rectangular Coordinates

Convert from local rectangular coordinates to global rectangular coordinates. The local coordinate origin is a *(1,1,1)*

```
globalcoord = local2globalcoord([0;1;0], 'rr',[1;1;1])
```

```
globalcoord = 3×1

     1
     2
     1
```

Convert Local Spherical Coordinates to Global Rectangular Coordinates

Convert local spherical coordinate to global rectangular coordinate.

```
globalcoord = local2globalcoord([30;45;4],'sr')
```

```
globalcoord = 3×1

    2.4495
    1.4142
    2.8284
```

Convert Two Vectors Between Local and Global Coordinates

Convert two vectors in global coordinates into two vectors in global coordinates using the `global2local` function. Then convert them back to local coordinates using the `local2global` function.

Start with two vectors in global coordinates, *(0,1,0)* and *(1,1,1)*. The local coordinate origins are *(1,5,2)* and *(-4,5,7)*.

```
gCoord = [0 1; 1 1; 0 1]
```

```
gCoord = 3×2

     0     1
     1     1
     0     1
```

```
lclOrig = [1 -4; 5 5; 2 7];
```

Construct two rotation matrices using the rotation functions.

```
lclAxes(:,:,1) = rotz(45)*roty(-15);
lclAxes(:,:,2) = roty(45)*rotx(35);
```

Convert the vectors in global coordinates into local coordinates.

```
lclCoord = global2localcoord(gCoord,'rr',lclOrig,lclAxes)
```

```
lclCoord = 3×2

   -3.9327    7.7782
   -2.1213   -3.6822
   -1.0168    1.7151
```

Convert the vectors in local coordinates back into global coordinates.

```
gCoord1 = local2globalcoord(lclCoord,'rr',lclOrig,lclAxes)
```

```
gCoord1 = 3×2

   -0.0000    1.0000
    1.0000    1.0000
         0    1.0000
```

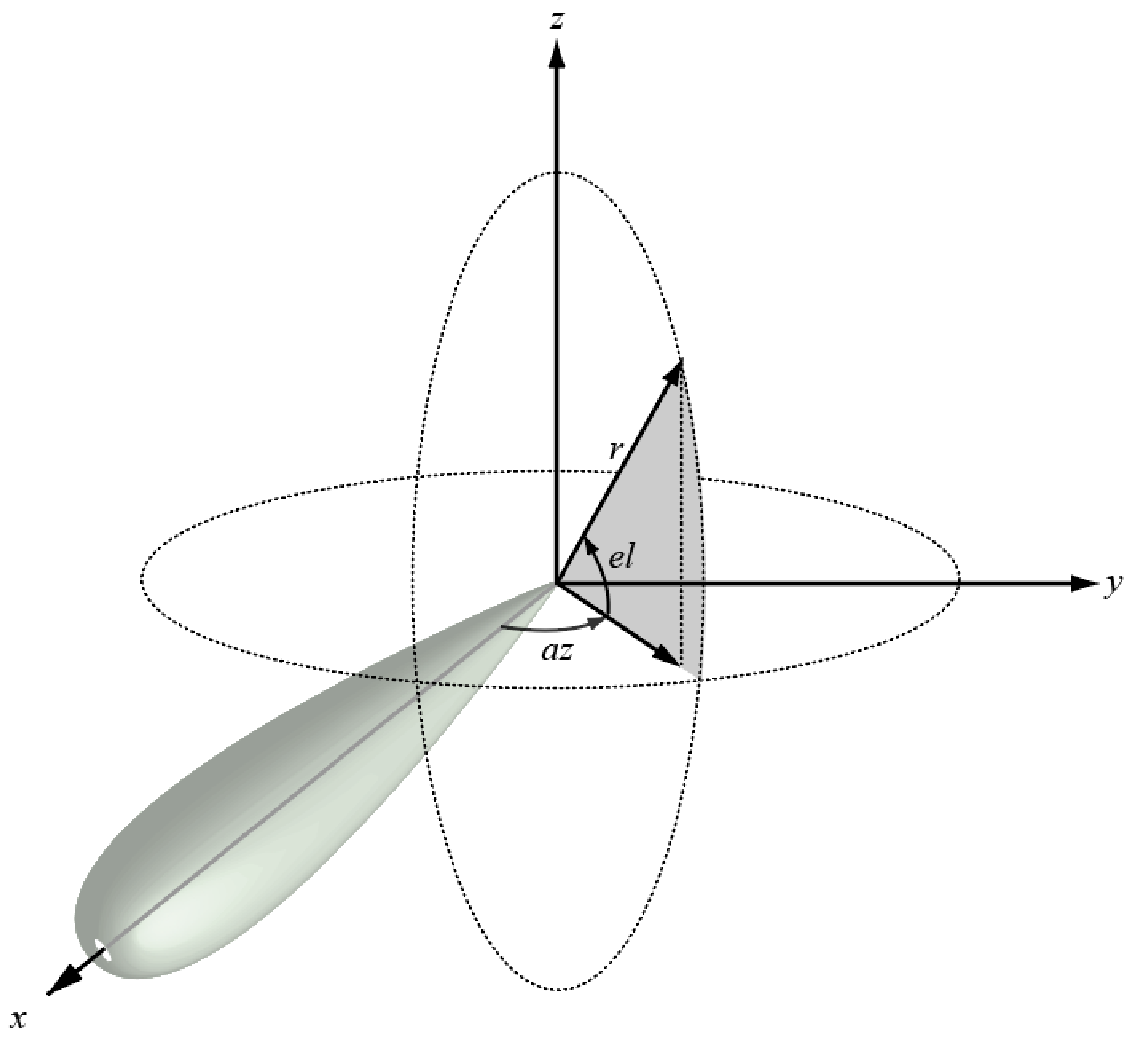
More About

Azimuth Angle, Elevation Angle

The azimuth angle of a vector is the angle between the *x*-axis and the orthogonal projection of the vector onto the *xy* plane. The angle is positive in going from the *x* axis toward the *y* axis. Azimuth angles lie between –180 and 180 degrees. The elevation angle is the angle between the vector and its orthogonal projection onto the *xy*-plane. The angle is positive when going toward the positive *z*-axis from the *xy* plane. By default, the boresight direction of an element or array is aligned with the positive *x*-axis. The boresight direction is the direction of the main lobe of an element or array.

Note The elevation angle is sometimes defined in the literature as the angle a vector makes with the positive *z*-axis. The MATLAB and Communications Toolbox products do not use this definition.

This figure illustrates the azimuth and elevation angles of a direction vector.



References

[1] Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice in C*, 2nd Ed. Reading, MA: Addison-Wesley, 1995.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`global2localcoord` | `rangeangle`

Introduced in R2020a

rainpl

RF signal attenuation due to rainfall

Syntax

```
L = rainpl(range,freq,rainrate)
L = rainpl(range,freq,rainrate,elev)
L = rainpl(range,freq,rainrate,elev,tau)
L = rainpl(range,freq,rainrate,elev,tau,pct)
```

Description

`L = rainpl(range,freq,rainrate)` returns the signal attenuation, `L`, due to rainfall. In this syntax, attenuation is a function of signal path length, `range`, signal frequency, `freq`, and rain rate, `rainrate`. The path elevation angle and polarization tilt angles are assumed to zero.

The `rainpl` function applies the International Telecommunication Union (ITU) rainfall attenuation model to calculate path loss of signals propagating in a region of rainfall [1]. The function applies when the signal path is contained entirely in a uniform rainfall environment. Rain rate does not vary along the signal path. The attenuation model applies only for frequencies at 1–1000 GHz.

`L = rainpl(range,freq,rainrate,elev)` also specifies the elevation angle, `elev`, of the propagation path.

`L = rainpl(range,freq,rainrate,elev,tau)` also specifies the polarization tilt angle, `tau`, of the signal.

`L = rainpl(range,freq,rainrate,elev,tau,pct)` also specifies the specified percentage of time, `pct`. `pct` is a scalar in the range of 0.001–1, inclusive. The attenuation, `L`, is computed from a power law using the long-term statistical 0.01% rain rate (in mm/h).

Examples

Signal Attenuation Due to Rainfall

Compute the signal attenuation due to rainfall for a 20 GHz signal over a distance of 10 km in light and heavy rain.

Propagate the signal in a light rainfall of 1 mm/hr.

```
rr = 1.0;
L = rainpl(10000,20.0e9,rr)
```

```
L = 1.3009
```

Propagate the signal in a heavy rainfall of 10 mm/hr.

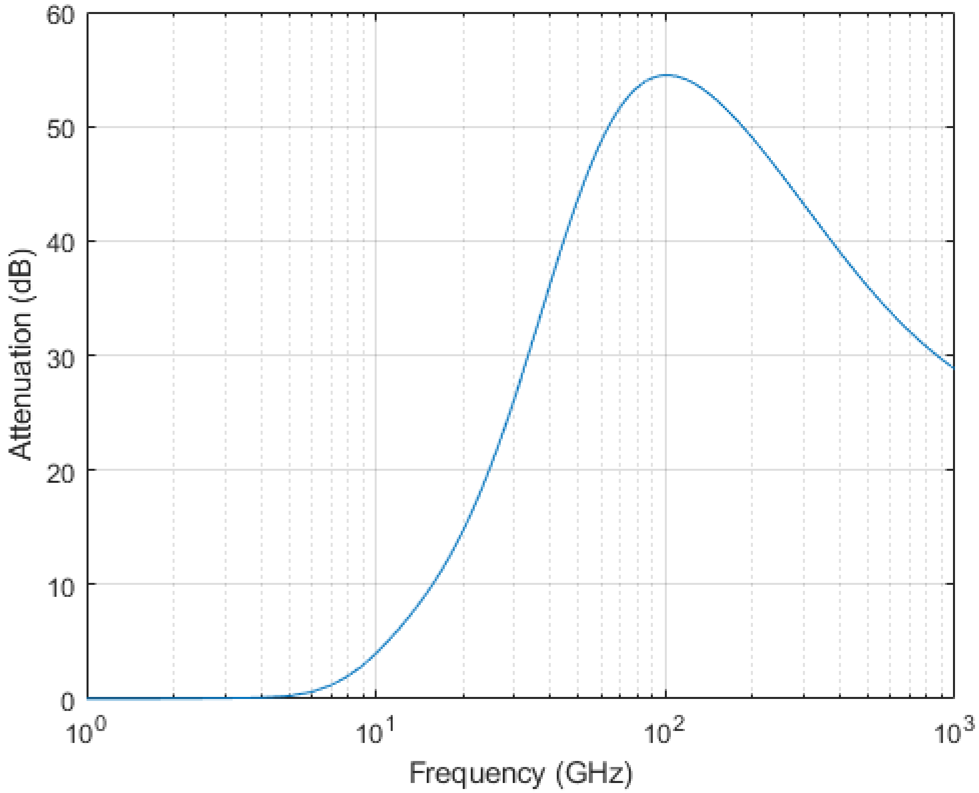
```
rr = 10.0;
L = rainpl(10000,20.0e9,rr)
```

```
L = 8.1584
```

Signal Attenuation Due to Rainfall as Function of Frequency

Plot the signal attenuation due to a 20 mm/hr statistical rainfall for signals in the frequency range from 1 to 1000 GHz. The path distance is 10 km.

```
rr = 20.0;
freq = [1:1000]*1e9;
L = rainpl(10000,freq,rr);
semilogx(freq/1e9,L)
grid
xlabel('Frequency (GHz)')
ylabel('Attenuation (dB)')
```



Signal Attenuation Due to Rainfall as Function of Elevation Angle

Compute the signal attenuation due to heavy rain as a function of elevation angle. Elevation angles vary from 0 to 90 degrees. Assume a path distance of 100 km and a signal frequency of 100 GHz.

Set the rain rate to 10 mm/hr.

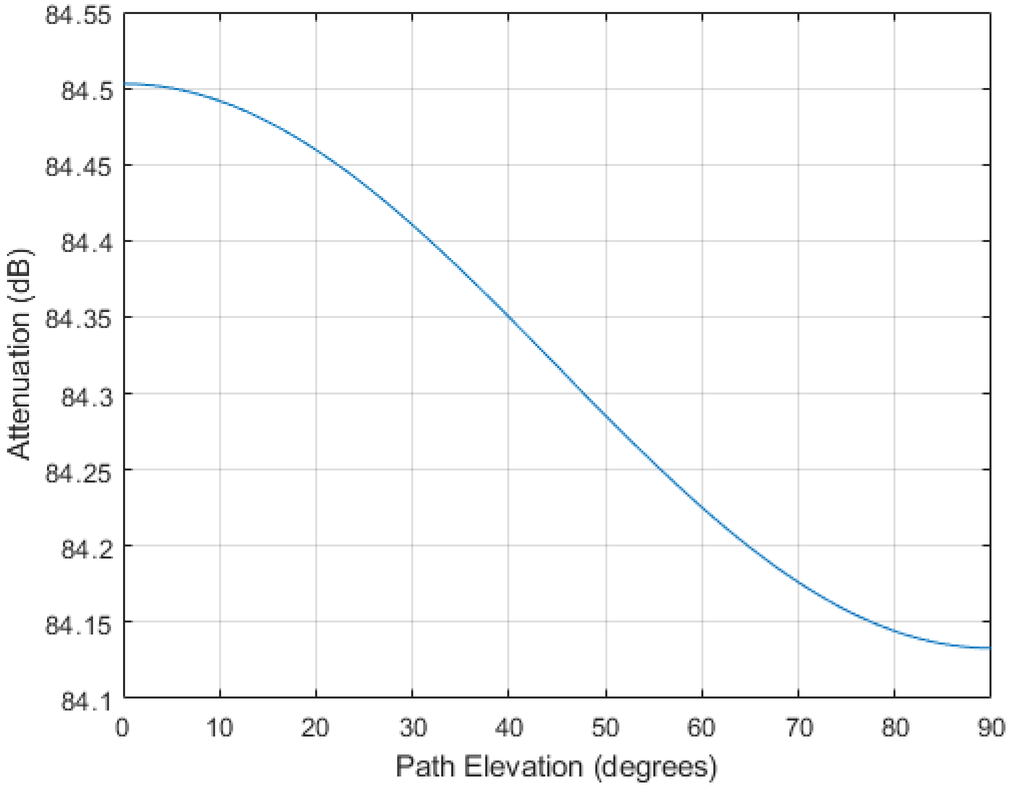
```
rr = 10.0;
```

Set the elevation angles, frequency, range.

```
elev = [0:1:90];
freq = 100.0e9;
rng = 100000.0*ones(size(elev));
```

Compute and plot the loss.

```
L = rainpl(rng,freq,rr,elev);
plot(elev,L)
grid
xlabel('Path Elevation (degrees)')
ylabel('Attenuation (dB)')
```



Signal Attenuation Due to Rainfall as Function of Polarization

Compute the signal attenuation due to heavy rainfall as a function of the polarization tilt angle. Assume a path distance of 100 km, a signal frequency of 100 GHz, and a path elevation angle of 0 degrees. Set the rainfall rate to 10 mm/hour. Plot the signal attenuation versus polarization tilt angle.

Set the polarization tilt angle to vary from -90 to 90 degrees.

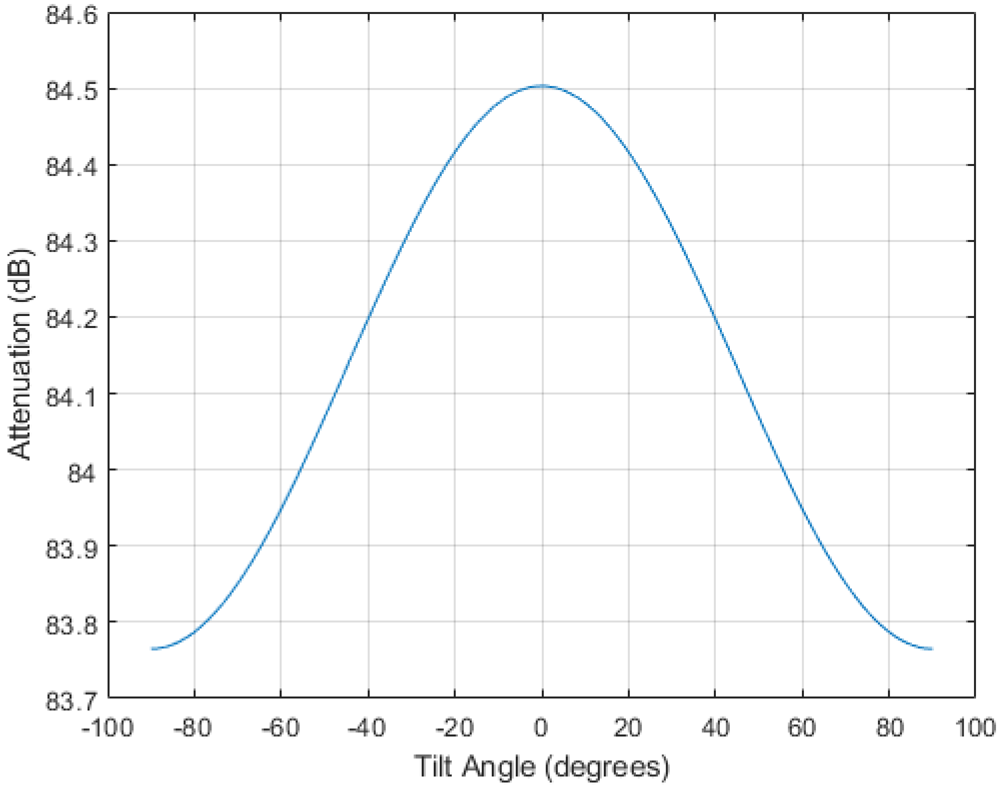
```
tau = -90:90;
```

Set the elevation angle, frequency, path distance, and rain rate.

```
elev = 0;
freq = 100.0e9;
rng = 100e3*ones(size(tau));
rr = 10.0;
```

Compute and plot the attenuation.

```
L = rainpl(rng,freq,rr,elev,tau);
plot(tau,L)
grid
xlabel('Tilt Angle (degrees)')
ylabel('Attenuation (dB)')
```



Input Arguments

range — Signal path length

nonnegative real-valued scalar | nonnegative real-valued *M*-by-1 column vector | nonnegative real-valued 1-by-*M* row vector

Signal path length, specified as a nonnegative real-valued scalar, or as a *M*-by-1 or 1-by-*M* vector. Units are in meters.

Example: `[13000.0,14000.0]`

freq — Signal frequency
positive real-valued scalar | nonnegative real-valued *N*-by-1 column vector | nonnegative real-valued 1-by-*N* row vector

Signal frequency, specified as a positive real-valued scalar, or as a nonnegative *N*-by-1 or 1-by-*N* vector. Frequencies must lie in the range 1–1000 GHz.

Example: `[1400.0e6,2.0e9]`

rainrate — Long-term statistical rain rate
nonnegative real-valued scalar

Long-term statistical rain rate, specified as a nonnegative real-valued scalar. The long-term statistical rain rate is the rain rate that is exceeded 0.01% of the time. You can adjust the percent of time using the `pct` argument. Units are in mm/hr.

Example: `1.5`

elev — Signal path elevation angle
0.0 (default) | real-valued scalar | real-valued *M*-by-1 column vector | real-valued 1-by-*M* row vector

Signal path elevation angle, specified as a real-valued scalar, or as an *M*-by-1 or 1-by- *M* vector. Units are in degrees between –90° and 90°. If `elev` is a scalar, all propagation paths have the same elevation angle. If `elev` is a vector, its length must match the dimension of `range` and each element in `elev` corresponds to a propagation range in `range`.

Example: `[0,45]`

tau — Tilt angle of polarization ellipse
0.0 (default) | real-valued scalar | real-valued *M*-by-1 column vector | real-valued 1-by-*M* row vector

Tilt angle of the signal polarization ellipse, specified as a real-valued scalar, or as an *M*-by-1 or 1-by-*M* vector. Units are in degrees between –90° and 90°. If `tau` is a scalar, all signals have the same tilt angle. If `tau` is a vector, its length must match the dimension of `range`. In that case, each element in `tau` corresponds to a propagation path in `range`.

The tilt angle is defined as the angle between the semi-major axis of the polarization ellipse and the *x*-axis. Because the ellipse is symmetrical, a tilt angle of 100° corresponds to the same polarization state as a tilt angle of -80°. Thus, the tilt angle need only be specified between ±90°.

Example: `[45,30]`

pct — Exceedance percentage of rainfall
`0.01` (default) | positive scalar between 0.001 and 1

Exceedance percentage of rainfall, specified as a positive scalar between 0.001 and 1. The long-term statistical rain rate is the rain rate that is exceeded `pct` of the time. Units are dimensionless.

Data Types: `double`

Output Arguments

L — Signal attenuation
real-valued *M*-by-*N* matrix

Signal attenuation, returned as a real-valued *M*-by-*N* matrix. Each matrix row represents a different path where *M* is the number of paths. Each column represents a different frequency where *N* is the number of frequencies. Units are in dB.

More About

Rainfall Attenuation Model

This model calculates the attenuation of signals that propagate through regions of rainfall. Rain attenuation is a dominant fading mechanism and can vary from location-to-location and from year-to-year.

Electromagnetic signals are attenuated when propagating through a region of rainfall. Rainfall attenuation is computed according to the ITU rainfall model *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. The model computes the specific attenuation (attenuation per kilometer) of a signal as a function of rainfall rate, signal frequency, polarization, and path elevation angle. The specific attenuation, γ_{R}, is modeled as a power law with respect to rain rate

$$\gamma_R = kR^{\alpha},$$

where *R* is rain rate. Units are in mm/hr. The parameter *k* and exponent *α* depend on the frequency, the polarization state, and the elevation angle of the signal path. The specific attenuation model is valid for frequencies from 1–1000 GHz.

To compute the total attenuation for narrowband signals along a path, the function multiplies the specific attenuation by the an effective propagation distance, d_{eff}. Then, the total attenuation is $L = d_{\mathrm{eff}}\gamma_{\mathrm{R}}$.

The effective distance is the geometric distance, *d*, multiplied by a scale factor

$$r = \frac{1}{0.477d^{0.633}R_{0.01}^{0.073\alpha}f^{0.123} - 10.579(1 - \exp(-0.024d))}$$

where *f* is the frequency. The article *Recommendation ITU-R P.530-17 (12/2017): Propagation data and prediction methods required for the design of terrestrial line-of-sight systems* presents a complete discussion for computing attenuation.

The rain rate, *R*, used in these computations is the long-term statistical rain rate, $R_{0.01}$. This is the rain rate that is exceeded 0.01% of the time. The calculation of the statistical rain rate is discussed in *Recommendation ITU-R P.837-7 (06/2017): Characteristics of precipitation for propagation modelling*. This article also explains how to compute the attenuation for other percentages from the 0.01% value.

You can apply the attenuation model to wideband signals. First, divide the wideband signal into frequency subbands and apply attenuation to each subband. Then, sum all attenuated subband signals into the total attenuated signal.

References

[1] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.838-3: Specific attenuation model for rain for use in prediction methods*. 2005.

[2] Radiocommunication Sector of International Telecommunication Union. *Recommendation ITU-R P.530-17: Propagation data and prediction methods required for the design of terrestrial line-of-sight systems*. 2017.

[3] *Recommendation ITU-R P.837-7: Characteristics of precipitation for propagation modelling*

[4] Seybold, J. *Introduction to RF Propagation*. New York: Wiley & Sons, 2005.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`cranerainpl` | `fogpl` | `fspl` | `gaspl`

Introduced in R2017b

rangeangle

Range and angle calculation

Syntax

```
[rng,ang] = rangeangle(pos)
[rng,ang] = rangeangle(pos,refpos)
[rng,ang] = rangeangle(pos,refpos,refaxes)
[rng,ang] = rangeangle( ___ ,model)
```

Description

The function `rangeangle` determines the propagation path length and path direction of a signal from a source point or set of source points to a reference point. The function supports two propagation models – the *free space* model and the *two-ray* model. The *free space* model is a single line-of-sight path from a source point to a reference point. The *two-ray* multipath model generates two paths. The first path follows the free-space path. The second path is a reflected path off a boundary plane at $z = 0$. Path directions are defined with respect to either the global coordinate system at the reference point or a local coordinate system at the reference point. Distances and angles at the reference point do not depend upon which direction the signal is travelling along the path.

`[rng,ang] = rangeangle(pos)` returns the propagation path length, `rng`, and direction angles, `ang`, of a signal path from a source point or set of source points, `pos`, to the origin of the global coordinate system. The direction angles are the azimuth and elevation with respect to the global coordinate axes at the origin. Signals follow a line-of-sight path from the source point to the origin. The line-of-sight path corresponds to the geometric straight line between the points.

`[rng,ang] = rangeangle(pos,refpos)` also specifies a reference point or set of reference points, `refpos`. `rng` now contains the propagation path length from the source points to the reference points. The direction angles are the azimuth and elevation with respect to the global coordinate axes at the reference points. You can specify multiple points and multiple reference points.

`[rng,ang] = rangeangle(pos,refpos,refaxes)` also specifies local coordinate system axes, `refaxes`, at the reference points. Direction angles are the azimuth and elevation with respect to the local coordinate axes centered at `refpos`.

`[rng,ang] = rangeangle( ___ ,model)`, also specifies a propagation model. When `model` is set to `'freespace'`, the signal propagates along a line-of-sight path from source point to reception point. When `model` is set to `'two-ray'`, the signal propagates along two paths from source point to reception point. The first path is the line-of-sight path. The second path is the reflecting path. In this case, the function returns the distances and angles for two paths for each source point and corresponding reference point.

Input Arguments

pos

Source point position, specified as a real-valued 3-by-1 vector or a real-valued 3-by-*N* matrix. A matrix represents multiple source points. The columns contain the Cartesian coordinates of *N* points in the form `[x;y;z]`.

When `pos` is a 3-by-*N* matrix, you must specify `refpos` as a 3-by-*N* matrix for *N* reference positions. If all the reference points are identical, you can specify `refpos` by a single 3-by-1 vector.

Position units are meters.

`refpos`

Reference point position, specified as a real-valued 3-by-1 vector or a real-valued 3-by-*N* matrix. A matrix represents multiple reference points. The columns contain the Cartesian coordinates of *N* points ins the form `[x;y;z]`.

When `refpos` is a 3-by-*N* matrix, you must specify `pos` as a 3-by-*N* matrix for *N* source positions. If all the source points are identical, you can specify `pos` by a single 3-by-1 vector.

Position units are meters.

Default: `[0;0;0]`

`refaxes`

Local coordinate system axes, specified as a real-valued 3-by-3 matrix or a 3-by-3-by-*N* array. For an array, each page corresponds to a local coordinate axes at each reference point. The columns in `refaxes` specify the direction of the coordinate axes for the local coordinate system in Cartesian coordinates. *N* must match the number of columns in `pos` or `refpos` when these dimensions are greater than one.

Default: `[1 0 0;0 1 0;0 0 1]`

`model`

Propagation model, specified as `'freespace'` or `'two-ray'`. Choosing `'freespace'` invokes the free space propagation model. Choosing `'two-ray'` invokes the two-ray propagation model.

Default: `'freespace'`

Output Arguments

`rng`

Propagation range, returned as a real-valued 1-by-*N* vector or real-valued 1-by-*2N* vector.

When `model` is set to `'freespace'`, the size of `rng` is 1-by-*N*. The propagation range is the length of the direct path from the position defined in `pos` to the corresponding reference position defined in `refpos`.

When `model` is set to `'two-ray'`, `rng` contains the ranges for the direct path and the reflected path. Alternate columns of `rng` refer to the line-of-sight path and reflected path, respectively for the same source-reference point pair. Position units are meters.

`ang`

Azimuth and elevation angles, returned as a 2-by-*N* matrix or 2-by-*2N* matrix. Each column represents a direction angle in the form `[azimuth;elevation]`.

When `model` is set to `'freespace'`, `ang` is a 2-by-*N* matrix and represents the angle of the path from a source point to a reference point.

When `model` is set to `'two-ray'`, `ang` is a 2-by-*2N* matrix. Alternate columns of `ang` refer to the line-of-sight path and reflected path, respectively.

Angle units are in degrees.

Examples

Range and Angle Computation

Compute the range and angle of a target located at *(1000,2000,50)* meters from the origin.

```
TargetLoc = [1000;2000;50];
[tgtrng,tgtang] = rangeangle(TargetLoc)
```

```
tgtrng = 2.2366e+03
```

```
tgtang = 2×1

   63.4349
    1.2810
```

Range and Angle With Respect to Local Origin

Compute the range and angle of a target located at *(1000,2000,50)* meters with respect to a local origin at *(100,100,10)* meters.

```
TargetLoc = [1000;2000;50];
Origin = [100;100;10];
[tgtrng,tgtang] = rangeangle(TargetLoc,Origin)
```

```
tgtrng = 2.1028e+03
```

```
tgtang = 2×1

   64.6538
    1.0900
```

Range and Angle With Respect to Local Coordinates

Compute the range and angle of a target located at *(1000,2000,50)* meters but with respect to a local coordinate system origin at *(100,100,10)* meters. Choose a local coordinate reference frame that is rotated about the z-axis by 45° from the global coordinate axes.

```
targetpos = [1000;2000;50];
origin = [100;100;10];
```

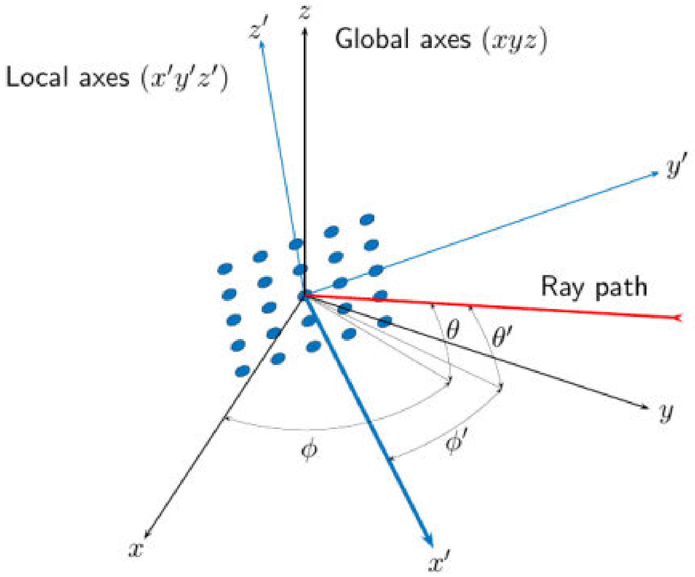
```
refaxes = [1/sqrt(2) -1/sqrt(2) 0; 1/sqrt(2) 1/sqrt(2) 0; 0 0 1];
[tgtrng,tgtang] = rangeangle(targetpos,origin,refaxes)
```

```
tgtrng = 2.1028e+03

tgtang = 2×1

   19.6538
    1.0900
```

More About

Angles in Local and Global Coordinate Systems

The `rangeangle` function returns the path distance and path angles in either the global or local coordinate systems. By default, the `rangeangle` function determines the angle a signal path makes with respect to global coordinates. If you add the `refaxes` argument, you can compute the angles with respect to local coordinates. As an illustration, this figure shows a 5-by-5 uniform rectangular array (URA) rotated from the global coordinates *(xyz)* using `refaxes`. The *x*' axis of the local coordinate system *(x'y'z')* is aligned with the main axis of the array and moves as the array moves. The path length is independent of orientation. The global coordinate system defines the azimuth and elevations angles *(Φ,θ)* and the local coordinate system defines the azimuth and elevations angles *(Φ',θ')*.



Local and Global Coordinate Axes

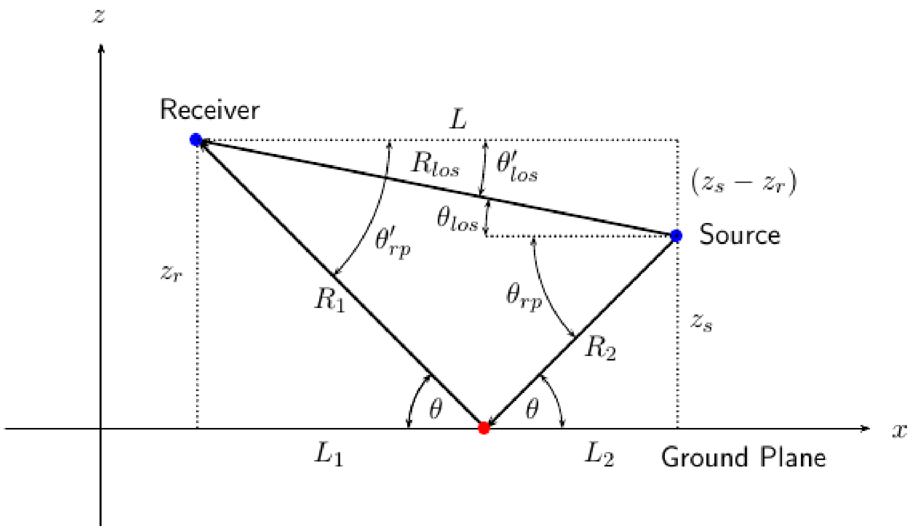
Free Space Propagation Model

The free-space signal propagation model states that a signal propagating from one point to another in a homogeneous, isotropic medium travels in a straight line, called the *line-of-sight* or *direct path*. The straight line is defined by the geometric vector from the radiation source to the destination.

Two-Ray Propagation Model

A two-ray propagation channel is the next step up in complexity from a free-space channel and is the simplest case of a multipath propagation environment. The free-space channel models a straight-line *line-of-sight* path from point 1 to point 2. In a two-ray channel, the medium is specified as a homogeneous, isotropic medium with a reflecting planar boundary. The boundary is always set at *z = 0*. There are at most two rays propagating from point 1 to point 2. The first ray path propagates along the same line-of-sight path as in the free-space channel. The line-of-sight path is often called the *direct path*. The second ray reflects off the boundary before propagating to point 2. According to the Law of Reflection , the angle of reflection equals the angle of incidence. In short-range simulations such as cellular communications systems and automotive radars, you can assume that the reflecting surface, the ground or ocean surface, is flat.

The figure illustrates two propagation paths. From the source position, s_s, and the receiver position, s_r, you can compute the arrival angles of both paths, θ'_{los} and θ'_{rp}. The arrival angles are the elevation and azimuth angles of the arriving radiation with respect to a local coordinate system. In this case, the local coordinate system coincides with the global coordinate system. You can also compute the transmitting angles, θ_{los} and θ_{rp}. In the global coordinates, the angle of reflection at the boundary is the same as the angles θ_{rp} and θ'_{rp}. The reflection angle is important to know when you use angle-dependent reflection-loss data. You can determine the reflection angle by using the `rangeangle` function and setting the reference axes to the global coordinate system. The total path length for the line-of-sight path is shown in the figure by R_{los} which is equal to the geometric distance between source and receiver. The total path length for the reflected path is $R_{rp}= R_1 + R_2$. The quantity L is the ground range between source and receiver.



You can easily derive exact formulas for path lengths and angles in terms of the ground range and object heights in the global coordinate system.

$$\vec{R} = \vec{x}_s - \vec{x}_r$$

$$R_{los} = \left|\vec{R}\right| = \sqrt{(z_r - z_s)^2 + L^2}$$

$$R_1 = \frac{z_r}{z_r + z_z}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_2 = \frac{z_s}{z_s + z_r}\sqrt{(z_r + z_s)^2 + L^2}$$

$$R_{rp} = R_1 + R_2 = \sqrt{(z_r + z_s)^2 + L^2}$$

$$\tan\theta_{los} = \frac{(z_s - z_r)}{L}$$

$$\tan\theta_{rp} = -\frac{(z_s + z_r)}{L}$$

$$\theta'_{los} = -\theta_{los}$$

$$\theta'_{rp} = \theta_{rp}$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

Introduced in R2019b

sph2cartvec

Convert vector from spherical basis components to Cartesian components

Syntax

```
vr = sph2cartvec(vs,az,el)
```

Description

`vr = sph2cartvec(vs,az,el)` converts the components of a vector or set of vectors, `vs`, from their spherical basis representation to their representation in a local Cartesian coordinate system. A spherical basis representation is the set of components of a vector projected into the right-handed spherical basis given by $(\hat{\mathbf{e}}_{az}, \hat{\mathbf{e}}_{el}, \hat{\mathbf{e}}_{R})$. The orientation of a spherical basis depends upon its location on the sphere as determined by azimuth, `az`, and elevation, `el`.

Examples

Cartesian Representation of Azimuthal Vector

Start with a vector in a spherical basis located at 45° azimuth, 45° elevation. The vector points along the azimuth direction. Compute the vector components with respect to Cartesian coordinates.

```
vs = [1;0;0];
vr = sph2cartvec(vs,45,45)
```

```
vr = 3×1

   -0.7071
    0.7071
         0
```

Input Arguments

vs — Vector in spherical basis representation

3-by-1 column vector | 3-by-*N* matrix

Vector in spherical basis representation specified as a 3-by-1 column vector or 3-by-*N* matrix. Each column of `vs` contains the three components of a vector in the right-handed spherical basis $(\hat{\mathbf{e}}_{az}, \hat{\mathbf{e}}_{el}, \hat{\mathbf{e}}_{R})$.

Example: `[4.0; -3.5; 6.3]`

Data Types: `double`
Complex Number Support: Yes

az — Azimuth angle

scalar in range [–180,180]

Azimuth angle specified as a scalar in the closed range [–180,180]. Angle units are in degrees. To define the azimuth angle of a point on a sphere, construct a vector from the origin to the point. The azimuth angle is the angle in the *xy*-plane from the positive *x*-axis to the vector's orthogonal projection into the *xy*-plane. As examples, zero azimuth angle and zero elevation angle specify a point on the *x*-axis while an azimuth angle of 90° and an elevation angle of zero specify a point on the *y*-axis.

Example: `45`

Data Types: `double`

`el` — Elevation angle
scalar in range [–90,90]

Elevation angle specified as a scalar in the closed range [–90,90]. Angle units are in degrees. To define the elevation of a point on the sphere, construct a vector from the origin to the point. The elevation angle is the angle from its orthogonal projection into the *xy*-plane to the vector itself. As examples, zero elevation angle defines the equator of the sphere and ±90° elevation define the north and south poles, respectively.

Example: `30`

Data Types: `double`

Output Arguments

`vr` — Vector in Cartesian representation
3-by-1 column vector | 3-by-*N* matrix

Cartesian vector returned as a 3-by-1 column vector or 3-by-*N* matrix having the same dimensions as `vs`. Each column of `vr` contains the three components of the vector in the right-handed *x*,*y*,*z* basis.

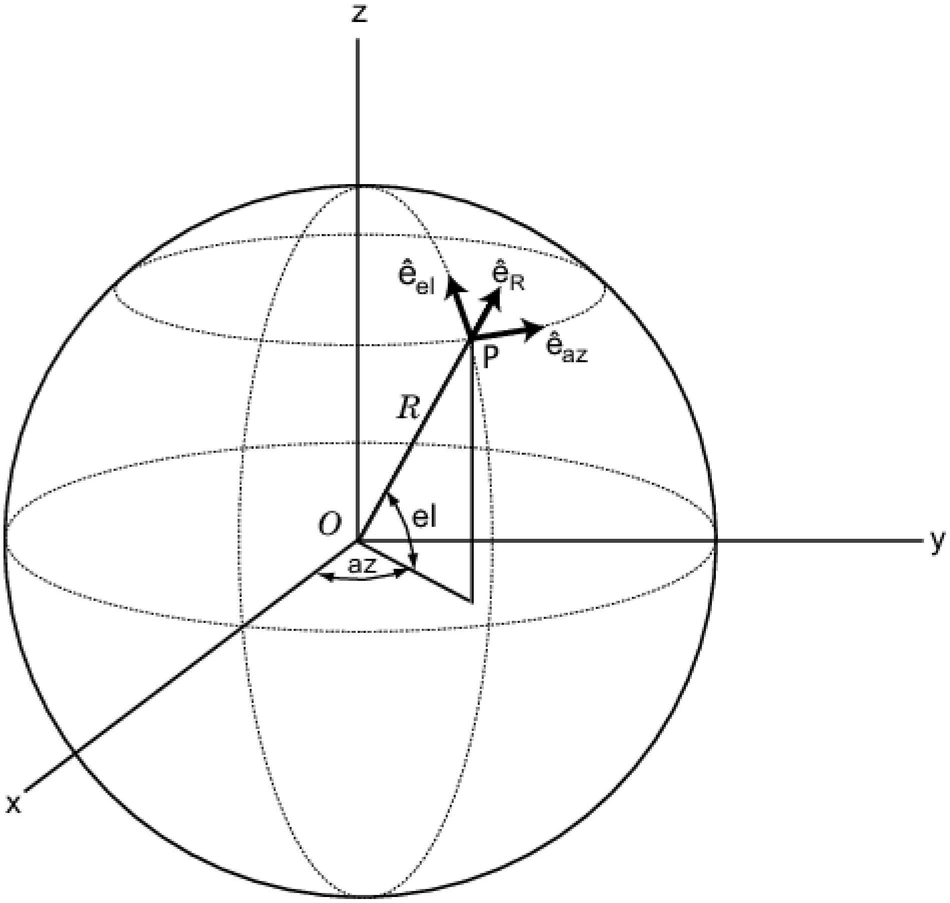
More About

Spherical basis representation of vectors

Spherical basis vectors are a local set of basis vectors which point along the radial and angular directions at any point in space.

The spherical basis is a set of three mutually orthogonal unit vectors ($\hat{\mathbf{e}}_{az}, \hat{\mathbf{e}}_{el}, \hat{\mathbf{e}}_{R}$) defined at a point on the sphere. The first unit vector points along lines of azimuth at constant radius and elevation. The second points along the lines of elevation at constant azimuth and radius. Both are tangent to the surface of the sphere. The third unit vector points radially outward.

The orientation of the basis changes from point to point on the sphere but is independent of *R* so as you move out along the radius, the basis orientation stays the same. The following figure illustrates the orientation of the spherical basis vectors as a function of azimuth and elevation:



For any point on the sphere specified by *az* and *el*, the basis vectors are given by:

$$\begin{aligned}\widehat{\mathbf{e}}_{\mathbf{az}} &= -\sin(az)\,\widehat{\mathbf{i}} + \cos(az)\,\widehat{\mathbf{j}}\\ \widehat{\mathbf{e}}_{\mathbf{el}} &= -\sin(el)\cos(az)\,\widehat{\mathbf{i}} - \sin(el)\sin(az)\,\widehat{\mathbf{j}} + \cos(el)\widehat{\mathbf{k}}\\ \widehat{\mathbf{e}}_{\mathbf{R}} &= \cos(el)\cos(az)\,\widehat{\mathbf{i}} + \cos(el)\sin(az)\,\widehat{\mathbf{j}} + \sin(el)\widehat{\mathbf{k}}\quad .\end{aligned}$$

Any vector can be written in terms of components in this basis as $\mathbf{v} = v_{az}\widehat{\mathbf{e}}_{\mathbf{az}} + v_{el}\widehat{\mathbf{e}}_{\mathbf{el}} + v_R\widehat{\mathbf{e}}_{\mathbf{R}}$. The transformations between spherical basis components and Cartesian components take the form

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} -\sin(az) & -\sin(el)\cos(az) & \cos(el)\cos(az) \\ \cos(az) & -\sin(el)\sin(az) & \cos(el)\sin(az) \\ 0 & \cos(el) & \sin(el) \end{bmatrix} \begin{bmatrix} v_{az} \\ v_{el} \\ v_R \end{bmatrix}$$

.

and

$$\begin{bmatrix} v_{az} \\ v_{el} \\ v_R \end{bmatrix} = \begin{bmatrix} -\sin(az) & \cos(az) & 0 \\ -\sin(el)\cos(az) & -\sin(el)\sin(az) & \cos(el) \\ \cos(el)\cos(az) & \cos(el)\sin(az) & \sin(el) \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}.$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Does not support variable-size inputs.

See Also

`cart2sphvec`

Introduced in R2020a

3

System Objects

comm.ACPR

Package: comm

Adjacent Channel Power Ratio measurements

Description

The ACPR System object measures adjacent channel power ratio (ACPR) of an input signal.

To measure adjacent channel power:

1. Define and set up your adjacent channel power object. See “Construction” on page 3-2.
2. Call step to measure the adjacent channel power ratio according to the properties of comm.ACPR. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.ACPR` creates a System object, H, that measures adjacent channel power ratio (ACPR) of an input signal.

`H = comm.ACPR(Name,Value)` creates object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

NormalizedFrequency

Assume normalized frequency values

Specify whether the frequency values are normalized. If you set this property to true, the object assumes that frequency values are normalized (in the [-1 1] range). The default is false. If you set this property to false, the object assumes that frequency values are measured in Hertz.

SampleRate

Sample rate of input signal

Specify the sample rate of the input signal, in samples per second, as a double-precision, positive scalar. The default is 1e6 samples per second. This property applies when you set the NormalizedFrequency property to false.

MainChannelFrequency

Main channel center frequency

Specify the main channel center frequency as a double-precision scalar. The default is `0` Hz.

When you set the NormalizedFrequency property to `true`, you must specify the center frequency as a normalized value between `-1` and `1`.

When you set the NormalizedFrequency property to `false`, you must specify the center frequency in Hertz. The object measures the main channel power in the bandwidth that you specify in the MainMeasurementBandwidth property. This measurement is taken at the center of the frequency that you specify in the `MainMeasurementBandwidth` property.

MainMeasurementBandwidth

Main channel measurement bandwidth

Specify the main channel measurement bandwidth as a double-precision, positive scalar. The default is `50e3` Hz.

When you set the NormalizedFrequency property to `true`, you must specify the measurement bandwidth as a normalized value between `0` and `1`.

When you set the NormalizedFrequency property to `false`, you must specify the measurement bandwidth in Hertz. The object measures the main channel power in the bandwidth that you specify in the `MainMeasurementBandwidth` property. This measurement is taken at the center of the frequency that you specify in the MainChannelFrequency property.

AdjacentChannelOffset

Adjacent channel frequency offsets

Specify the adjacent channel offsets as a double-precision scalar or as a row vector comprising frequencies that define the location of adjacent channels of interest. The default is `[-100e3 100e3]` Hz.

When you set the NormalizedFrequency property to `true`, you must specify normalized frequency offset values between `-1` and `1`. When you set the `NormalizedFrequency` property to `false`, you must specify frequency offset values in Hertz. The offset values indicate the distance between the main channel center frequency and adjacent channel center frequencies. Positive offsets indicate adjacent channels to the right of the main channel center frequency. Negative offsets indicate adjacent channels to the left of the main channel center frequency.

AdjacentMeasurementBandwidth

Adjacent channel measurement bandwidths

Specify the measurement bandwidth for each adjacent channel. The default is the scalar, `50e3`. The object assumes that each adjacent bandwidth is centered at the frequency defined by the corresponding frequency offset. You define this offset in the AdjacentChannelOffset property. Set this property to a double-precision scalar or row vector of length equal to the number of specified offsets in the AdjacentChannelOffset property.

When you set this property to a scalar, the object obtains all adjacent channel power measurements within equal measurement bandwidths. When you set the NormalizedFrequency property to `true`, you must specify normalized bandwidth values between `0` and `1`. When you set the NormalizedFrequency property to `false`, you must specify the adjacent channel bandwidth values in Hertz.

MeasurementFilterSource

Source of the measurement filter

Specify the measurement filter source as one of `None` | `Property`. The default is `None`. When you set this property to `None` the object does not apply filtering to obtain ACPR measurements. When you set this property to `Property`, the object applies a measurement filter to the main channel before measuring the average power. Each of the adjacent channel bands also receives a measurement filter . In this case, you specify the measurement filter coefficients in the MeasurementFilter property.

MeasurementFilter

Measurement filter coefficients

Specify the measurement filter coefficients as a double-precision row vector containing the coefficients of an FIR filter in descending order of powers of z. Center the response of the filter at DC. The ACPR object automatically shifts and applies the filter response at each of the main and adjacent channel center frequencies before obtaining the average power measurements. The internal filter states persist and clear only when you call the reset method. This property applies when you set the MeasurementFilterSource property to `Property`. The default is `1`, which is an all-pass filter that has no effect on the measurements.

SpectralEstimation

Spectral estimation control

Specify the spectral estimation control as one of `Auto` | `Specify frequency resolution` | `Specify window parameters`. The default is `Auto`.

When you set this property to `Auto`, the object obtains power measurements with a Welch spectral estimator with zero-percent overlap, a Hamming window, and a segment length equal to the length of the input data vector. In this setting, the spectral estimator set should achieve the maximum frequency resolution attainable with the input data length.

When you set this property to `Specify frequency resolution`, you specify the desired spectral frequency resolution, in normalized units or in Hertz, using the FrequencyResolution property. In this setting, the object uses the value in the `FrequencyResolution` property to automatically compute the size of the spectral estimator data window.

When you set this property to `Specify window parameters`, several spectral estimator properties become available so that you can control the Welch spectral estimation settings. These properties are: `SegmentLength`, `OverlapPercentage`, Window, and SidelobeAttenuation. Sidelobe attenuation applies only when you set the Window property to `Chebyshev`.

When you set this property to `Specify window parameters`, the `FrequencyResolution` property does not apply, and you control the resolution using the above properties.

SegmentLength

Segment length

Specify the segment length, in samples, for the spectral estimator as a numeric, positive, integer scalar. The default is `64`. The length of the segment allows you to make tradeoffs between frequency resolution and variance in the spectral estimates. A long segment length results in better resolution.

A short segment length results in more averaging and a decrease in variance. This property applies when you set the SpectralEstimation property to `Specify window parameters`.

OverlapPercentage

Overlap percentage

Specify the percentage of overlap between each segment in the spectral estimator as a double-precision scalar in the `[0 100]` interval. This property applies when you set the SpectralEstimation property to `Specify window parameters`. The default is `0` percent.

Window

Window function

Specify a window function for the spectral estimator as one of `Bartlett` | `Bartlett-Hanning` | `Blackman` | `Blackman-Harris` | `Bohman` | `Chebyshev` | `Flat Top` | `Hamming` | `Hann` | `Nuttall` | `Parzen` | `Rectangular` | `Triangular`. The default is `Hamming`. A Hamming window has `42.5`dB of sidelobe attenuation. This attenuation may mask spectral content below this value, relative to the peak spectral content. Choosing different windows allows you to make tradeoffs between resolution and sidelobe attenuation. This property applies when you set the SpectralEstimation property to `Specify window parameters`.

SidelobeAttenuation

Sidelobe attenuation for Chebyshev window

Specify the sidelobe attenuation, in decibels, for the Chebyshev window function as a double-precision, nonnegative scalar. The default is `100` dB. This property applies when you set the SpectralEstimation property to `Specify window parameters` and the Window property to `Chebyshev`.

FrequencyResolution

Frequency resolution

Specify the frequency resolution of the spectral estimator as a double-precision scalar. The default is `10625` Hz.

When you set the NormalizedFrequency property to `true`, you must specify the frequency resolution as a normalized value between `0` and `1`. When you set the NormalizedFrequency property to `false`, you must specify the frequency resolution in Hertz. The object uses the value in the FrequencyResolution property to calculate the size of the data window used by the spectral estimator. This property applies when you set the SpectralEstimation property to `Specify frequency resolution`.

FFTLength

FFT length

Specify the FFT length that the Welch spectral estimator uses as one of `Next power of 2` | `Same as segment length` | `Custom`. The default is `Next power of 2`.

When you set this property to `Custom`, the CustomFFTLength property becomes available to specify the desired FFT length.

When you set this property to `Next power of 2`, the object sets the length of the FFT to the next power of 2. This length is greater than the spectral estimator segment length or `256`, whichever is greater.

When you set this property to `Same as segment length`, the object sets the length of the FFT. This length equals the spectral estimator segment length or `256`, whichever is greater.

`CustomFFTLength`

Custom FFT length

Specify the number of FFT points that the spectral estimator uses as a numeric, positive, integer scalar. This property applies when you set the FFTLength property to `Custom`. The default is `256`.

`MaxHold`

Max-hold setting control

Specify the maximum hold setting. The default is `false`.

When you set this property to `true`, the object compares two vectors. One vector compared is the current estimated power spectral density vector (obtained with the current input data frame). The object checks this vector against the previous maximum-hold accumulated power spectral density vector, (obtained at the previous call to the `step` method). The object stores the maximum values at each frequency bin and uses them to compute average power measurements. You clear the maximum-hold spectrum by calling the `reset` method on the object. When you set this property to `false`, the object obtains power measurements using instantaneous power spectral density estimates. This property is tunable.

`PowerUnits`

Power units

Specify power measurement units as one of `dBm` | `dBW` | `Watts`. The default is `dBm`.

When you set this property to `dBm`, or `dBW`, the `step` method outputs ACPR measurements in a dBc scale (adjacent channel power referenced to main channels power). If you set this property to `Watts`, the `step` method outputs ACPR measurements in a linear scale.

`MainChannelPowerOutputPort`

Enable main channel power measurement output

When you set this property to `true`, the `step` method outputs the main channel power measurement. The default is `false`. The main channel power is the power of the input signal measured in the band that you define with the MainChannelFrequency and MainMeasurementBandwidth properties. The `step` method returns power measurements in the units that you specify in the PowerUnits property.

`AdjacentChannelPowerOutputPort`

Enable adjacent channel power measurements output

When you set this property to `true`, the `step` method outputs a vector of adjacent channel power measurements. The default is `false`. The adjacent channel powers correspond to the input signal's power measured in the bands that you define with the AdjacentChannelOffset and

AdjacentMeasurementBandwidth properties. The `step` method returns power measurements in the units that you specify in the PowerUnits property.

Methods

reset	Reset states of ACPR measurement object
step	Adjacent Channel Power Ratio measurements

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Measure ACPR of a 16-QAM signal with symbol rate of 3.84 Msps

Generate data with an alphabet size of 16 and modulate the data

```
x  = randi([0 15],5000,1);
y = qammod(x,16);
```

Upsample the data by L = 8 using a rectangular pulse shape

```
L = 8;
yPulse = rectpulse(y,L);
```

Create an ACPR measurement object and measure the modulated signal

```
acpr = comm.ACPR(...
    'SampleRate', 3.84e6*8,...
    'MainChannelFrequency', 0,...
    'MainMeasurementBandwidth', 3.84e6,...
    'AdjacentChannelOffset', [-5e6 5e6],...
    'AdjacentMeasurementBandwidth', 3.84e6,...
    'MainChannelPowerOutputPort', true,...
    'AdjacentChannelPowerOutputPort', true);
[ACPR,mainChnlPwr,adjChnlPwr] = acpr(yPulse)
```

```
ACPR = 1×2

  -14.3659  -14.3681


mainChnlPwr = 38.8668

adjChnlPwr = 1×2

   24.5010   24.4988
```

Algorithms

Note The following conditions must be true, otherwise power measurements fall out of the Nyquist interval.

$$\left| MainChannelFreq \pm \frac{MainChannelMeasBW}{2} \right| < F\text{max}$$

$$\left| (MainChannelFreq + AdjChannelOffset) \pm \frac{AdjChannelMeasBW}{2} \right| < F\text{max}$$

F_{max} = Fs/2 if NormalizedFrequency = false

F_{max} = 1 if NormalizedFrequency = true

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CCDF` | `comm.EVM` | `comm.MER`

Introduced in R2012a

reset

System object: `comm.ACPR`
Package: `comm`

Reset states of ACPR measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ACPR` object, `H`.

step

System object: `comm.ACPR`
Package: `comm`

Adjacent Channel Power Ratio measurements

Syntax

```
A = step(H,X)
[A,MAINPOW] = step(H,X)
[A,ADJPOW] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`A = step(H,X)` returns a vector of the adjacent channel power ratio, `A`, measured in the input data, `X`. The measurements are at the frequency bands that you specify with the `MainChannelFrequency`, `MainMeasurementBandwidth`, `AdjacentChannelOffset`, and `AdjacentMeasurementBandwidth` properties. Input `X` must be a double precision column vector. The length of the output vector, `A`, equals the number of adjacent channels that you specify in the `AdjacentChannelOffset` property.

`[A,MAINPOW] = step(H,X)` returns the measured main channel power, `MAINPOW`, when you set the `MainChannelPowerOutputPort` property to true. The step method outputs the main channel power measured within the main channel frequency band of interest that you specify with the `MainChannelFrequency` and `MainMeasurementBandwidth` properties.

`[A,ADJPOW] = step(H,X)` returns a vector of the measured adjacent channel powers, `ADJPOW`, when you set the `AdjacentChannelPowerOutputPort` property to true. The adjacent channel powers are measured at the adjacent frequency bands of interest that you specify with the `AdjacentChannelOffset` and `AdjacentMeasurementBandwidth` properties. The length of the output vector, `ADJPOW`, equals the length of the vector that you specify in the `AdjacentChannelOffset` property. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.AGC

Package: `comm`

Adaptively adjust gain for constant signal level output

Description

The `comm.AGC` System object creates an automatic gain controller (AGC) that adaptively adjusts its gain to achieve a constant signal level at the output. For more information, see “Logarithmic-Loop AGC” on page 3-26. This object is designed for streaming applications. For more information, see “Tips” on page 3-28.

To adaptively adjust gain for a constant signal level at the output:

1. Create the `comm.AGC` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
agc = comm.AGC
agc = comm.AGC(Name,Value)
```

Description

`agc = comm.AGC` creates an AGC System object that adaptively adjusts its gain to achieve a constant signal level at the output.

`agc = comm.AGC(Name,Value)` set properties using one or more name-value pairs. Enclose each name in quotes. For example, '`AdaptationStepSize`',0.05 sets the step size for gain updates to 0.05.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`AdaptationStepSize` — Step size for gain updates
`0.01` (default) | positive scalar

Step size for gain updates, specified as a positive scalar. Increasing the step size enables the AGC to respond more quickly to changes in the input signal level but increases variation in the output signal level after reaching steady-state operation. For more information, see “AGC Performance Criteria” on page 3-28, and the “Vary AGC Step Size” on page 3-18, and “Plot Effect of Step Size on AGC Performance” on page 3-21 examples.

Tunable: Yes

Data Types: `double`

`DesiredOutputPower` — Target output power level

`1` W (default) | positive scalar

Target output power level, specified as a positive scalar. The power is measured in Watts referenced to 1 ohm. For more information, see the “Adaptively Adjust Received Signal Amplitude Using AGC” on page 3-20 example.

Data Types: `double`

`AveragingLength` — Length of averaging window

`100` samples (default) | positive integer

Length of the averaging window in samples, specified as a positive integer. For more information on how the averaging length influences the variance of the AGC output signal in steady-state operation and the execution speed, see “Tips” on page 3-28 and the “Vary AGC Averaging Length” on page 3-13 example.

Data Types: `double`

`MaxPowerGain` — Maximum power gain

`60` dB (default) | positive scalar

Maximum power gain in decibels, specified as a positive scalar. Large gain adjustments can cause clipping when a small input signal power suddenly increases. Use this property to avoid large gain adjustments by limiting the gain that the AGC applies to the input signal. For more information, see the “Vary AGC Maximum Gain” on page 3-15 and “Demonstrate Effect of Maximum AGC Gain on Packet Data” on page 3-24 examples.

Data Types: `double`

Usage

Syntax

```
y = agc(x)
[y,powerlevel] = agc(x)
```

Description

`y = agc(x)` adaptively adjusts the gain to the input signal to achieve a reference signal level at the output. The AGC System object uses a square law detector to determine the output signal level. For more information, see “AGC Detector” on page 3-27.

`[y,powerlevel] = agc(x)` returns `powerlevel`, the power level estimate of the input signal. You can use `powerlevel` as an energy detector output.

Input Arguments

x — Input signal
column vector

Input signal, specified as a column vector.

Data Types: `single` | `double`

Output Arguments

y — Output signal
column vector

Output signal, returned as a column vector. The output signal is the same data type as the input signal, `x`.

`powerlevel` — Power level estimate
N_S-element column vector

Power level estimate, returned as an N_S-element column vector. N_S is the length of the input signal, `x`. You can use `powerlevel` as an energy detector output.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Vary AGC Averaging Length

Apply different AGC averaging lengths to QAM-modulated signals. Compare the variance and plot of the signals after AGC is applied.

Create three AGC System objects with their average window lengths set to 10, 100, and 1000 samples, respectively.

```
agc1 = comm.AGC('AveragingLength',10);
agc2 = comm.AGC('AveragingLength',100);
agc3 = comm.AGC('AveragingLength',1000);
```

Generate 16-QAM modulated and raised cosine pulse shaped packetized data.

```
M = 16;
d = randi([0 M-1],1000,1);
```

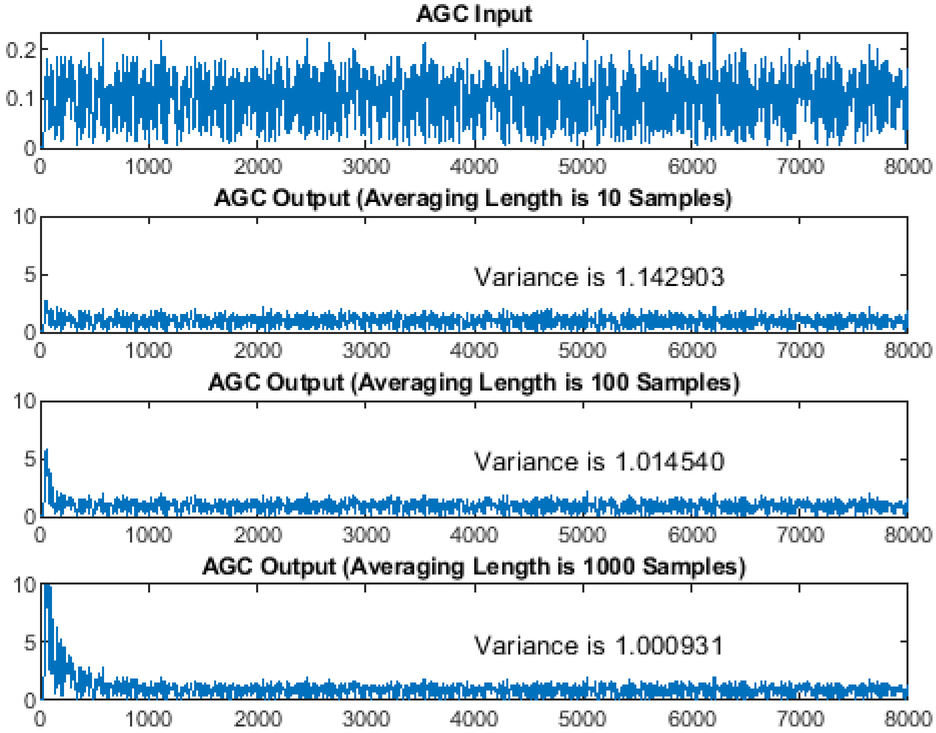
```
s = qammod(d,M);
x = 0.1*s;
pulseShaper = comm.RaisedCosineTransmitFilter;
y = awgn(pulseShaper(x),inf);
```

Apply AGC to the data capturing separate outputs for each AGC object.

```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot and compare the signals. As the averaging length increases, the variance at the output of AGC decreases.

```
figure(1)
subplot(4,1,1)
plot(abs(y))
title('AGC Input')
subplot(4,1,2)
plot(abs(r1))
axis([0 8000 0 10])
title('AGC Output (Averaging Length is 10 Samples)')
text(4000,5,sprintf('Variance is %f',var(r1(3000:end))))
subplot(4,1,3)
plot(abs(r2))
axis([0 8000 0 10])
title('AGC Output (Averaging Length is 100 Samples)')
text(4000,5,sprintf('Variance is %f',var(r2(3000:end))))
subplot(4,1,4)
plot(abs(r3))
axis([0 8000 0 10])
title('AGC Output (Averaging Length is 1000 Samples)')
text(4000,5,sprintf('Variance is %f',var(r3(3000:end))))
```



Vary AGC Maximum Gain

Apply different AGC maximum gain levels to QPSK-modulated signals. Compare the plot of the signals after AGC is applied.

Create three AGC System objects with their maximum gain values set to 10, 20, and 30 dB, respectively.

```
agc1 = comm.AGC('MaxPowerGain',10);
agc2 = comm.AGC('MaxPowerGain',20);
agc3 = comm.AGC('MaxPowerGain',30);
```

Generate QPSK-modulated data. Pass the data through raised cosine pulse shaped filtering and an AWGN channel.

```
M = 4;
pktLen = 10000;
d = randi([0 M-1],pktLen,1);
s = pskmod(d,M,pi/4);
x = repmat([zeros(pktLen,1); 0.3*s],3,1);
pulseShaper = comm.RaisedCosineTransmitFilter;
y = awgn(pulseShaper(x),50);
```

Apply AGC to the data capturing separate outputs for each AGC object.

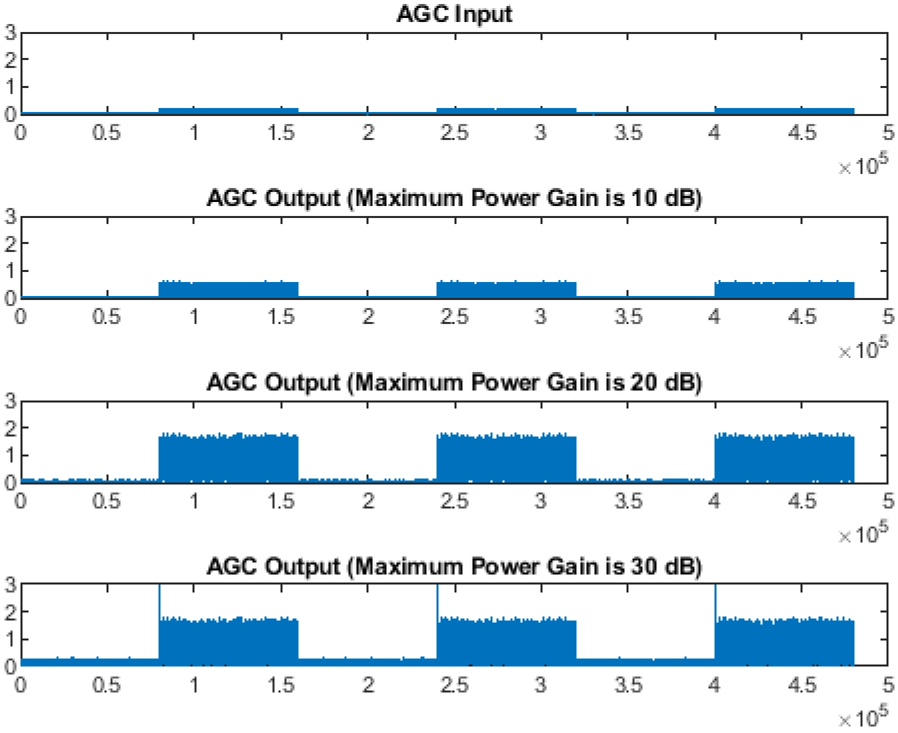
```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot the input signal and the AGC-adjusted signal with various maximum gain levels. Compare the results for the conditions in this example.

- A maximum gain setting of 10 dB is too small, and the AGC output does not reach the desired output signal level risking data loss due to decreased signal dynamic range.
- A maximum gain setting of 20 dB is optimal, and the AGC output reaches the desired level without signal loss due to saturation.
- A maximum gain setting of 30 dB is too large, and the AGC output overshoots the desired signal level risking signal saturation and data loss at the start of received packets.
- Between packets the input signal contains only noise.

As shown in the plots, the packet transmissions have extended periods when no data is received. The extended periods with no data received results in AGC increasing to the maximum gain setting. If a packet arrives when the AGC gain is too high, the output power overshoots the desired signal level until the AGC can respond to the change in the input power level and reduce its gain.

```
limits = [0 3];
figure(1)
subplot(4,1,1)
plot(abs(y))
ylim(limits)
title('AGC Input')
subplot(4,1,2)
plot(abs(r1))
ylim(limits)
title('AGC Output (Maximum Power Gain is 10 dB)')
subplot(4,1,3)
plot(abs(r2))
ylim(limits)
title('AGC Output (Maximum Power Gain is 20 dB)')
subplot(4,1,4)
plot(abs(r3))
ylim(limits)
title('AGC Output (Maximum Power Gain is 30 dB)')
```



Show Power Level Estimate Output from AGC

Plot a signal and the power level estimate. Compare the results. The power level estimate can serve as a power detector, indicating exactly when the received packet has arrived.

Create an AGC System object with its maximum gain value set to 20 dB.

```
agc20 = comm.AGC('MaxPowerGain',20);
```

Generate QPSK-modulated data. Pass the data through raised cosine pulse shaped filtering and an AWGN channel.

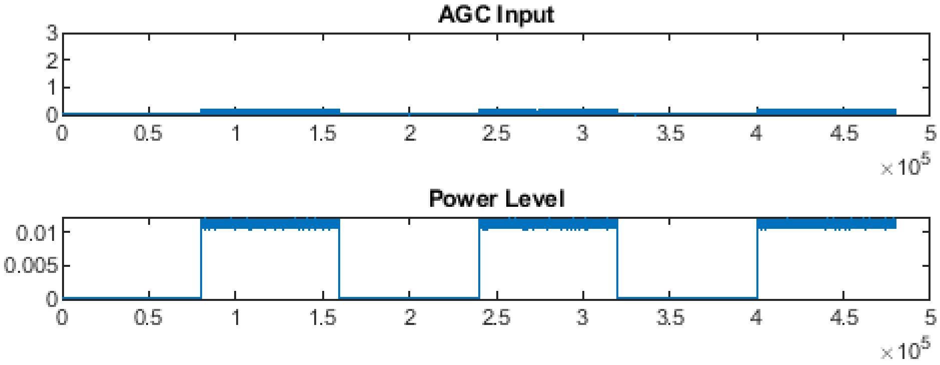
```
modOrd = 4; % Modulation order
pktLen = 10000; % Packet length
d = randi([0 modOrd-1],pktLen,1);
s = pskmod(d,modOrd,pi/4);
x = repmat([zeros(pktLen,1); 0.3*s],3,1);
pulseShaper = comm.RaisedCosineTransmitFilter;
y = awgn(pulseShaper(x),50);
```

Apply AGC to the data capturing separate outputs for each AGC object.

```
[r2,p2] = agc20(y);
```

Plot the input signal and the received signal power level estimate. Compare the results. Reception of packetized data with extended periods when no data is received results in the detected power level estimate decreasing to nearly zero. While an input signal is detected, the output power level estimate, p2, serves as a power detector, indicating exactly when the received packet has arrived.

```
limits = [0 3];
figure(1)
subplot(4,1,1)
plot(abs(y))
ylim(limits)
title('AGC Input')
subplot(4,1,2)
plot(abs(p2))
title('Power Level')
```



Vary AGC Step Size

Apply different AGC step sizes to QPSK-modulated signals. Compare the signals after applying AGC.

Create three AGC System objects with their step sizes set to 1e-1, 1e-3, and 1e-4, respectively.

```
agc1 = comm.AGC('AdaptationStepSize',1e-1);
agc2 = comm.AGC('AdaptationStepSize',1e-3);
agc3 = comm.AGC('AdaptationStepSize',1e-4);
```

Generate QPSK-modulated data with raised cosine pulse shaping.

```
d = randi([0 3],500,1);
s = pskmod(d,4,pi/4);
x = 0.1*s;
pulseShaper = comm.RaisedCosineTransmitFilter;
y = pulseShaper(x);
```

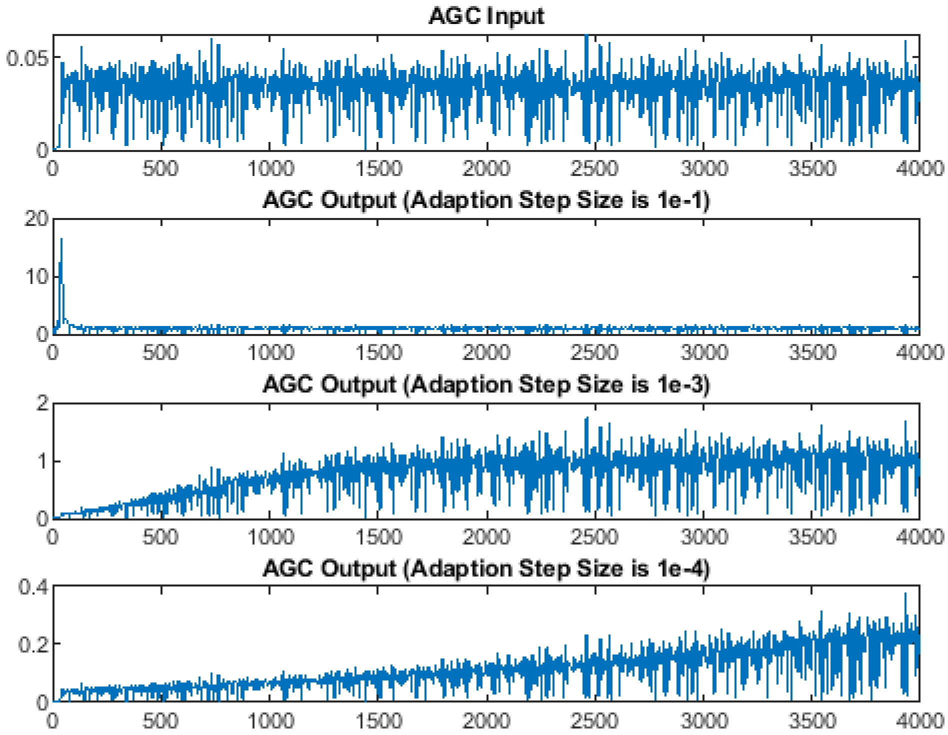
Apply AGC to the data capturing separate outputs for each AGC object.

```
r1 = agc1(y);
r2 = agc2(y);
r3 = agc3(y);
```

Plot the input and output signal after the various AGC step sizes.

- With the step size set to 1e-1, the AGC output signal overshoot is evident. The output signal converges very quickly.
- With the step size set to 1e-3, the AGC output signal overshoot disappears. The output signal gradually converges.
- With the step size set to 1e-4, the AGC output signal takes 2 to 3 times longer to converge than a step size of 1e-3.

```
figure
subplot(4,1,1)
plot(abs(y))
title('AGC Input')
subplot(4,1,2)
plot(abs(r1))
title('AGC Output (Adaption Step Size is 1e-1)')
subplot(4,1,3)
plot(abs(r2))
title('AGC Output (Adaption Step Size is 1e-3)')
subplot(4,1,4)
plot(abs(r3))
title('AGC Output (Adaption Step Size is 1e-4)')
```



Adaptively Adjust Received Signal Amplitude Using AGC

Modulate and amplify a QPSK signal. Set the received signal amplitude to approximately 1 volt by using an AGC. Plot the output.

Create a QPSK-modulated signal by using the QPSK System object.

```
data = randi([0 3],1000,1);
qpsk = comm.QPSKModulator;
modData = qpsk(data);
```

Attenuate the modulated signal.

```
txSig = 0.1*modData;
```

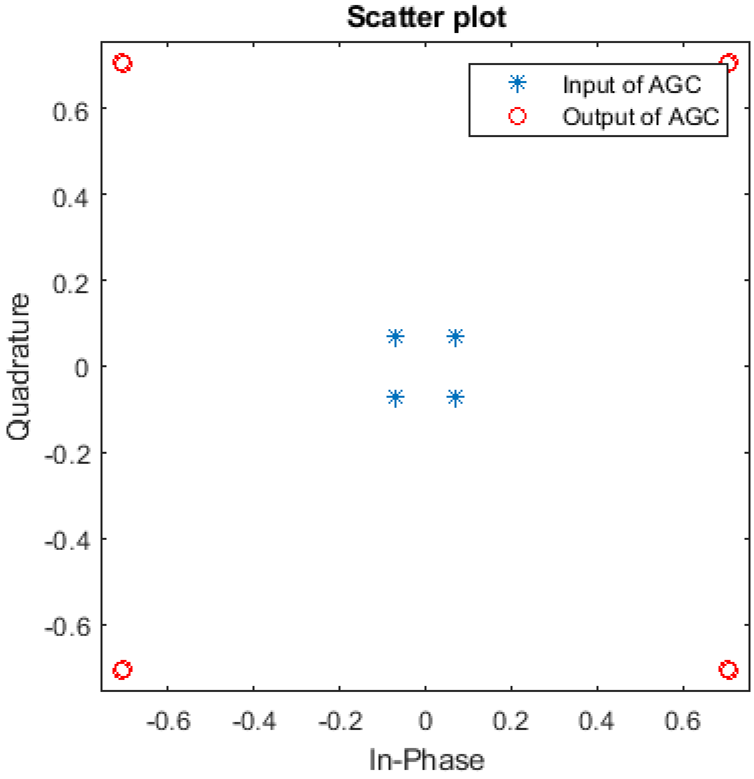
Create an AGC System object and pass the transmitted signal through it. The AGC adjusts the received signal power to approximately 1 W.

```
agc = comm.AGC;
rxSig = agc(txSig);
```

Plot the signal constellations of the transmit and received signals after the AGC reaches steady-state.

```
h = scatterplot(txSig(200:end),1,0,'*');
hold on
```

```
scatterplot(rxSig(200:end),1,0,'or',h);
legend('Input of AGC','Output of AGC')
```



Measure and compare the power of the transmitted and received signals after the AGC reaches a steady state. The power of the transmitted signal is 100 times smaller than the power of the received signal.

```
txPower = var(txSig(200:end));
rxPower = var(rxSig(200:end));
[txPower rxPower]
```

```
ans = 1×2

    0.0100    0.9970
```

Plot Effect of Step Size on AGC Performance

Create two AGC System objects™ to adjust the level of the received signal using two different step sizes with identical update periods.

Generate an 8-PSK signal such that its power is 10 W.

```
data = randi([0 7],200,1);
modData = sqrt(10)*pskmod(data,8,pi/8,'gray');
```

Create a pair of raised cosine matched filters with their `Gain` property set so that they have unity output power.

```
txfilter = comm.RaisedCosineTransmitFilter('Gain',sqrt(8));
rxfilter = comm.RaisedCosineReceiveFilter('Gain',sqrt(1/8));
```

Filter the modulated signal through the raised cosine transmit filter.

```
txSig = txfilter(modData);
```

Create two AGC System objects to adjust the received signal level. Set a step size of 0.01 and 0.1, respectively.

```
agc1 = comm.AGC('AdaptationStepSize',0.01);
agc2 = comm.AGC('AdaptationStepSize',0.1);
```

Apply AGC to the modulated signal capturing separate outputs for each AGC object.

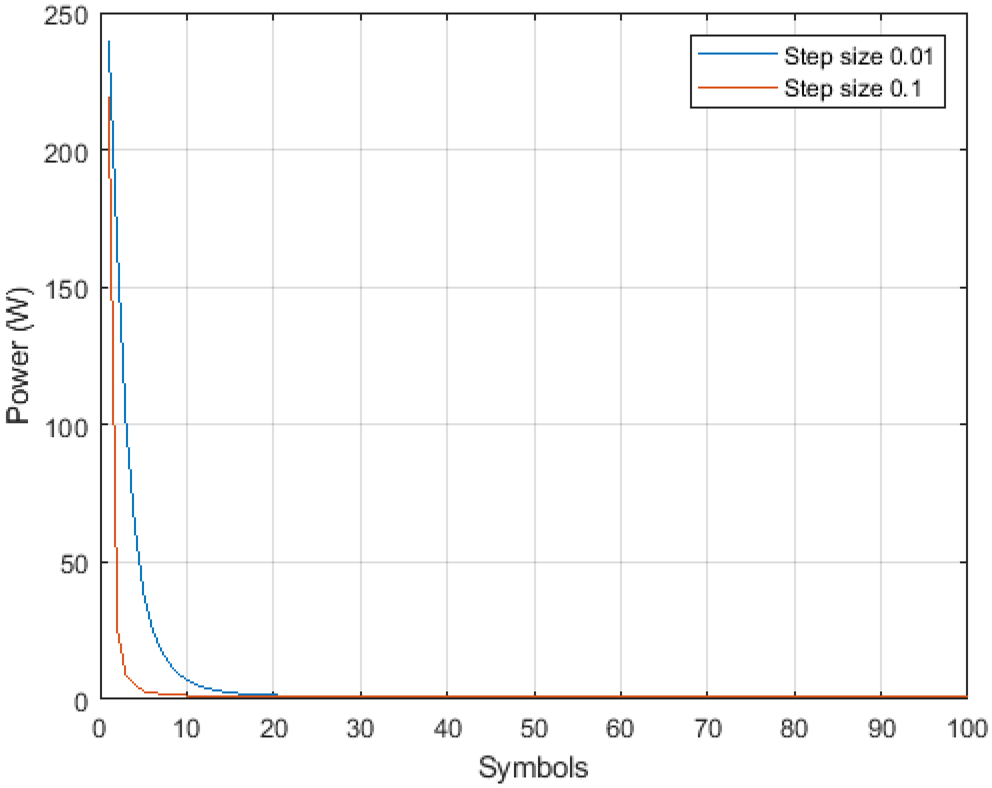
```
agcOut1 = agc1(txSig);
agcOut2 = agc2(txSig);
```

Filter the AGC output signals by using the raised cosine receive filter.

```
rxSig1 = rxfilter(agcOut1);
rxSig2 = rxfilter(agcOut2);
```

Plot the power of the filtered AGC responses while accounting for the 10 symbol delay through the transmit-receive filter pair.

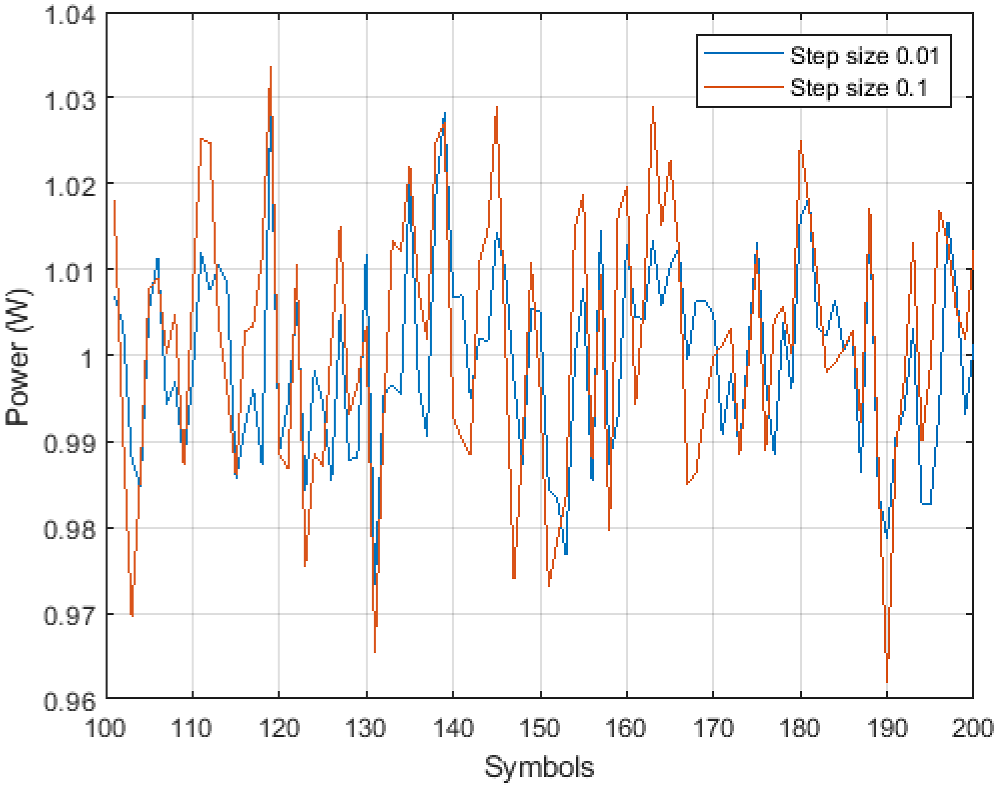
```
plot([abs(rxSig1(11:110)).^2 abs(rxSig2(11:110)).^2])
grid on
xlabel('Symbols')
ylabel('Power (W)')
legend('Step size 0.01','Step size 0.1')
```



The signal with the larger step size converges faster to the AGC target power level of 1 W.

Plot the power of the steady-state filtered AGC signals by including only the last 100 symbols. The larger AGC step size results in less accurate gain correction. Larger AGC step size values result in faster convergence at the expense of less accurate gain control.

```
plot((101:200),[abs(rxSig1(101:200)).^2 abs(rxSig2(101:200)).^2])
grid on
xlabel('Symbols')
ylabel('Power (W)')
legend('Step size 0.01','Step size 0.1')
```



Demonstrate Effect of Maximum AGC Gain on Packet Data

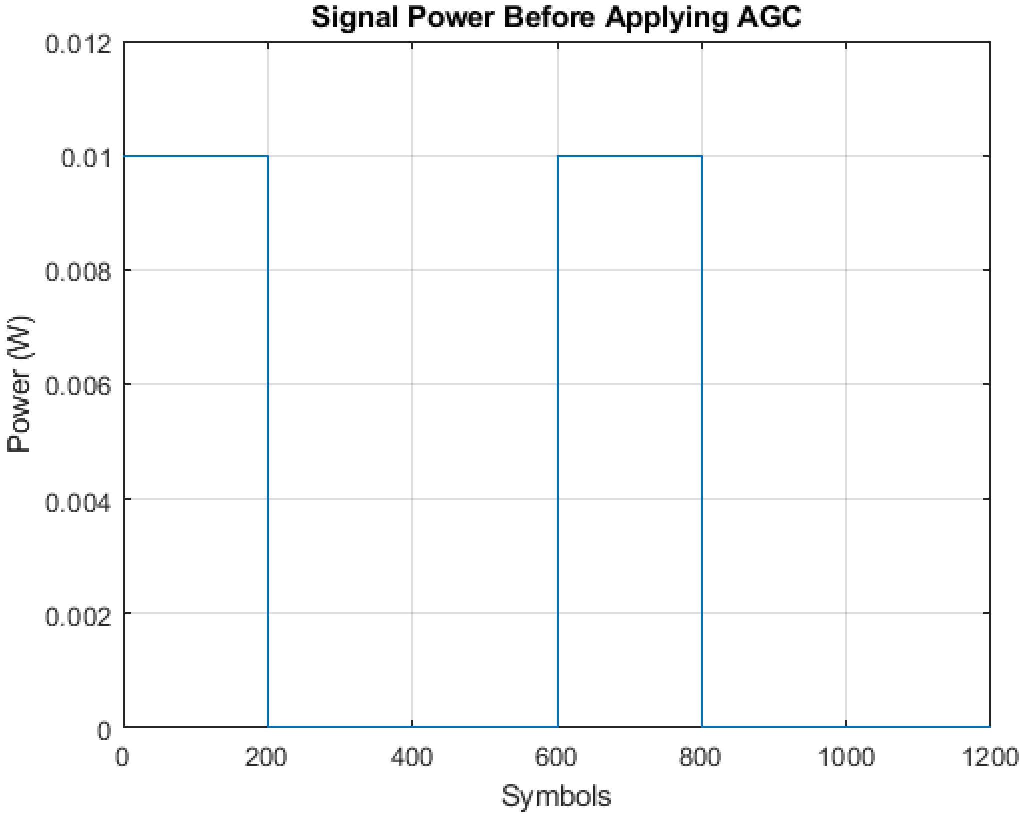
Pass attenuated QPSK packet data to two AGCs with different maximum gains. Plot the results.

Create two, 200-symbol QSPK data packets. Transmit the packets over a 1200-symbol frame.

```
modData1 = pskmod(randi([0 3],200,1),4,pi/4);
modData2 = pskmod(randi([0 3],200,1),4,pi/4);
txSig = [modData1; zeros(400,1); modData2; zeros(400,1)];
```

Attenuate the transmitted burst signal by 20 dB and plot its power.

```
rxSig = 0.1*txSig;
rxSigPwr = abs(rxSig).^2;
plot(rxSigPwr)
grid
xlabel('Symbols')
ylabel('Power (W)')
title('Signal Power Before Applying AGC')
```



Create two AGCs with maximum power gains of 30 dB and 24 dB, respectively.

```
agc1 = comm.AGC('MaxPowerGain',30,'AdaptationStepSize',0.02);
```

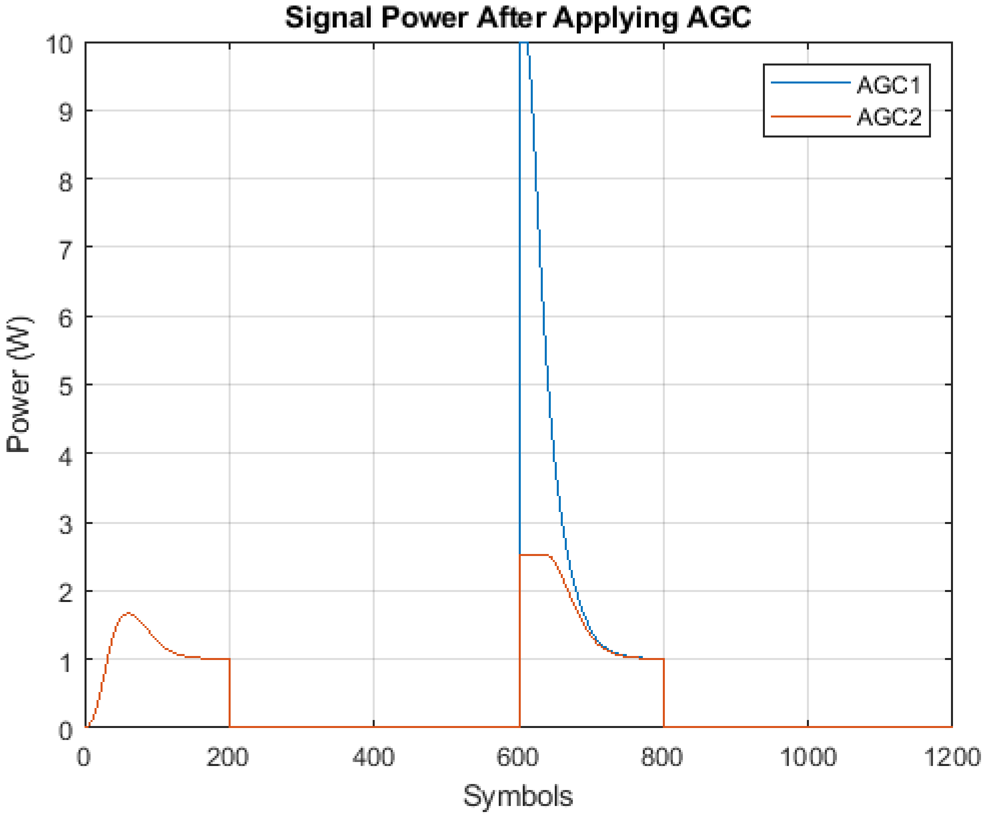
```
agc2 = comm.AGC('MaxPowerGain',24,'AdaptationStepSize',0.02);
```

Apply AGC to the attenuated signal capturing separate outputs for each AGC object. Calculate the output power for each case.

```
rxAGC1 = agc1(rxSig);
rxAGC2 = agc2(rxSig);

pwrAGC1 = abs(rxAGC1).^2;
pwrAGC2 = abs(rxAGC2).^2;
```

Plot the output powers. Initially, for the second packet, the `agc1` output signal power is too high because the AGC applied its maximum gain during the period when no data was transmitted. The corresponding `agc2` output signal power (2.5 W) overshoots the target power level of 1 W by significantly less than the `agc1` output signal power (10 W). The convergence time for `agc2` is shorter than the convergence time for `agc1`, because the signal input to `agc2` applies a smaller maximum gain than `agc1`.

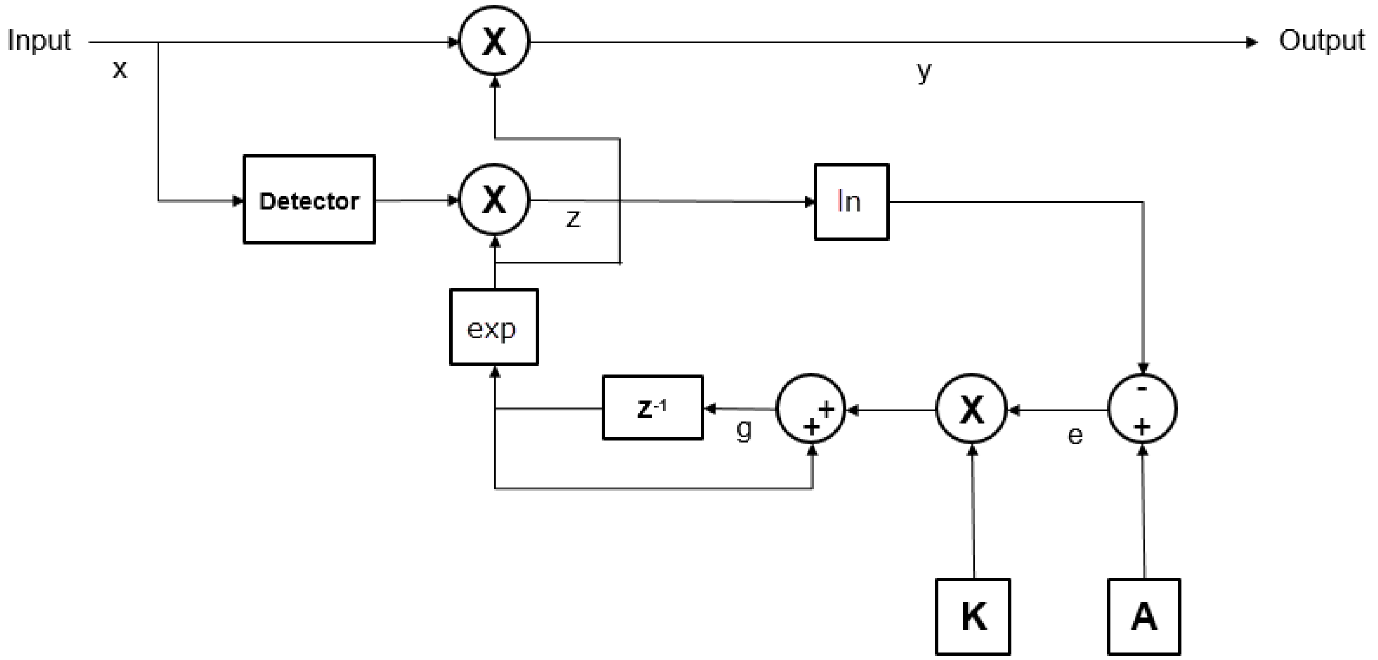
```
plot([pwrAGC1 pwrAGC2])
legend('AGC1','AGC2')
grid
xlabel('Symbols')
ylabel('Power (W)')
title('Signal Power After Applying AGC')
```



More About

Logarithmic-Loop AGC

The AGC implementation uses a logarithmic feedback loop. As this figure of the logarithmic-loop AGC algorithm shows, the output signal is the product of the input signal and the exponential of the loop gain. The error signal is the difference between the reference level and the product of the logarithm of the detector output and the exponential of the loop gain. After multiplying by the step size, the AGC passes the error signal to an integrator.



The logarithmic-loop AGC performs well for a variety of signal types, including amplitude modulation. The “AGC Detector” on page 3-27 is applied to the input signal, which improves convergence times, but increases signal power variation at the detector input. Large signal variation at the detector input is acceptable for floating-point systems.

Mathematically, the algorithm is summarized as

$$y(n) = x(n) \cdot \exp(g(n-1)),$$
$$z(n) = D(x(n)) \cdot \exp(2g(n-1)),$$
$$e(n) = A - \ln(z(n)), \text{ and}$$
$$g(n) = g(n-1) + K \cdot e(n),$$

where:

- *x* is the input signal.
- *y* is the output signal.
- *g* is the loop gain.
- *D(•)* is the detector function.
- *z* is the detector output.
- *A* is the reference value.
- *e* is the error signal.
- *K* is the step size.

AGC Detector

The AGC detector output, *z*, computes a square law detector given by

$$z(m) = \frac{1}{N}\sum_{n = mN}^{(m+1)N-1} |y(n)|^2 ,$$

where *N* is the update period. The square law detector produces an output proportional to the square of the input signal *y*.

AGC Performance Criteria

Increasing the step size decreases the attack time and decay times, but it also increases gain pumping.

- Attack time — The duration taken for the AGC to respond to an increase in the input amplitude
- Decay time — The duration taken for the AGC to respond to a decrease in the input amplitude
- Gain pumping — The variation in the gain value during steady-state operation

Tips

- This System object is designed for streaming applications.
- If the signal amplitude does not change within the frame, you can simulate an ideal AGC by calculating the average gain desired for a frame of samples. Then, apply the gain to each sample in the frame.
- If you use the AGC with higher order QAM signals, you might need to reduce the variation in the gain during steady-state operation. Inspect the constellation diagram at the output of the AGC during steady-state operation. You can increase the averaging length to avoid frequent gain adjustments. An increase in averaging length reduces execution speed.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Blocks

AGC

Introduced in R2013a

comm.AlgebraicDeinterleaver

Package: comm

(To be removed) Deinterleave input symbols using algebraically derived permutation vector

Compatibility

comm.AlgebraicDeinterleaver will be removed in a future release. Use algdeintrlv instead. For more information, see “Compatibility Considerations” on page 3-32.

Description

The AlgebraicDeinterleaver object restores the original ordering of a sequence that was interleaved using the AlgebraicInterleaver object. In typical usage, the properties of the two objects have the same values.

To deinterleave input symbols using an algebraically derived permutation vector:

1. Define and set up your algebraic deinterleaver object. See “Construction” on page 3-29.
2. Call step to deinterleave the input symbols according to the properties of comm.AlgebraicDeinterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.AlgebraicDeinterleaver creates a deinterleaver System object, H. This object restores the original ordering of a sequence from the corresponding algebraic interleaver object.

H = comm.AlgebraicDeinterleaver(Name,Value) creates an Algebraic deinterleaver System object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

Method

Algebraic method to generate permutation vector

Specify the algebraic method as one of Takeshita-Costello| Welch-Costas. The default is Takeshita-Costello. The algebraic interleaver performs all computations in modulo *N*, where *N* equals the length you set in the Length property.

For the Welch-Costas method, the value of (N+1) must be a prime number, where N equals the value you specify in the Length property. You must set the PrimitiveElement property to an integer, A, between 1 and N. This integer represents a primitive element of the finite field $GF(N+1)$.

For the Takeshita-Costello method, you must set the Length property to a value equal to 2^m, for any integer m. You must also set the MultiplicativeFactor property to an odd integer that is less than the value of the Length property. The CyclicShift property requires a nonnegative integer which is less than the value of the Length property. The Takeshita-Costello interleaver method uses a cycle vector of length N, which you specify in the Length property. The cycle vector calculation uses the equation, $\mathrm{mod}(k \times (n - 1) \times \frac{n}{2}, N)+1$, for any integer n, between 1 and N. The object creates an intermediate permutation function using the relationship, $P(c(n)) = c(n+1)$. You can shift the elements of the intermediate permutation vector to the left by the amount specified by the CyclicShift property. Doing so produces the interleaver's actual permutation vector.

Length

Number of elements in input vector

Specify the number of elements in the input as a positive, integer, scalar. When you set the Method property to Welch-Costas, then the value of Length+1 must equal a prime number. When you set the Method property to Takeshita-Costello, then the value of the Length property requires a power of two. The default is 256.

MultiplicativeFactor

Cycle vector computation factor

Specify the factor the object uses to compute the interleaver's cycle vector as a positive, integer, scalar. This property applies when you set the Method property to Takeshita-Costello. The default is 13.

CyclicShift

Amount of cyclic shift

Specify the amount by which the object shifts indices, when the object creates the final permutation vector, as a nonnegative, integer, scalar. The default is 0. This property applies when you set the Method property to Takeshita-Costello.

PrimitiveElement

Primitive element

Specify the primitive element as an element of order N in the finite field $GF(N+1)$. N is the value you specify in the Length on page 3-0 property. You can express every nonzero element of $GF(N+1)$ as the value of the PrimitiveElement property raised to some integer power. In a Welch-Costas interleaver, the permutation maps the integer k to mod(A^k,N+1)-1, where A represents the value of the PrimitiveElement property. This property applies when you set the Method property to Welch-Costas. The default is 6.

Methods

step (To be removed) Deinterleave input symbols using algebraically derived permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Algebraic Interleaving and Deinterleaving

Create algebraic interleaver and deinterleaver objects having a length of 16.

```
interleaver = comm.AlgebraicInterleaver('Length',16);
```

```
Warning: COMM.ALGEBRAICINTERLEAVER will be removed in a future release. Use ALGINTRLV instead. Se
```

```
deinterleaver = comm.AlgebraicDeinterleaver('Length',16);
```

```
Warning: COMM.ALGEBRAICDEINTERLEAVER will be removed in a future release. Use ALGDEINTRLV instead
```

Generate 8-ary data. Interleave and deinterleave the data.

```
data = randi([0 7],16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Compare the original, interleaved, and deinterleaved data.

```
[data,intData,deIntData]
```

```
ans = 16×3

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

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Algebraic Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.AlgebraicDeinterleaver` will be removed

Not recommended starting in R2019b

comm.AlgebraicDeinterleaver will be removed in a future release. Use `algdeintrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`algdeintrlv` | `algintrlv` | `deintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.AlgebraicDeinterleaver`
Package: `comm`

(To be removed) Deinterleave input symbols using algebraically derived permutation vector

Compatibility

step will be removed in a future release. Use `algdeintrlv` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using an algebraic interleaver. An algebraically derived permutation vector based on the algebraic method you specify in the Method property forms the base of the output, `Y`. `X` must be a column vector of length specified by the Length property. `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.AlgebraicInterleaver

Package: `comm`

(To be removed) Permute input symbols using algebraically derived permutation vector

Compatibility

comm.AlgebraicInterleaver will be removed in a future release. Use `algintrlv` instead. For more information, see "Compatibility Considerations" on page 3-37.

Description

The `AlgebraicInterleaver` object rearranges the elements of its input vector using an algebraically derived permutation.

To interleave input symbols using an algebraically derived permutation vector:

1. Define and set up your algebraic interleaver object. See "Construction" on page 3-34.
2. Call `step` to interleave the input symbols according to the properties of `comm.AlgebraicInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.AlgebraicInterleaver` creates an interleaver System object, `H`, that permutes the symbols in the input signal. This permutation is based on an algebraically derived permutation vector.

`H = comm.AlgebraicInterleaver(Name,Value)` creates an algebraic interleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Method`

Algebraic method to generate permutation vector

Algebraic method to generate permutation vector

Specify the algebraic method as one of `Takeshita-Costello`| `Welch-Costas`. The default is `Takeshita-Costello`. The algebraic interleaver performs all computations in modulo *N*, where *N* is the length you set in the Length property.

For the Welch-Costas method, the value of (N+1) must be a prime number, where N is the value you specify in the Length property. You must set the PrimitiveElement property to an integer, A, between 1 and N. This integer represents a primitive element of the finite field $GF(N+1)$.

For the Takeshita-Costello method, you must set the Length property to a value equal to 2^m, for any integer m. You must also set the MultiplicativeFactor property to an odd integer which is less than the value of the Length property. In addition, you must set the CyclicShift property to a nonnegative integer which is less than the value of the Length property. The Takeshita-Costello interleaver method uses a cycle vector of length N, which you specify in the Length property. The cycle vector calculation uses the equation, $\mathrm{mod}(k \times (n-1) \times \frac{n}{2}, N)+1$, for any integer n, between 1 and N. The object creates an intermediate permutation function using the relationship, $P(c(n)) = c(n+1)$. You can shift the elements of the intermediate permutation vector to the left by the amount specified by the CyclicShift property. Doing so produces the actual permutation vector of the interleaver.

Length

Number of elements in input vector

Specify the number of elements in the input as a positive, integer, scalar. When you set the Method property to Welch-Costas, then the value of Length+1 must equal a prime number. When you set the Method property to Takeshita-Costello, then the value of the Length property requires a power of two. The default is 256.

MultiplicativeFactor

Cycle vector computation method

Specify the factor the object uses to compute the cycle vector for the interleaver as a positive, integer, scalar. This property applies when you set the Method on page 3-0 property to Takeshita-Costello. The default is 13.

CyclicShift

Amount of cyclic shift

Specify the amount by which the object shifts indices, when it creates the final permutation vector, as a nonnegative, integer, scalar. This property applies when you set the Method on page 3-0 property to Takeshita-Costello. The default is 0.

PrimitiveElement

Primitive element

Specify the primitive element as an element of order N in the finite field $GF(N+1)$. N is the value you specify in the Length property. You can express every nonzero element of $GF(N+1)$ as the value of the PrimitiveElement property raised to an integer power. In a Welch-Costas interleaver, the permutation maps the integer k to $\mathrm{mod}(A^k,N+1)-1$, where A represents the value of the PrimitiveElement property. This property applies when you set the Method property to Welch-Costas. The default is 6.

Methods

step (To be removed) Permute input symbols using an algebraically derived permutation vector

Common to All System Objects	
release	Allow System object property value changes

Examples

Algebraic Interleaving and Deinterleaving

Create algebraic interleaver and deinterleaver objects having a length of 16.

```
interleaver = comm.AlgebraicInterleaver('Length',16);
```

```
Warning: COMM.ALGEBRAICINTERLEAVER will be removed in a future release. Use ALGINTRLV instead. Se
```

```
deinterleaver = comm.AlgebraicDeinterleaver('Length',16);
```

```
Warning: COMM.ALGEBRAICDEINTERLEAVER will be removed in a future release. Use ALGDEINTRLV insteac
```

Generate 8-ary data. Interleave and deinterleave the data.

```
data = randi([0 7],16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Compare the original, interleaved, and deinterleaved data.

```
[data,intData,deIntData]
```

```
ans = 16×3

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

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Algebraic Interleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.AlgebraicInterleaver` will be removed

Not recommended starting in R2019b

comm.AlgebraicInterleaver will be removed in a future release. Use `algintrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`algdeintrlv` | `algintrlv` | `deintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.AlgebraicInterleaver`
Package: `comm`

(To be removed) Permute input symbols using an algebraically derived permutation vector

Compatibility

step will be removed in a future release. Use `algintrlv` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The object uses an algebraically derived permutation vector, based on the algebraic method you specify in the Method property, to form the output. The input `X` must be a column vector of length specified by the Length property. `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.APPDecoder

Decode convolutional code by using APP method

Description

The `APPDecoder` system object performs a posteriori probability (APP) decoding of a convolutional code.

To decode convolutional code by using APP method:

1. Create the `comm.APPDecoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
appDec = comm.APPDecoder
appDec = comm.APPDecoder(Name,Value)
appDec = comm.APPDecoder(trellis,Name,Value)
```

Description

`appDec = comm.APPDecoder` creates an APP decoder System object, `appDec`, that decodes a convolutional code using the APP method.

`appDec = comm.APPDecoder(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.APPDecoder('Algorithm','True APP')` configures the System object, `appDec`, to implement true a posteriori probability decoding. Enclose each property name in quotes.

`appDec = comm.APPDecoder(trellis,Name,Value)` creates an APP decoder object, `appDec`, with the TrellisStructure property set to `trellis`.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`TrellisStructure` — Trellis description of constituent convolutional code
`poly2trellis(7,[171 133],171)` (default) | structure

Trellis description, specified as a MATLAB structure that contains the trellis description for a rate *K*/*N* code. *K* represents the number of input bit streams, and *N* represents the number of output bit streams.

You can either use the `poly2trellis` function to create the trellis structure or create it manually. For more about this structure, see “Trellis Description of a Convolutional Code” and the `istrellis` function.

The trellis structure contains these fields.

`numInputSymbols` — Number of symbols input to encoder
2^K

Number of symbols input to the encoder, specified as an integer equal to 2^K, where *K* is the number of input bit streams.

Data Types: `double`

`numOutputSymbols` — Number of symbols output from encoder
2^N

Number of symbols output from the encoder, specified as an integer equal to 2^N, where *N* is the number of output bit streams.

Data Types: `double`

`numStates` — Number of states in encoder
power of 2

Number of states in the encoder, specified as a power of 2.

Data Types: `double`

`nextStates` — Next states
matrix of integers

Next states for all combinations of current states and current inputs, specified as a matrix of integers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

`outputs` — Outputs
matrix of octal numbers

Outputs for all combinations of current states and current inputs, specified as a matrix of octal numbers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

Data Types: `struct`

`TerminationMethod` — Termination method of encoded frame
`'Truncated'` (default) | `'Terminated'`

Termination method of encoded frame, specified as `'Truncated'` or `'Terminated'`. When you set this property to `'Truncated'`, this System object assumes that the encoder stops after encoding the last symbol in the input frame. When you set this property to `'Terminated'`, this System object assumes that the encoder forces the trellis to end each frame in the all-zeros state by encoding additional symbols. If you use the `comm.ConvolutionalEncoder` System object to generate the encoded frame, this property value must match the property value of the convolutional encoder and this System object.

Data Types: `char` | `string`

`Algorithm` — Decoding algorithm
`'Max*'` (default) | `'True APP'` | `'Max'`

Decoding algorithm, specified as `'Max*'`, `'True APP'`, or `'Max'`. When you set this property to `'True APP'`, this System object implements true APP decoding. When you set this property to any other value, this System object uses approximations to increase the speed of the computations. For more information, see "Algorithms" on page 3-43.

Data Types: `char` | `string`

`NumScalingBits` — Number of scaling bits
`3` (default) | integer in the range [0, 8]

Number of scaling bits, specified as an integer in the range [0, 8]. This property specifies the number of bits the decoder uses to scale the input data to avoid losing precision during the computations.

Dependencies

To enable this property, set the "Algorithm" on page 3-0 property to `'Max*'`.

Data Types: `double`

`CodedBitLLROutputPort` — Option to enable coded-bit log-likelihood ratio output
`true or 1` (default) | `false or 0`

Option to enable coded-bit log-likelihood ratio (LLR) output, specified as a numeric or logical `1` (`true`) or `0` (`false`). To disable the second output when you call this System object, set this property to `0` (`false`).

Data Types: `logical`

Usage

Syntax

```
[LUD,LCD] = appDec(LU,LC)
LUD = appDec(LU,LC)
```

Description

`[LUD,LCD] = appDec(LU,LC)` performs APP decoding on the sequence of LLRs of encoder input bits, `LU`, and the sequence of LLRs of encoded bits, `LC`. The System object returns `LUD` and `LCD`. These output values are the updated versions of `LU` and `LC`, respectively, and are obtained based on the encoder information.

`LUD = appDec(LU,LC)` performs APP decoding with the `LCD` output disabled. To disable the `LCD` output, set the "CodedBitLLROutputPort" on page 3-0 property to `0` (`false`).

Input Arguments

LU — Sequence of LLRs of encoder input data

real-valued column vector

Sequence of LLRs of encoder input data, specified as a real-valued column vector. A positive soft input is interpreted as a logical `1`, and a negative soft input is interpreted as a logical `0`.

Data Types: `single` | `double`

LC — Sequence of LLRs of encoded data

real-valued column vector

Sequence of LLRs of encoded data, specified as a real-valued column vector of. A positive soft input is interpreted as a logical `1`, and a negative soft input is interpreted as a logical `0`.

Data Types: `single` | `double`

Output Arguments

LUD — Updated value of LU

real-valued column vector

Updated value of LU, returned as a real-valued column vector.

Data Types: `single` | `double`

LCD — Updated value of LC

real-valued column vector

Updated value of LC, returned as a real-valued column vector.

Data Types: `single` | `double`

Note If the convolutional code uses an alphabet of 2^n possible symbols, where n is the number of bits per input symbol, then the `LC` and `LCD` vector lengths are $L \times n$ for some positive integer L. Similarly, if the decoded data uses an alphabet of 2^k output symbols, where k is the number of bits per output symbol, then the `LU` and `LUD` vector lengths are $L \times k$.

This System object accepts a column vector input signal with any positive integer value for L. For variable-sized inputs, L can vary during multiple calls.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Decode Convolutional Code Using the APP Decoder

Specify noise variance and the frame length in bits. Create convolutional encoder, PSK modulator, and AWGN channel System objects.

```
noiseVar = 2e-1;
frameLength = 300;
convEncoder = comm.ConvolutionalEncoder('TerminationMethod','Truncated');
pskMod = comm.PSKModulator('BitInput',true,'PhaseOffset',0);
awgnChan = comm.AWGNChannel('NoiseMethod','Variance', ...
    'Variance',noiseVar);
```

Create convolutional APP decoder, PSK demodulator, and error rate System objects.

```
appDecoder = comm.APPDecoder(...
    'TrellisStructure',poly2trellis(7,[171 133]), ...
    'Algorithm','True APP','CodedBitLLROutputPort',false);
pskDemod = comm.PSKDemodulator('BitOutput',true,'PhaseOffset',0, ...
    'DecisionMethod','Approximate log-likelihood ratio', ...
    'Variance',noiseVar);
errRate = comm.ErrorRate;
```

Transmit a convolutionally encoded 8-PSK-modulated bit stream through an AWGN channel. Demodulate the received signal using soft-decision. Decode the demodulated signal using the APP decoder.

```
for counter = 1:5
     data = randi([0 1],frameLength,1);
     encodedData = convEncoder(data);
     modSignal = pskMod(encodedData);
     receivedSignal = awgnChan(modSignal);
     demodSignal = pskDemod(receivedSignal);
     % The APP decoder assumes a polarization of the soft inputs that is
     % inverse to that of the demodulator soft outputs. Change the sign of
     % demodulated signal.
     receivedSoftBits = appDecoder(zeros(frameLength,1),-demodSignal);
     % Convert from soft-decision to hard-decision.
     receivedBits = double(receivedSoftBits > 0);
     % Count errors
     errorStats = errRate(data,receivedBits);
end
```

Display the error rate information.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

```
Error rate = 0.000000
Number of errors = 0
```

Algorithms

This System object implements the soft-input-soft-output APP decoding algorithm according to [1] and [2].

The `'True APP'` option of the `Algorithm` property implements APP decoding as per the equations 20–23 in section V of [1]. To gain speed, the `'Max*'` and `'Max'` values of the `Algorithm` property approximate expressions like $\log\sum_i \exp(a_i)$ by other quantities. The `'Max'` option uses $\max(a_i)$ as the approximation. The `'Max*'` option uses $\max(a_i)$ plus a correction term given by the expression $\ln(1 + \exp(-|a_{i-1} - a_i|))$.

Setting the `Algorithm` property to `'Max*'` enables the `NumScalingBits` property of this System object. This property denotes the number of bits by which this System object scales the data it processes internally (multiplies the input by $2^{\texttt{NumScalingBits}}$ and divides the pre-output by the same factor). Use this property to avoid losing precision during computations.

References

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

[2] Viterbi, A.J. “An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes.” *IEEE Journal on Selected Areas in Communications* 16, no. 2 (February 1998): 260–64. https://doi.org/10.1109/49.661114.

[3] Benedetto, S., and G. Montorsi. “Performance of Continuous and Blockwise Decoded Turbo Codes.” *IEEE Communications Letters* 1, no. 3 (May 1997): 77–79. https://doi.org/10.1109/4234.585802.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`convenc` | `poly2trellis`

Objects

`comm.ConvolutionalEncoder` | `comm.TurboDecoder` | `comm.ViterbiDecoder`

Blocks

APP Decoder | Convolutional Encoder

Introduced in R2012a

arrayConfig

Phased array configuration object

Description

The `arrayConfig` object sets phased array configuration properties. Use an `arrayConfig` object to configure a uniform rectangular array (URA) with isotropic antenna elements, a uniform linear array (ULA) with isotropic antenna elements, or a single isotropic antenna element.

Creation

Syntax

```
cfgArray = arrayConfig
cfgArray = arrayConfig(Name,Value)
```

Description

`cfgArray = arrayConfig` creates a configuration object with default property values. The *x*-axis is normal to the plane on which the elements are placed. The default array is a 2-by-2 URA with an element spacing of 0.5 meter.

`cfgArray = arrayConfig(Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `arrayConfig('Size',[8 1],'ElementSpacing',0.1)` specifies an eight-element ULA along the *z*-axis with an element spacing of 0.1 meter.

Properties

`Element` — Array element
`'isotropic'` (default)

This property is read-only.

Array element, returned as `'isotropic'`. Array elements are isotropic radiators.

Data Types: `char` | `string`

`Size` — Antenna array size
`[2 2]` (default) | two-element row vector of positive integers

Antenna array size, specified as a two-element row vector of positive integers. The first element specifies the number of rows of the antenna array and the second element specifies the number of columns of the antenna array. The rows of the array are along the *z*-axis. The columns of the array are along the *y*-axis.

- When both elements of this vector are greater than 1, the array is a URA.

- When one element of this vector is 1, the array is a ULA.
- When both elements of this vector are 1, the array is a single isotropic element.

Array elements are indexed from top to bottom along a column, continuing to the next column from left to right. For more information, see “Array Alignment” on page 3-47.

Data Types: `double`

`ElementSpacing` — Antenna array element spacing
`0.5` (default) | positive scalar | two-element row vector

Antenna array element spacing in meters, specified as one of these values.

- A positive scalar — This value specifies the spacing between rows and the spacing between columns of the antenna array.
- A two-element vector of positive values — The first element of the vector specifies the spacing between rows of the antenna array. The second element specifies the spacing between columns of the antenna array.

The rows of the array are along the *z*-axis, and the columns of the array are along the *y*-axis. For more information, see “Array Alignment” on page 3-47.

Dependencies

To enable this property, set at least one element in the `Size` property vector to a value greater than `1`.

Data Types: `double`

Examples

Configure 4-by-4 URA

Configure a 4-by-4 URA with an element spacing of 0.1 meter along rows and 0.2 meter along columns.

```
cfgArray = arrayConfig("Size",[4 4],"ElementSpacing",[0.1 0.2])
```

```
cfgArray = 
  arrayConfig with properties:

           Element: 'isotropic'
              Size: [4 4]
    ElementSpacing: [0.1000 0.2000]
```

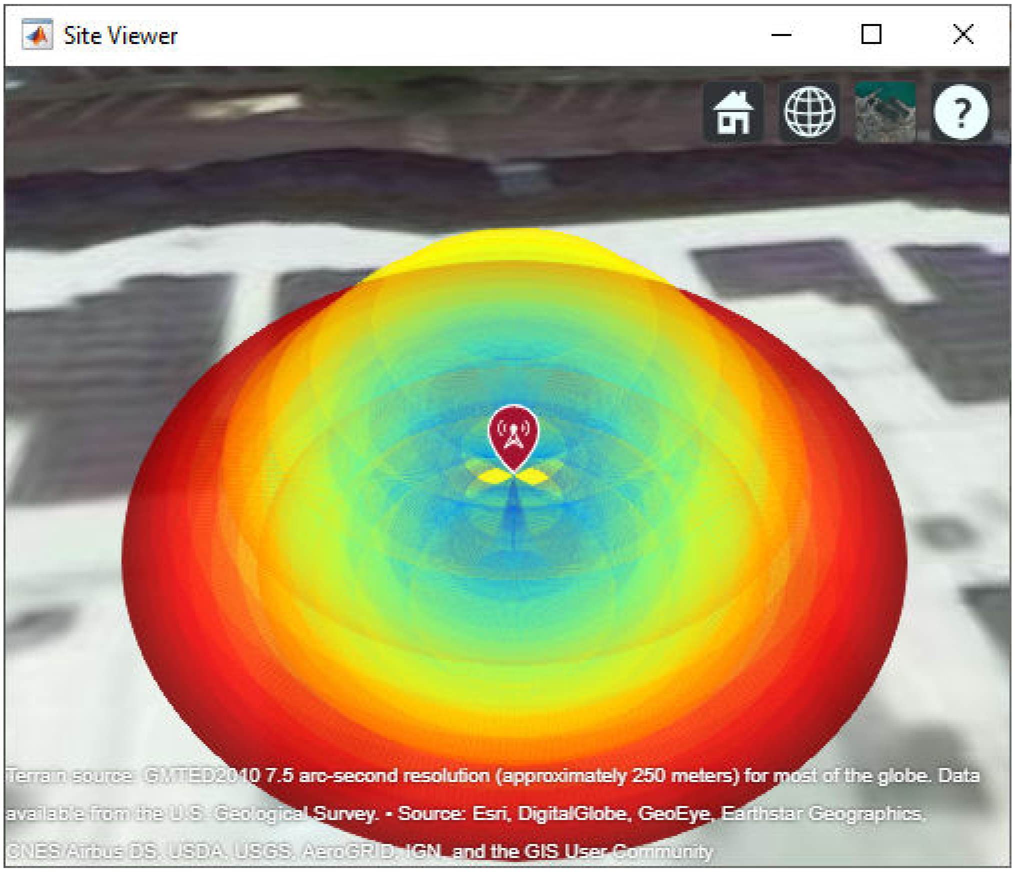
Configure Eight-Element ULA for Transmitter Site

Configure an eight-element ULA along the *z*-axis with an element spacing of 0.1 meter.

```
cfgArray = arrayConfig("Size",[8 1],"ElementSpacing",0.1);
```

Assign the array to a transmitter site and display the antenna pattern.

```
tx = txsite("Antenna",cfgArray);
pattern(tx,'Size',6);
```

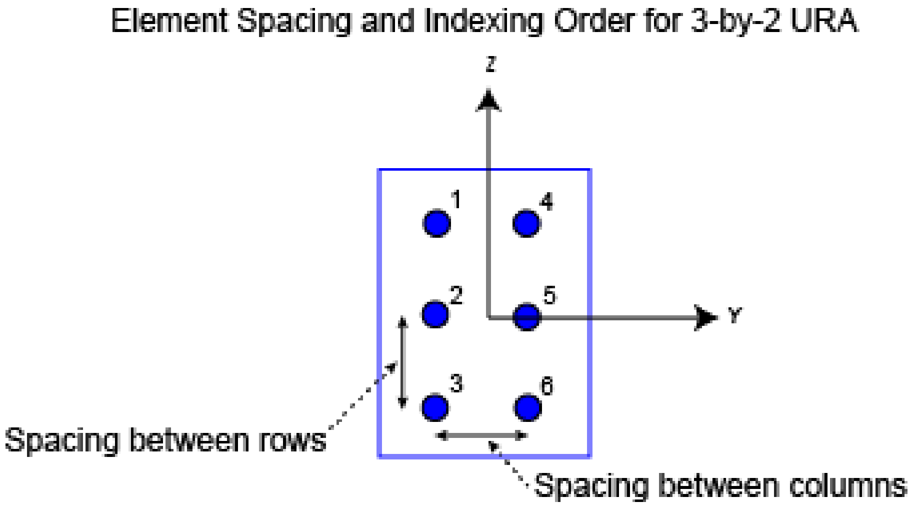

Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

More About

Array Alignment

Array elements are indexed from top to bottom along a column, continuing to the next column from left to right. The spacing between columns is the distance between adjacent elements in the same row. The spacing between rows is the distance between elements in the same column.

This illustration shows orientation and spacing between rows and columns for a URA of size 3-by-2. The array has three rows and two columns.



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`comm.Ray` | `comm.RayTracingChannel` | `phased.ConformalArray` | `phased.CustomAntennaElement` | `phased.IsotropicAntennaElement` | `phased.ULA` | `phased.URA` | `rxsite` | `siteviewer` | `txsite`

Introduced in R2020b

comm.AWGNChannel

Package: `comm`

Add white Gaussian noise to input signal

Description

comm.AWGNChannel adds white Gaussian noise to the input signal.

When applicable, if inputs to the object have a variable number of channels, the EbNo, EsNo, SNR, BitsPerSymbol, SignalPower, SamplesPerSymbol, and Variance properties must be scalars.

To add white Gaussian noise to an input signal:

1. Create the `comm.AWGNChannel` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
awgnchan = comm.AWGNChannel
awgnchan = comm.AWGNChannel(Name,Value)
```

Description

`awgnchan = comm.AWGNChannel` creates an additive white Gaussian noise (AWGN) channel System object, `awgnchan`. This object then adds white Gaussian noise to a real or complex input signal.

`awgnchan = comm.AWGNChannel(Name,Value)` creates a AWGN channel object, `awgnchan`, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`NoiseMethod` — Noise level method
`'Signal to noise ratio (Eb/No)'` (default) | `'Signal to noise ratio (Es/No)'` | `'Signal to noise ratio (SNR)'` | `'Variance'`

Noise level method, specified as `'Signal to noise ratio (Eb/No)'`, `'Signal to noise ratio (Es/No)'`, `'Signal to noise ratio (SNR)'`, or `'Variance'`. For more information, see Specifying the Variance Directly or Indirectly on page 3-68.

Data Types: `char`

`EbNo` — Ratio of energy per bit to noise power spectral density
`10` (default) | scalar | row vector

Ratio of energy per bit to noise power spectral density (Eb/No) in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`.

Data Types: `double`

`EsNo` — Ratio of energy per symbol to noise power spectral density
`10` (default) | scalar | row vector

Ratio of energy per symbol to noise power spectral density (Es/No) in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Es/No)'`.

Data Types: `double`

`SNR` — Ratio of signal power to noise power
`10` (default) | scalar | row vector

Ratio of signal power to noise power in decibels, specified as a scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (SNR)'`.

Data Types: `double`

`BitsPerSymbol` — Number of bits per symbol
`1` (default) | positive integer

Number of bits per symbol, specified as a positive integer.

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`.

Data Types: `double`

`SignalPower` — Input signal power

1 (default) | positive scalar | row vector

Input signal power in watts, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels. The object assumes a nominal impedance of 1 Ω.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'`, `'Signal to noise ratio (Es/No)'`, or `'Signal to noise ratio (SNR)'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

1 (default) | positive integer | row vector

Number of samples per symbol, specified as a positive integer or 1-by-N_C vector. N_C is the number of channels.

Dependencies

This property applies when NoiseMethod is set to `'Signal to noise ratio (Eb/No)'` or `'Signal to noise ratio (Es/No)'`.

Data Types: `double`

`VarianceSource` — Source of noise variance

`'Property'` (default) | `'Input port'`

Source of noise variance, specified as `'Property'` or `'Input port'`.

- Set `VarianceSource` to `'Property'` to specify the noise variance value using the Variance property.
- Set `VarianceSource` to `'Input port'` to specify the noise variance value using an input to the object, when you call it as a function.

For more information, see Specifying the Variance Directly or Indirectly on page 3-68.

Dependencies

This property applies when NoiseMethod is `'Variance'`.

Data Types: `char`

`Variance` — White Gaussian noise variance

1 (default) | positive scalar | row vector

White Gaussian noise variance, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels.

Tunable: Yes

Dependencies

This property applies when NoiseMethod is set to `'Variance'` and VarianceSource is set to `'Property'`.

Data Types: double

RandomStream — Source of random number stream
'Global stream' (default) | 'mt19937ar with seed'

Source of random number stream, specified as 'Global stream' or 'mt19937ar with seed'.

- When you set RandomStream to 'Global stream', the object uses the MATLAB default random stream to generate random numbers. To generate reproducible numbers using this object, you can reset the MATLAB default random stream. For example reset(RandStream.getGlobalStream). For more information, see RandStream.
- When you set RandomStream to 'mt19937ar with seed', the object uses the mt19937ar algorithm for normally distributed random number generation. In this scenario, when you call the reset function, the object reinitializes the random number stream to the value of the Seed property. You can generate reproducible numbers by resetting the object.

For a complex input signal, the object creates the random data as follows:

noise = randn(N_S,N_C)+1i(randn(N_S,N_C))

N_S is the number of samples and N_C is the number of channels.

Dependencies

This property applies when NoiseMethod is set to 'Variance'.

Data Types: char

Seed — Initial seed
67 (default) | nonnegative integer

Initial seed of the mt19937ar random number stream, specified as a nonnegative integer. For each call to the reset function, the object reinitializes the mt19937ar random number stream to the Seed value.

Dependencies

This property applies when RandomStream is set to 'mt19937ar with seed'.

Data Types: double

Usage

Syntax

```
outsignal = awgnchan(insignal)
outsignal = awgnchan(insignal,var)
```

Description

outsignal = awgnchan(insignal) adds white Gaussian noise, as specified by awgnchan, to the input signal. The result is returned in outsignal.

outsignal = awgnchan(insignal,var) specifies the variance of the white Gaussian noise. This syntax applies when you set the NoiseMethod to 'Variance' and VarianceSource to 'Input port'.

For example:

```
awgnchan = comm.AWGNChannel('NoiseMethod','Variance', ...
     'VarianceSource','Input port');
var = 12;
...
outsignal = awgnchan(insignal,var);
```

Input Arguments

`insignal` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, an N_S-element vector, or an N_S-by-N_C matrix. N_S is the number of samples and N_C is the number of channels.

Data Types: `double`
Complex Number Support: Yes

`var` — Variance of additive white Gaussian noise
positive scalar | row vector

Variance of additive white Gaussian noise, specified as a positive scalar or 1-by-N_C vector. N_C is the number of channels, as determined by the number of columns in the input signal matrix.

Output Arguments

`outsignal` — Output signal
matrix

Output signal, returned with the same dimensions as `insignal`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Create Default AWGN Channel System Object

Create an AWGN channel System object with the default configuration. Pass signal data through this channel.

Create an AWGN channel object and signal data.

```
awgnchan = comm.AWGNChannel;
insignal = randi([0 1],100,1);
```

Send the input signal through the channel.

```
outsignal = awgnchan(insignal);
```

Add White Gaussian Noise to 8-PSK Signal

Modulate an 8-PSK signal, add white Gaussian noise, and plot the signal to visualize the effects of the noise.

Create a M-PSK modulator System object™. The default modulation order for the object is 8.

```
pskModulator = comm.PSKModulator;
```

Modulate the signal.

```
modData = pskModulator(randi([0 7],2000,1));
```

Add white Gaussian noise to the modulated signal by passing the signal through an additive white Gaussian noise (AWGN) channel.

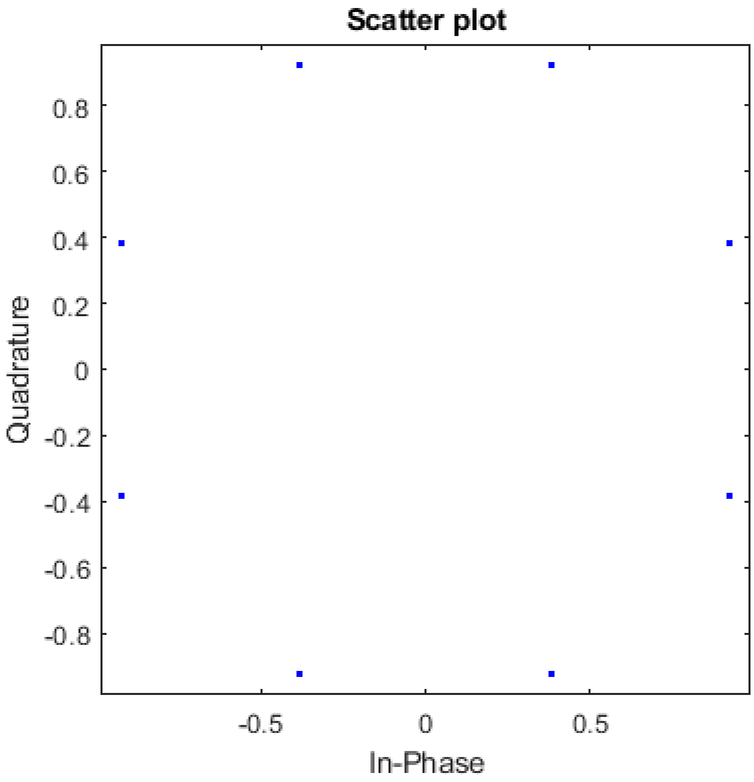
```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',3);
```

Transmit the signal through the AWGN channel.

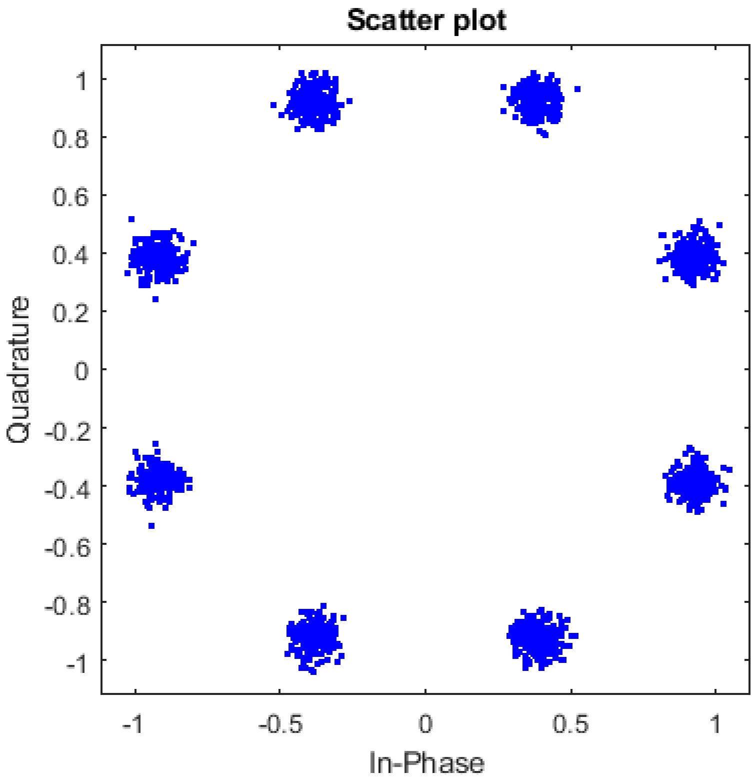
```
channelOutput = channel(modData);
```

Plot the noiseless and noisy data by using scatter plots to visualize the effects of the noise.

```
scatterplot(modData)
```



```
scatterplot(channelOutput)
```



Change the `EbNo` property to 10 dB to increase the noise.

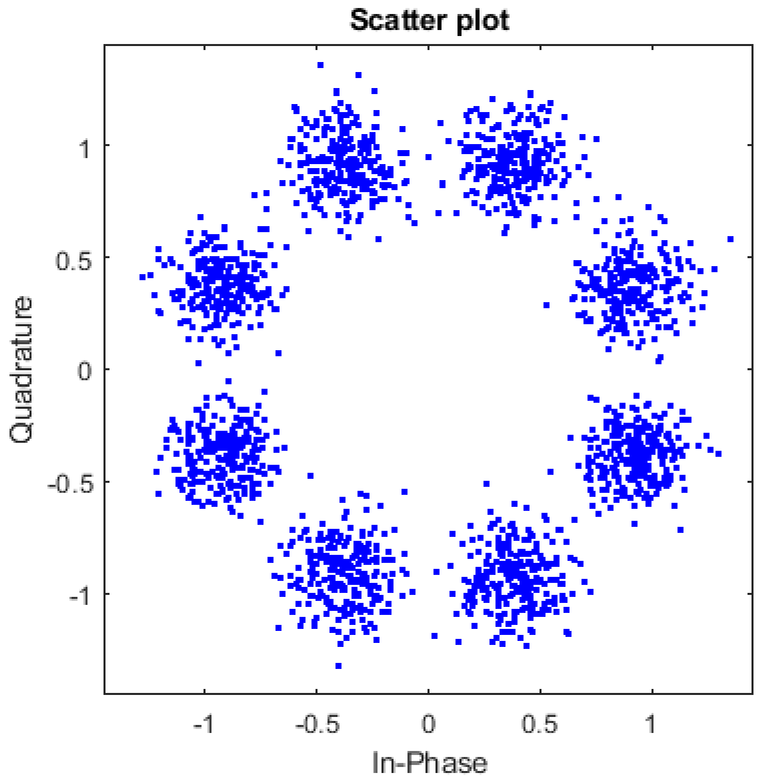
```
channel.EbNo = 10;
```

Pass the modulated data through the AWGN channel.

```
channelOutput = channel(modData);
```

Plot the channel output. You can see the effects of increased noise.

```
scatterplot(channelOutput)
```



Process Signals When Number of Channels Changes

Pass a single-channel and multichannel signal through an AWGN channel System object™.

Create an AWGN channel System object with the Eb/No ratio set for a single channel input. In this case, the EbNo property is a scalar.

```
channel = comm.AWGNChannel('EbNo',15);
```

Generate random data and apply QPSK modulation.

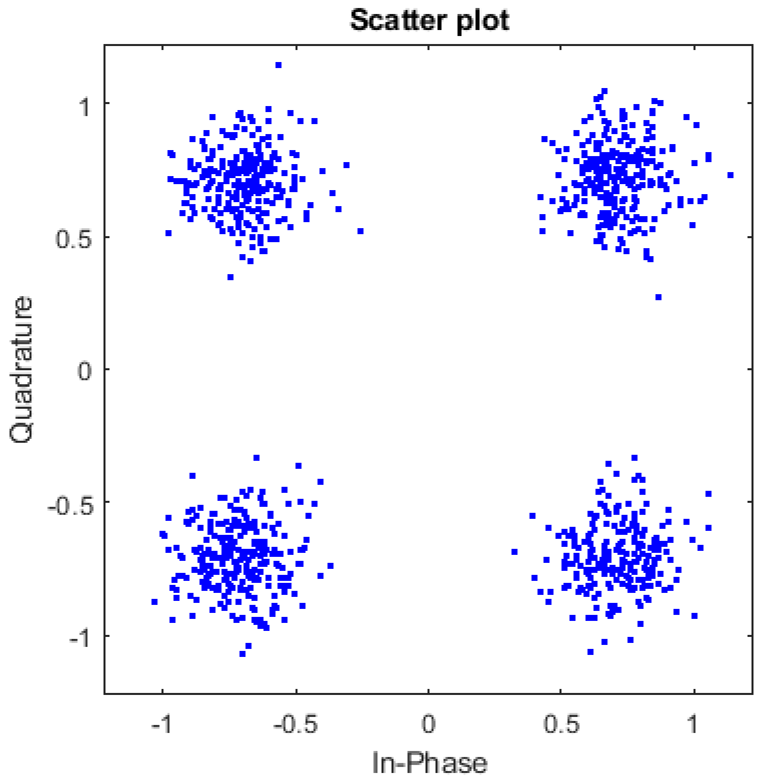
```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Pass the modulated data through the AWGN channel.

```
rxSig = channel(modData);
```

Plot the noisy constellation.

```
scatterplot(rxSig)
```



Generate two-channel input data and apply QPSK modulation.

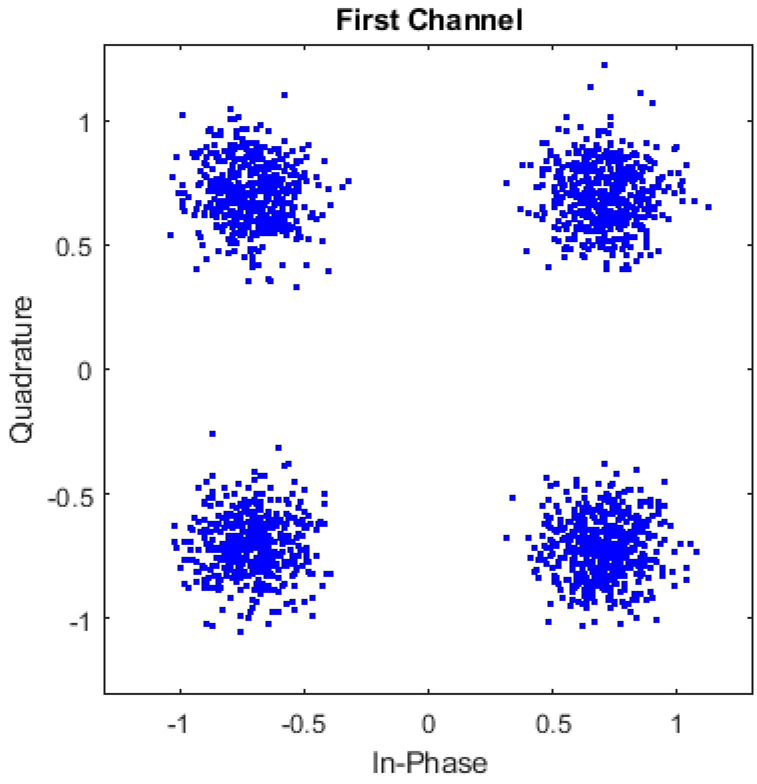
```
data = randi([0 3],2000,2);
modData = pskmod(data,4,pi/4);
```

Pass the modulated data through the AWGN channel.

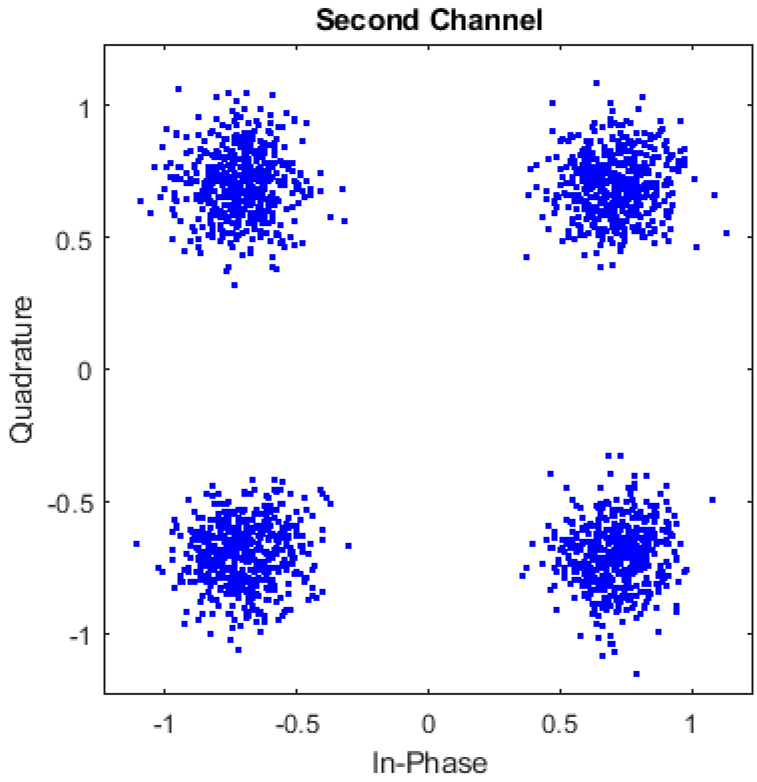
```
rxSig = channel(modData);
```

Plot the noisy constellations. Each channel is represented as a single column in `rxSig`. The plots are nearly identical, because the same Eb/No value is applied to both channels.

```
scatterplot(rxSig(:,1))
title('First Channel')
```



```
scatterplot(rxSig(:,2))
title('Second Channel')
```



Modify the AWGN channel object to apply a different Eb/No value to each channel. To apply different values, set the `EbNo` property to a 1-by-2 vector. When changing the dimension of the `EbNo` property, you must release the AWGN channel object.

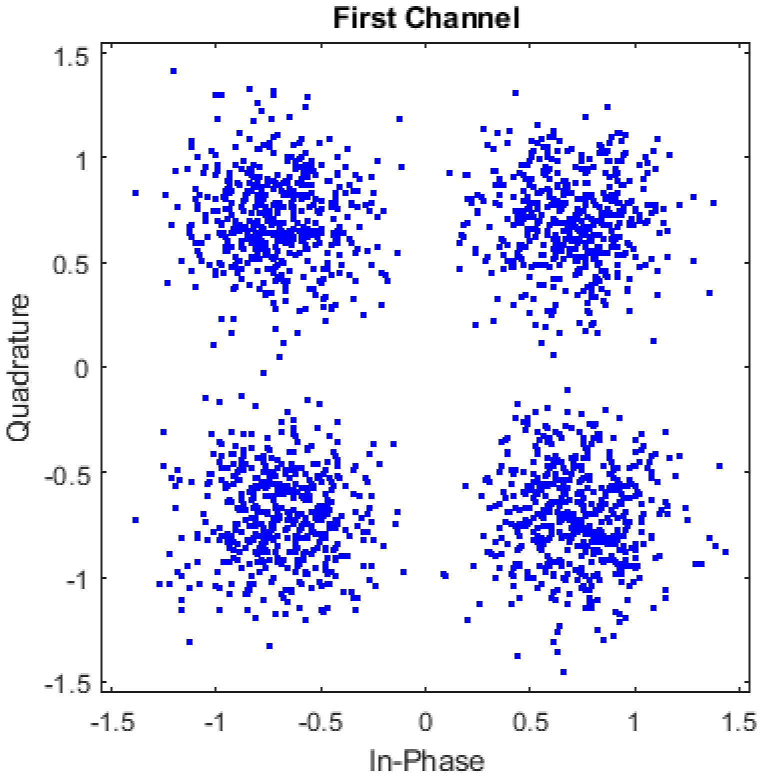
```
release(channel)
channel.EbNo = [10 20];
```

Pass the data through the AWGN channel.

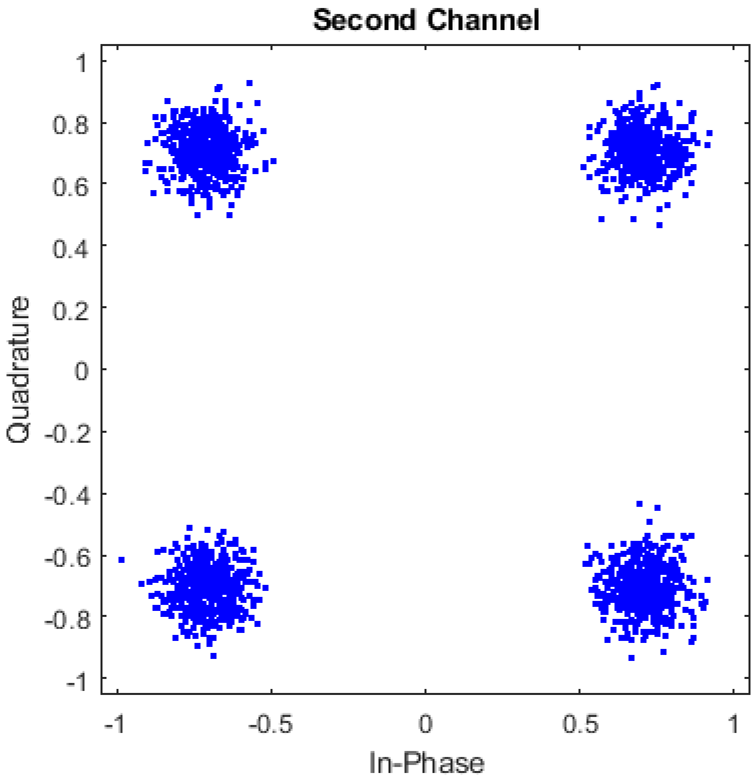
```
rxSig = channel(modData);
```

Plot the noisy constellations. The first channel has significantly more noise due to its lower Eb/No value.

```
scatterplot(rxSig(:,1))
title('First Channel')
```



```
scatterplot(rxSig(:,2))
title('Second Channel')
```



Add AWGN Using Noise Variance Input Port

Apply the noise variance input as a scalar or a row vector, with a length equal to the number of channels of the current signal input.

Create an AWGN channel System object™ with the `NoiseMethod` property set to '`Variance`' and the `VarianceSource` property set to '`Input port`'.

```
channel = comm.AWGNChannel('NoiseMethod','Variance', ...
'VarianceSource','Input port');
```

Generate random data for two channels and apply 16-QAM modulation.

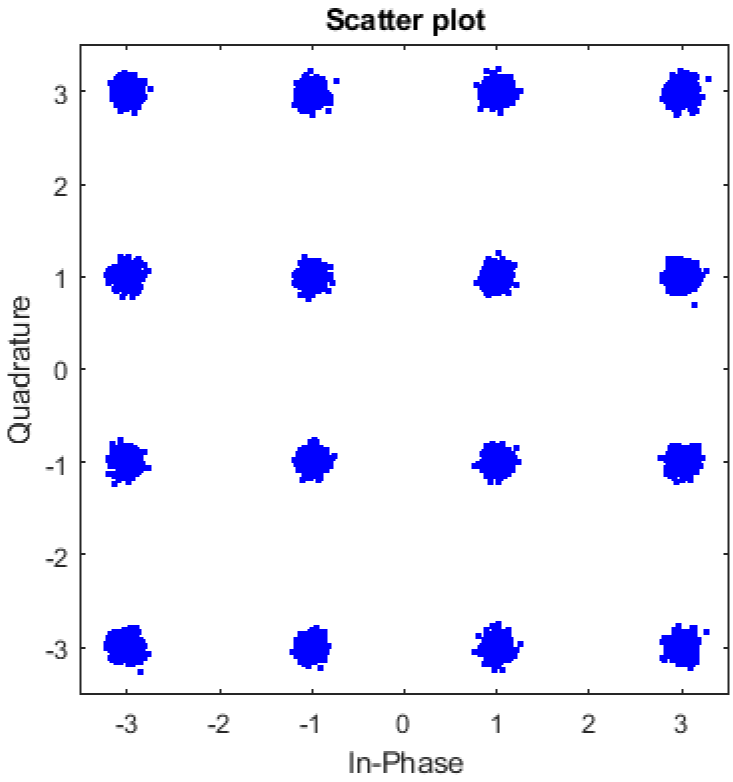
```
data = randi([0 15],10000,2);
txSig = qammod(data,16);
```

Pass the modulated data through the AWGN channel. The AWGN channel object processes data from two channels. The variance input is a 1-by-2 vector.

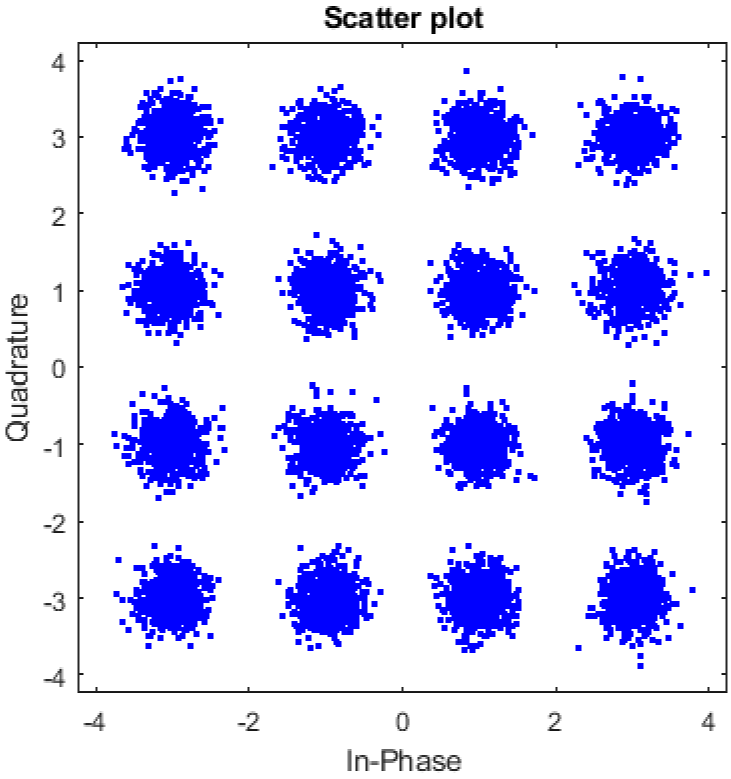
```
rxSig = channel(txSig,[0.01 0.1]);
```

Plot the constellation diagrams for the two channels. The second signal is noisier because its variance is ten times larger.

```
scatterplot(rxSig(:,1))
```

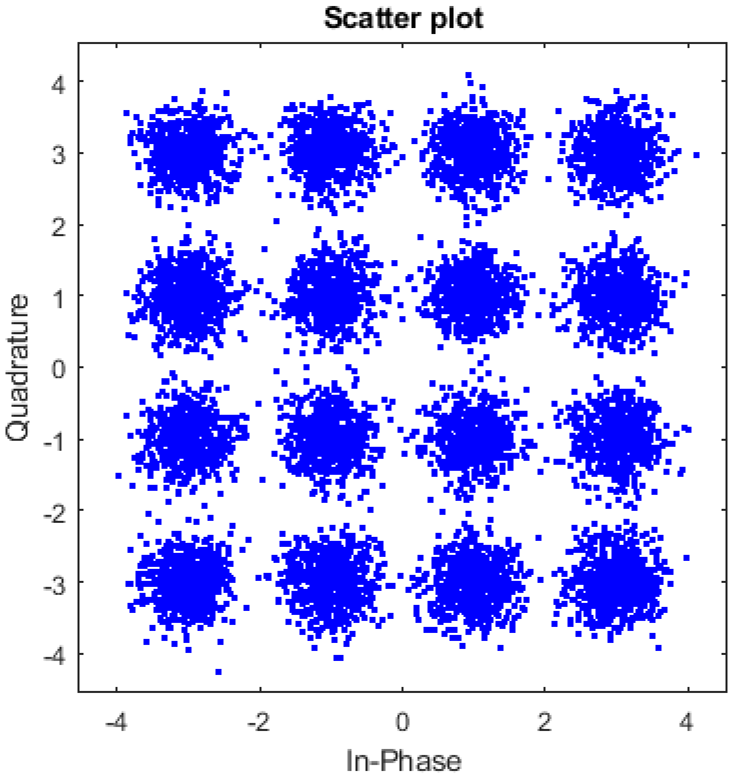


```
scatterplot(rxSig(:,2))
```

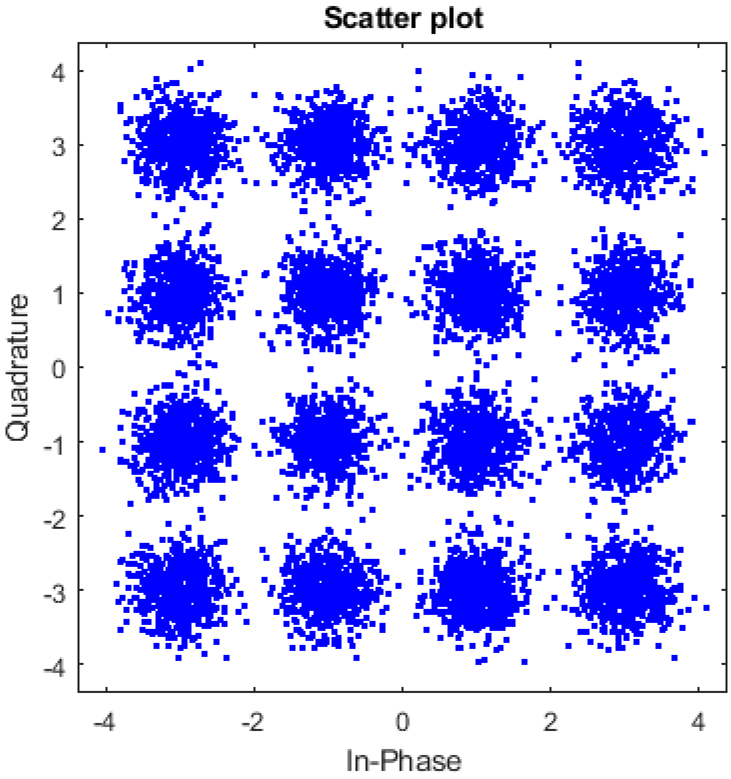


Repeat the process where the noise variance input is a scalar. The same variance is applied to both channels. The constellation diagrams are nearly identical.

```
rxSig = channel(txSig,0.2);
scatterplot(rxSig(:,1))
```



```
scatterplot(rxSig(:,2))
```



Set Random Number Seed for Repeatability

Specify a seed to produce the same outputs when using a random stream in which you specify the seed.

Create an AWGN channel System object™. Set the `NoiseMethod` property to `'Variance'`, the `RandomStream` property to `'mt19937ar with seed'`, and the `Seed` property to `99`.

```
channel = comm.AWGNChannel( ...
    'NoiseMethod','Variance', ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',99);
```

Pass data through the AWGN channel.

```
y1 = channel(zeros(8,1));
```

Pass another all-zeros vector through the channel.

```
y2 = channel(zeros(8,1));
```

Because the seed changes between function calls, the output is different.

```
isequal(y1,y2)
```

```
ans = logical
   0
```

Reset the AWGN channel object by calling the `reset` function. The random data stream is reset to the initial seed of `99`.

```
reset(channel);
```

Pass the all-zeros vector through the AWGN channel.

```
y3 = channel(zeros(8,1));
```

Confirm that the two signals are identical.

```
isequal(y1,y3)
```

```
ans = logical
   1
```

Algorithms

Relationship Among Eb/No, Es/No, and SNR Modes

For uncoded complex input signals, comm.AWGNChannel relates $E_b/N_{0,}$ E_s/N_0, and SNR according to these equations:

$E_s/N_0 = N_{sps} \times SNR$

$E_s/N_0 = E_b/N_0 + 10\log_{10}(k)$ in dB

where

- E_s represents the signal energy in joules.
- E_b represents the bit energy in joules.
- N_0 represents the noise power spectral density in watts/Hz.
- N_{sps} represents the number of samples per symbol, SamplesPerSymbol.
- k represents the number of information bits per input symbol, BitsPerSymbol.

For real signal inputs, the comm.AWGNChannel relates E_s/N_0 and SNR according to this equation:

$E_s/N_0 = 0.5\ (N_{sps}) \times SNR$

Note

- All values of power assume a nominal impedance of 1 ohm.
- The equation for the real case differs from the corresponding equation for the complex case by a factor of 2. Specifically, the object uses a noise power spectral density of $N_0/2$ watts/Hz for real input signals, versus N_0 watts/Hz for complex signals.

For more information, see AWGN Channel Noise Level.

Specifying the Variance Directly or Indirectly

To directly specify the variance of the noise generated by comm.AWGNChannel, specify VarianceSource as:

- `'Property'`, then set NoiseMethod to `'Variance'` and specify the variance with the Variance property.
- `'Input port'` then specify the variance level for the object as an input with an input argument, var.

To specify variance indirectly, that is, to have it calculated by comm.AWGNChannel, specify VarianceSource as `'Property'` and the NoiseMethod as:

- `'Signal to noise ratio (Eb/No)'`, where the object uses these properties to calculate the variance:
  - EbNo, the ratio of bit energy to noise power spectral density
  - BitsPerSymbol
  - SignalPower, the actual power of the input signal samples
  - SamplesPerSymbol
- `'Signal to noise ratio (Es/No)'`, where the object uses these properties to calculate the variance:
  - EsNo, the ratio of signal energy to noise power spectral density
  - SignalPower, the actual power of the input signal samples
  - SamplesPerSymbol
- `'Signal to noise ratio (SNR)'`, where the object uses these properties to calculate the variance:
  - SNR, the ratio of signal power to noise power
  - SignalPower, the actual power of the input signal samples

Changing the number of samples per symbol (SamplesPerSymbol) affects the variance of the noise added per sample, which also causes a change in the final error rate.

$$NoiseVariance = \text{SignalPower} \times \text{SamplesPerSymbol} / 10^{(\text{EsNo})/10}$$

Tip Select the number of samples per symbol based on what constitutes a symbol and the oversampling applied to it. For example, a symbol could have 3 bits and be oversampled by 4. For more information, see AWGN Channel Noise Level.

References

[1] Proakis, John G. *Digital Communications*. 4th Ed. McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Blocks
AWGN Channel | MIMO Fading Channel

Objects
`comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Functions
`bsc`

Topics
AWGN Channel

Introduced in R2012a

comm.BarkerCode

Package: `comm`

Generate bipolar Barker code

Description

The `comm.BarkerCode` System object generates a bipolar Barker code. Barker codes have low autocorrelation properties. The short length and low correlation sidelobes make Barker codes useful for frame synchronization in digital communications systems. For more information, see "Barker Codes" on page 3-73.

To generate a Barker code:

1. Create the `comm.BarkerCode` object and set its properties.
2. Call the object, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
barkerCode = comm.BarkerCode
barkerCode = comm.BarkerCode(Name,Value)
```

Description

`barkerCode = comm.BarkerCode` creates a bipolar Barker code generator System object to generate a Barker code.

`barkerCode = comm.BarkerCode(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.BarkerCode('Length',11,'SamplesPerFrame','11')` configures a bipolar Barker code generator System object to output a length 11 Barker code in an 11-sample frame. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Length` — Length of generated code
7 (default) | 1 | 2 | 3 | 4 | 5 | 11 | 13

Length of the generated code, specified as `1`, `2`, `3`, `4`, `5`, `7`, `11`, or `13`. For more information, see "Barker Codes" on page 3-73.

Example: `'Length',2` outputs the Barker code [–1;1].

Data Types: `double`

`SamplesPerFrame` — Samples per output frame
`1` (default) | positive integer

Samples per output frame, specified as a positive integer. If `SamplesPerFrame` is *M*, the object outputs a frame containing *M* samples comprised of length *N* Barker code sequences. If necessary, the object repeats the code sequence to reach *M* samples. *N* is the length of the generated code, which is set by the `Length` property.

Data Types: `double`

`OutputDataType` — Output data type
`double` (default) | `int8`

Output data type, specified as `double` or `int8`.

Data Types: `char` | `string`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

```
y = barkerCode
```

Description

`y = barkerCode` outputs a Barker code frame, as a column vector. If the frame length exceeds the Barker code length, the object fills the frame by repeating the Barker code.

Set the data type of the output with the `OutputDataType` property.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.BarkerCode`

clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

Examples

Generate Barker Code Sequence

Create a Barker code System object with 10 samples per frame.

```
barker = comm.BarkerCode('SamplesPerFrame',10)
```

```
barker = 
  comm.BarkerCode with properties:

              Length: 7
     SamplesPerFrame: 10
      OutputDataType: 'double'
```

Generate multiple frames by using the default Barker code sequence of length 7. The code wraps within the frame and continues in the next frame.

```
for ii = 1:2
    seq = barker()
end
```

```
seq = 10×1

    -1
    -1
    -1
     1
     1
    -1
     1
    -1
    -1
    -1


seq = 10×1

     1
     1
    -1
     1
    -1
    -1
    -1
     1
     1
    -1
```

Compute Barker Code Sidelobe Level

Compute the peak sidelobe level for each Barker code.

```
CodeLength = [1 2 3 4 5 7 11 13]';
psl = zeros(length(CodeLength),1);
ac = dsp.Autocorrelator;
barker = comm.BarkerCode;
for ii=1:length(CodeLength)
    spf = CodeLength(ii);
    barker.Length = CodeLength(ii);
    barker.SamplesPerFrame = spf;
    seq = barker();
    sll_dB = 20*log10(abs(ac(seq)));
    psl(ii) = -(max(sll_dB));
    release(barker);
    release(ac);
end
Sidelobe_dB = psl;
T = table(CodeLength,Sidelobe_dB)
```

```
T=8×2 table
    CodeLength    Sidelobe_dB
    __________    ___________

         1              0
         2        -6.0206
         3        -9.5424
         4        -12.041
         5        -13.979
         7        -16.902
        11        -20.828
        13        -22.279
```

More About

Barker Codes

Barker codes have a maximum autocorrelation sequence, which has off-peak autocorrelations no larger than 1.

A correlation sidelobe is the correlation of a codeword with a time-shifted version of itself. The correlation sidelobe, C_k, for a k-symbol shift of an N-bit code sequence, $\{X_j\}$, is

$$C_k = \sum_{j=1}^{N-k} X_j X_{j+k}$$

For j=1, 2, 3,..., N, X_j is an individual code symbol that is equal to +1 or –1. The adjacent symbols are assumed to be 0.

The output code is in a bipolar format with 0 and 1 mapped to 1 and –1. The maximum known Barker code length is 13. The short length and low correlation sidelobes make Barker codes useful for frame synchronization in digital communications systems. The Barker code generator outputs the Barker codes listed in this table.

Barker Code Length	Barker Code	Sidelobe Level
1	[–1]	0 dB
2	[–1; 1]	–6 dB
3	[–1; –1; 1]	–9.5 dB
4	[–1; –1; 1; –1]	–12 dB
5	[–1; –1; –1; 1; –1]	–14 dB
7	[–1; –1; –1; 1; 1; –1; 1]	–16.9 dB
11	[–1; –1; –1; 1; 1; 1; –1; 1; 1; –1; 1]	–20.8 dB
13	[–1; –1; –1; –1; –1; 1; 1; –1; –1; 1; –1; 1; –1]	–22.3 dB

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.HadamardCode` | `comm.OVSFCode` | `comm.WalshCode`

Blocks

Barker Code Generator

Topics

“Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization”

Introduced in R2012a

comm.BasebandFileReader

Package: `comm`

Read baseband signals from file

Description

The `comm.BasebandFileReader` object reads a baseband signal from a specific type of binary file written by `comm.BasebandFileWriter`. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. The `SampleRate` and `CenterFrequency` properties are saved when the file is created. The `comm.BasebandFileReader` object automatically reads the sample rate, center frequency, number of channels, and any descriptive data and saves them to its read-only properties.

To create an input signal from a saved baseband file:

1. Create a `comm.BasebandFileReader` object and set the properties of the object.
2. Call `step` to generate a baseband signal from saved data.
3. Call `release` to close the file.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`bbr = comm.BasebandFileReader` returns a baseband reader object, `bbr`, using the default properties.

`bbr = comm.BasebandFileReader(fname)` returns a baseband reader object and sets `fname` as the `Filename` property.

`bbr = comm.BasebandFileReader(fname,spf)` also sets `spf` as the `SamplesPerFrame` property.

`bbr = comm.BasebandFileReader(___,Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

Example:

```
bbr = comm.BasebandFileReader('recorded_data',100);
```

Properties

Filename — Name of the baseband file to read

`'example.bb'` (default) | character vector

Name of the baseband file to read, specified as a character vector. Specify the absolute path only if the file is not on the MATLAB path. Only the absolute path is saved and displayed.

`SampleRate` — Sample rate of the saved baseband signal
`1` (default) | positive scalar

This property is read-only.

Sample rate of the saved baseband signal in Hz.

`CenterFrequency` — Center frequency of the saved baseband signal
`100000000` (default) | positive scalar | row vector

This property is read-only.

Center frequency of the saved baseband signal in Hz. When this property is a row vector, each element represents the center frequency of a channel in a multichannel signal.

`NumChannels` — Number of channels of the saved baseband signal
`1` (default) | positive integer

This property is read-only.

Number of channels of the saved baseband signal.

`Metadata` — Data describing the baseband signal
`struct()` (default) | structure

This property is read-only.

Data describing the baseband signal. If the file has no descriptive data, this property is an empty structure.

`SamplesPerFrame` — Number of samples per output frame
`100` (default) | positive integer

Number of samples per output frame, specified as a positive integer.

Data Types: `double`

`CyclicRepetition` — Flag to repeatedly read baseband file
`false` (default) | `true`

Flag to repeatedly read baseband file, specified as a logical scalar. To repeatedly read the baseband file specified by `Filename`, set this property to `true`.

Methods

info	Characteristic information about baseband file reader

Common to All System Objects	
`release`	Allow System object property value changes
`step`	Generate baseband signal from file
`reset`	Reset states of baseband file reader object
`isDone`	Read status of baseband file samples

Examples

Read Baseband Data from File

Read a baseband signal from file using the `comm.BasebandFileReader` System object.

Create a baseband file reader object.

```
bbr = comm.BasebandFileReader('example.bb')
```

```
bbr = 
  comm.BasebandFileReader with properties:

            Filename: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\10\tpa899180b\comm-ex87872352\exampl
          SampleRate: 1
     CenterFrequency: 100000000
         NumChannels: 1
            Metadata: [1x1 struct]
     SamplesPerFrame: 100
    CyclicRepetition: false
```

Use the `info` method to gain additional information about `bbr`. The file contains 10000 samples of type `'double'`. No samples have been read.

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 0
```

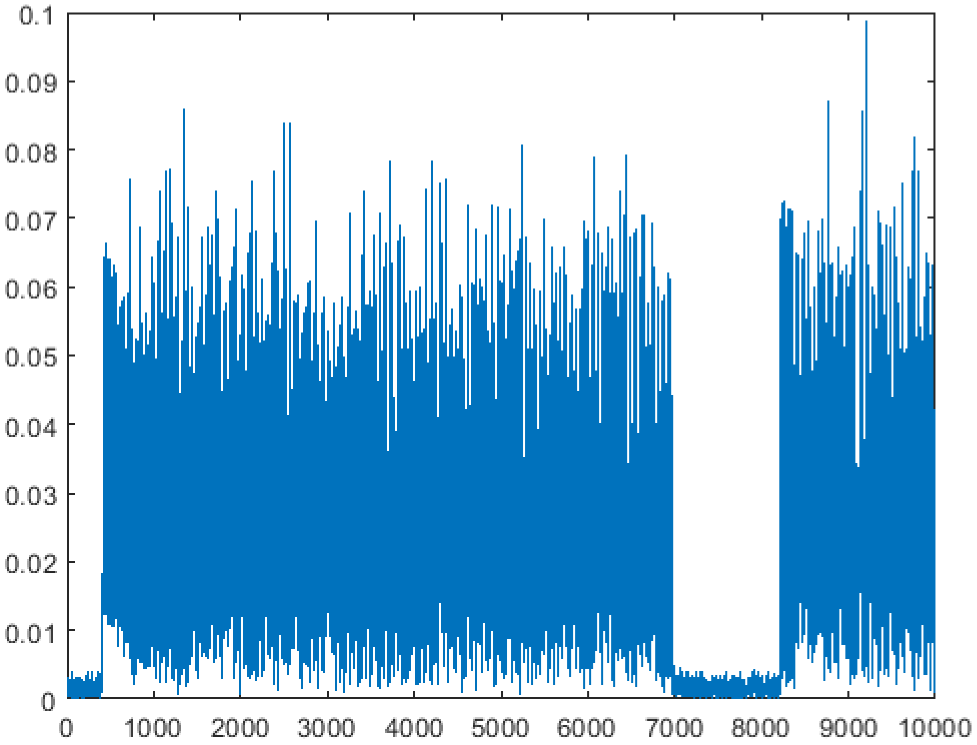
Reading Data with Multiple Calls to Baseband File Reader Object

Since the `NumSamplesPerFrame` is 100, and `NumSamplesInData` is 10000, reading the entire contents of the `example.bb` file will require multiple calls to the `bbr` object. This can be achieved by using the `isDone` method to terminate a `while` loop.

```
y = [];

while ~isDone(bbr)
    x = bbr();
    y = cat(1,y,x);
end
```

Plot the absolute magnitude of the baseband data.

```
plot(abs(y))
```



Confirm that all the samples have been read.

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 10000
```

The total number of samples and the number of samples read are the same.

Release the baseband file reader resources.

```
release(bbr)
```

Reading Data with Single Call to Baseband File Reader Object

Alternatively, to read all samples from the file in one call to the object, you can set the samples per frame equal to the number of samples in the data file.

```
bbrinfo = bbr.info
```

```
bbrinfo = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 10000
```

```
bbr.SamplesPerFrame = bbrinfo.NumSamplesInData
```

```
bbr = 
  comm.BasebandFileReader with properties:

            Filename: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\10\tpa899180b\comm-ex87872352\examp
          SampleRate: 1
     CenterFrequency: 100000000
         NumChannels: 1
            Metadata: [1x1 struct]
     SamplesPerFrame: 10000
    CyclicRepetition: false
```

Now read the entire contents of the `example.bb` file with a single call to the `bbr` object. Confirm that all the samples have been read.

```
xx = bbr();
isequal(y,xx)
```

```
ans = logical
   1
```

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 10000
            DataType: 'double'
      NumSamplesRead: 10000
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BasebandFileWriter`

Introduced in R2016b

info

System object: `comm.BasebandFileReader`
Package: `comm`

Characteristic information about baseband file reader

Syntax

```
s = info(bbr)
```

Description

`s = info(bbr)` returns a structure, `s`, containing characteristic information for the `BasebandFileReader` System object, `bbr`. `s` has these fields:

- `NumSamplesInData` is the total number of baseband data samples in the file, returned as a positive integer.
- `DataType` is the data type of the baseband signal in the file.
- `NumSamplesRead` is the number of samples that have been read from the file, returned as a positive integer. It cannot exceed the `NumSamplesInData` property when `CyclicRepetition` is `false`.

Introduced in R2016b

comm.BasebandFileWriter

Package: comm

Write baseband signals to file

Description

A baseband file is a specific type of binary file written by the comm.BasebandFileWriter System object. Baseband signals are typically down-converted from a nonzero center frequency to 0 Hz. The SampleRate and CenterFrequency properties are saved when the file is created.

To save a baseband signal to a file:

1. Create a comm.BasebandFileWriter object and set the properties of the object.
2. Call step to save a baseband signal to a file.
3. Call release to save the baseband signal to a file and to close the file.

Note Alternatively, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

bbw = comm.BasebandFileWriter returns a baseband writer object, bbw, using the default properties.

bbw = comm.BasebandFileWriter(fname) returns bbw and sets fname as the Filename property.

bbw = comm.BasebandFileWriter(fname,fs) also sets fs as the SampleRate property.

bbw = comm.BasebandFileWriter(fname,fs,fc) also sets fc as the CenterFrequency property.

bbw = comm.BasebandFileWriter(fname,fs,fc,md) also sets structure md as the MetaData property.

bbw = comm.BasebandFileWriter(___ ,Name,Value) specifies additional properties using Name,Value pairs. Unspecified properties have default values.

Example:

```
bbw = comm.BasebandFileWriter('qpsk_data.bb',10e6,2e9);
```

Properties

Filename — Name of saved file

'untitled.bb' (default) | character vector

Name of saved file, specified as a character vector. The filename can include a relative or an absolute path.

SampleRate — Sample rate of output signal

1 (default) | positive scalar

Sample rate of the output signal, specified in Hz as a positive scalar.

CenterFrequency — Center frequency of the baseband signal

100000000 (default) | positive integer scalar | row vector

Center frequency of the baseband signal, specified in Hz as a positive integer scalar or row vector. If CenterFrequency is a row vector, each element corresponds to a channel.

Metadata — Data describing the baseband signal

empty structure (default) | structure

Data describing the baseband signal, specified as a structure. The structure can have any number of fields and any field name. The field values can be of any numeric, logical, or character data type and have any number of dimensions.

NumSamplesToWrite — Number of samples to save

Inf (default) | positive integer

Number of samples to save, specified as a positive integer.

- To write all the baseband signal samples to a file, set NumSamplesToWrite to Inf.
- To write only the last NumSamplesToWrite samples to a file, set NumSamplesToWrite to a finite number.

Data Types: double

Methods

info	Characteristic information about baseband file writer
reset	Reset states of baseband file writer object
step	Write baseband signal to file

Common to All System Objects	
release	Allow System object property value changes

Examples

Write Baseband Signal to File

Create a baseband file writer object having a sample rate of 1 kHz and a 0 Hz center frequency.

```
bbw = comm.BasebandFileWriter('baseband_data.bb',1000,0);
```

Save today's date in the Metadata structure.

```
bbw.Metadata = struct('Date',date);
```

Generate two channels of QPSK-modulated data.

```
d = randi([0 3],1000,2);
x = pskmod(d,4,pi/4,'gray');
```

Write the baseband data to file `'baseband_data.bb'`.

```
bbw(x)
```

Display information about `bbw`. Release the object.

```
info(bbw)
```

```
ans = struct with fields:
              Filename: 'C:\TEMP\Bdoc20b_1465442_5924\ib8F4264\6\tp07b81c5a\comm-ex66490302\baseba
      SamplesPerFrame: 1000
          NumChannels: 2
             DataType: 'double'
    NumSamplesWritten: 1000
```

```
release(bbw)
```

Create a baseband file reader object to read the saved data. Read the metadata from the file.

```
bbr = comm.BasebandFileReader('baseband_data.bb','SamplesPerFrame',100);
bbr.Metadata
```

```
ans = struct with fields:
    Date: '25-Aug-2020'
```

Read the data from the file.

```
z = [];

while ~isDone(bbr)
    y = bbr();
    z = cat(1,z,y);
end
```

Display information about `bbr`. Release `bbr`.

```
info(bbr)
```

```
ans = struct with fields:
    NumSamplesInData: 1000
            DataType: 'double'
      NumSamplesRead: 1000
```

```
release(bbr)
```

Confirm the original modulated data, `x`, matches the data read from file `'baseband_data.bb'`, `z`.

```
isequal(x,z)
```

```
ans = logical
   1
```

Tips

- `comm.BasebandFileWriter` writes baseband signals to uncompressed binary files. To share these files, you can compress them to a zip file using the `zip` function. For more information, see “Create and Extract from Zip Archives”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BasebandFileReader`

Introduced in R2016b

info

Characteristic information about baseband file writer

Syntax

```
s = info(bbw)
```

Description

`s = info(bbw)` returns a structure, `s`, containing characteristic information for the `BasebandFileWriter` System object, `bbw`. `s` has these fields:

- `Filename` is the name of the baseband data file, returned as a character vector. The filename shows the absolute path.
- `SamplesPerFrame` is the number of samples in each frame, returned as a positive integer.
- `NumChannels` is the number of channels, returned as a positive integer greater than or equal to 1.
- `DataType` is the input data type.
- `NumSamplesWritten` is the number of samples written to the file, returned as a positive integer. This field returns the smaller of the total number of samples processed by the object and the `NumSamplesWritten` property.

Note All fields are available when the object is locked. When the object is unlocked, only the `Filename` and `NumSamplesWritten` fields are available.

reset

System object: `comm.BasebandFileWriter`
Package: `comm`

Reset states of baseband file writer object

Syntax

```
reset(bbw)
```

Description

`reset(bbw)` resets the states of the `BasebandFileWriter` object, `bbw`.

Introduced in R2016b

step

System object: `comm.BasebandFileWriter`
Package: `comm`

Write baseband signal to file

Syntax

```
step(bbw,x)
bbw(x)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`step(bbw,x)` writes a baseband signal, `x`, to the file specified by the `Filename` property of the `BasebandFileWriter` object, `bbw`. The number of samples written to the file is determined by the `NumSamplesToWrite` property of `bbw`.

`bbw(x)` is equivalent to the first syntax.

Note `bbw` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

comm.BCHDecoder

Package: comm

Decode data using BCH decoder

Description

The BCHDecoder object recovers a binary message vector from a binary BCH codeword vector. For proper decoding, the codeword and message length values in this object must match the properties in the corresponding comm.BCHEncoder System object.

To decode a binary message from a BCH codeword:

1. Define and set up your BCH decoder object. See “Construction” on page 3-88.
2. Call step to recover a binary message vector from a binary BCH codeword vector according to the properties of comm.BCHDecoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

dec = comm.BCHDecoder creates a BCH decoder System object, dec, that performs BCH decoding.

dec = comm.BCHDecoder(N,K) creates a BCH decoder object, dec, with the CodewordLength property set to N and the MessageLength property set to K.

dec = comm.BCHDecoder(N,K,GP) creates a BCH decoder object, dec, with the CodewordLength property set to N, the MessageLength property set to K, and the GeneratorPolynomial property set to GP.

dec = comm.BCHDecoder(N,K,GP,S) creates a BCH decoder object, dec, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, and the ShortMessageLength property set to S.

dec = comm.BCHDecoder(N,K,GP,S,Name,Value) creates a BCH decoder object, dec, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, the ShortMessageLength property set to S, and each specified property Name set to the specified Value.

dec = comm.BCHDecoder(Name,Value) creates a BCH decoder object, dec, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

CodewordLength

Codeword length

Specify the codeword length of the BCH code as a double-precision positive integer scalar. The default is 15. The values of the CodewordLength and MessageLength on page 3-0 properties must produce a valid narrow-sense BCH code. For a full-length BCH code, the value of this property must take the form $2^M - 1$, where M is an integer such that $3 \le M \le 16$. The default is 15.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar. The values of the CodewordLength on page 3-0 and MessageLength properties must produce a valid narrow-sense BCH code. The default is 5.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as either Auto or Property. When this property is set to Auto, the BCH code is defined by the CodewordLength on page 3-0 , MessageLength on page 3-0 , GeneratorPolynomial on page 3-0 , and PrimitivePolynomial on page 3-0 properties. When ShortMessageLengthSource is set to Property, you must specify the ShortMessageLength on page 3-0 property, which is used with the other properties to define the BCH code. The default is Auto.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to MessageLength on page 3-0 . When ShortMessageLength < MessageLength, the BCH code is shortened. The default is 5.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as either Auto or Property. Set this property to Auto to create the generator polynomial automatically. Set GeneratorPolynomialSource to Property to specify a generator polynomial using the GeneratorPolynomial on page 3-0 property. The default is Auto.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial as a binary double-precision row vector, a binary Galois field row vector that represents the coefficients of the generator polynomial in order of descending powers, or as a polynomial character vector. The length of the generator polynomial requires a value of CodewordLength on page 3-0 –MessageLength on page 3-0 +1. This property applies when you set GeneratorPolynomialSource on page 3-0 to Property. The default is 'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1', which is the result of bchgenpoly(15,5,[],'double') and corresponds to a 15,5 code.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to true to perform a generator polynomial check the first time you call the step method. The default is true. This check verifies that the specified generator polynomial is valid. For larger codes, disabling the check reduces processing time. As a best practice, perform the check at least once before setting this property to false. This property applies when you set GeneratorPolynomialSource on page 3-0 to Property. The default is true.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as Auto or Property. Set this property to Auto to create a primitive polynomial of degree *M*=ceil(log2(CodewordLength on page 3-0 +1)). Set PrimitivePolynomialSource to Property to specify a polynomial using the PrimitivePolynomial on page 3-0 property. The default is Auto.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial of order M, that defines the finite Galois field GF(2). Use a double-precision, binary row vector with the coefficients of the polynomial in order of descending powers or a polynomial character vector. This property applies when you set the PrimitivePolynomialSource on page 3-0 property to Property. The default is 'X^4 + X + 1', which is the result of fliplr(de2bi(primpoly(4))).

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as None or Property. Set this property to None to disable puncturing. Set it to Property to decode punctured codewords. This decoding is based on a puncture pattern vector you specify in the PuncturePattern on page 3-0 property. The default is None.

PuncturePattern

Puncture pattern vector

Specify the pattern that the object uses to puncture the encoded data. Use a double-precision binary column vector of length CodewordLength on page 3-0 –MessageLength on page 3-0 . Zeros in the puncture pattern vector indicate the position of the parity bits that the object punctures or excludes from each codeword. This property applies when you set PuncturePatternSource on page 3-0 to Property. The default is [ones(8,1); zeros(2,1)].

ErasuresInputPort

Enable erasures input

Set this property to true to specify a vector of erasures as a step method input. The erasures vector is a double-precision or logical binary column vector that indicates which bits of the input codewords to erase or ignore. Values of 1 in the erasures vector correspond to erased bits in the same position of the (possibly punctured) input codewords. Set this property to false to disable erasures. The default is false.

NumCorrectedErrorsOutputPort

Output number of corrected errors

Set this property to true so that the step method outputs the number of corrected errors. The default is true.

Input and Output Signal Lengths in BCH and RS System Objects

The notation y = c * x denotes that y is an integer multiple of x.

The number of punctures equals the number of zeros in the puncture vector.

M is the degree of the primitive polynomial. Each group of *M* bits represents an integer between 0 and 2^M–1 that belongs to the finite Galois field GF(2^M).

ShortMessageLengthSource	comm.BCHEncoder comm.RSEncoder (BitInput = false)	comm.BCHDecoder comm.RSDecoder (BitInput = false)	comm.RSEncoder (BitInput = true)	comm.RSDecoder (BitInput = true)
Auto	**Input Length:** c * MessageLength **Output Length:** c * (CodewordLength – number of punctures)	**Input Length:** c * (CodewordLength – number of punctures) **Output Length:** c * MessageLength **Erasures Length:** c * (CodewordLength – number of punctures)	**Input Length:** c * (MessageLength * M) **Output Length:** c * ((CodewordLength – number of punctures) * M)	**Input Length:** c * ((CodewordLength – number of punctures) * M) **Output Length:** c * (MessageLength * M) **Erasures Length:** c * (CodewordLength – number of punctures)

ShortMessageLengthSource	**comm.BCHEncoder** **comm.RSEncoder (BitInput = false)**	**comm.BCHDecoder** **comm.RSDecoder (BitInput = false)**	**comm.RSEncoder (BitInput = true)**	**comm.RSDecoder (BitInput = true)**
Property	**Input Length:** c * ShortMessageLength **Output Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)	**Input Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures) **Output Length:** c * ShortMessageLength **Erasures Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)	**Input Length:** c * (ShortMessageLength * M) **Output Length:** c * ((CodewordLength - MessageLength + ShortMessageLength - number of punctures) * M)	**Input Length:** c * ((CodewordLength - MessageLength + ShortMessageLength - number of punctures) * M) **Output Length:** c * (ShortMessageLength * M) **Erasures Length:** c * (CodewordLength - MessageLength + ShortMessageLength - number of punctures)

Methods

step	Decode data using a BCH decoder

Common to All System Objects	
release	Allow System object property value changes

Examples

Transmit and decode an 8-DPSK-modulated signal, then count errors

```
% The following code transmits a BCH-encoded, 8-DPSK-modulated bit stream
% through an AWGN channel.  Then, the example demodulates, decodes, and counts errors.
```

```
enc = comm.BCHEncoder;
mod = comm.DPSKModulator('BitInput',true);
chan = comm.AWGNChannel(...
         'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
demod = comm.DPSKDemodulator('BitOutput',true);
dec = comm.BCHDecoder;
errorRate = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(enc, data);
  modSignal = step(mod, encodedData);
  receivedSignal = step(chan, modSignal);
  demodSignal = step(demod, receivedSignal);
  receivedBits = step(dec, demodSignal);
  errorStats = step(errorRate, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.015075
Number of errors = 9
```

Transmit and receive a BPSK-modulated signal

Transmit and receive a BPSK-modulated signal encoded with a shortened BCH code, then count errors.

Specify the codeword, message, and shortened message lengths.

```
N = 255;
K = 239;
S = 63;
```

Create a BCH (255,239) generator polynomial. Use the generator polynomial to create a BCH encoder and decoder pair. The BCH code is based on the AMR standard.

```
gp = bchgenpoly(255,239);
bchEncoder = comm.BCHEncoder(N,K,gp,S);
bchDecoder = comm.BCHDecoder(N,K,gp,S);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Main processing loop.

```
for counter = 1:20
  data = randi([0 1],630,1);                % Generate binary data
  encodedData = bchEncoder(data);           % BCH encode data
  modSignal = pskmod(encodedData,2);        % BPSK modulate
  receivedSignal = awgn(modSignal,5);       % Pass through AWGN channel
  demodSignal = pskdemod(receivedSignal,2); % BSPK demodulate
  receivedBits = bchDecoder(demodSignal);   % BCH decode data
  errorStats = errorRate(data,receivedBits); % Compute error statistics
end
```

Display the error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.000318
Number of errors = 4
```

Shorten a BCH Code

Shorten a (31,26) BCH code to an (11,6) BCH code and use it to encode and decode random binary data.

Create a BCH encoder and decoder pair for a (31,26) code. Specify the generator polynomial, $x^5 + x^2 + 1$, and a shortened message length of 6.

```
enc = comm.BCHEncoder(31,26,'x5+x2+1',6);
dec = comm.BCHDecoder(31,26,'x5+x2+1',6);
```

Encode and decode random binary data and verify that the decoded bit stream matches the original data.

```
x = randi([0 1],60,1);
y = step(enc,x);
z = step(dec,y);
isequal(x,z)
```

```
ans = logical
   1
```

Selected Bibliography

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

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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`bchdec` | `bchgenpoly` | `comm.BCHEncoder` | `comm.RSDecoder` | `primpoly`

step

System object: `comm.BCHDecoder`
Package: `comm`

Decode data using a BCH decoder

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
Y = step(H,X,ERASURES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` decodes input binary codewords in `X` using a (`CodewordLength`,`MessageLength`) BCH decoder with the corresponding narrow-sense generator polynomial. The step method returns the estimated message in `Y`. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to false. The input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91.

`[Y,ERR] = step(H,X)` returns the number of corrected errors in output `ERR` when you set the `NumCorrectedErrorsOutputPort` property to `true`. A non- negative value in the *i*-th element of the `ERR` output vector denotes the number of corrected errors in the *i*-th input codeword. A value of `-1` in the *i*-th element of the `ERR` output indicates that a decoding error occurred for the *i*-th input codeword. A decoding error occurs when an input codeword has more errors than the error correction capability of the BCH code.

`Y = step(H,X,ERASURES)` uses `ERASURES` as the erasures pattern input when you set the `ErasuresInputPort` property to `true`. The object decodes the binary encoded data input, `X`, and treats as erasures the bits of the input codewords specified by the binary column vector, `ERASURES`. The length of `ERASURES` must equal the length of `X`, and its elements must be of data type double or logical. Values of `1` in the erasures vector correspond to erased bits in the same position of the (possibly punctured) input codewords.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.BCHEncoder

Package: `comm`

Encode data using BCH encoder

Description

The `BCHEncoder` object creates a BCH code with specified message and codeword lengths.

To encode data using a BCH coding scheme:

1. Define and set up your BCH encoder object. See "Construction" on page 3-97.
2. Call `step` to create a BCH code with message and codeword lengths specified according to the properties of `comm.BCHEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`enc = comm.BCHEncoder` creates a BCH encoder System object, `enc`, that performs BCH encoding.

`enc = comm.BCHEncoder(N,K)` creates a BCH encoder object, `enc`, with the `CodewordLength` property set to `N` and the `MessageLength` property set to `K`.

`enc = comm.BCHEncoder(N,K,GP)` creates a BCH encoder object, `enc`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K` and the `GeneratorPolynomial` property set to `GP`.

`enc = comm.BCHEncoder(N,K,GP,S)` creates a BCH encoder object, `enc`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, the `GeneratorPolynomial` property set to `GP` and the `ShortMessageLength` property set to `S`.

`enc = comm.BCHEncoder(N,K,GP,S,Name,Value)` creates a BCH encoder object, `enc`, with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, the `GeneratorPolynomial` property set to `GP`, the `ShortMessageLength` property set to `S`, and each specified property `Name` set to the specified `Value`.

`enc = comm.BCHEncoder(Name,Value)` creates a BCH encoder object, `enc`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

Note The input and output signal lengths are listed in "Input and Output Signal Lengths in BCH and RS System Objects" on page 3-91 on the `comm.BCHDecoder` reference page.

CodewordLength

Codeword length

Specify the codeword length of the BCH code as a double-precision positive integer scalar. The default is 15. The values of the CodewordLength and MessageLength on page 3-0 properties must produce a valid narrow-sense BCH code. For a full-length BCH code, the value of the property must use the form $2^M - 1$, where M is an integer such that $3 \leq M \leq 16$. The default is 15.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar. The values of the CodewordLength on page 3-0 and MessageLength properties must produce a valid narrow-sense BCH code. The default is 5.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as either Auto or Property. When this property is set to Auto, the BCH code is defined by the CodewordLength on page 3-0 , MessageLength on page 3-0 , GeneratorPolynomial on page 3-0 , and PrimitivePolynomial on page 3-0 properties. When ShortMessageLengthSource is set to Property, you must specify the ShortMessageLength on page 3-0 property that is used with the other properties to define the RS code. The default is Auto.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to MessageLength on page 3-0 . When ShortMessageLength < MessageLength, the BCH code is shortened. The default is 5.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as either Auto or Property. Set this property to Auto to create the generator polynomial automatically. Set it to Property to specify a generator polynomial using the GeneratorPolynomial on page 3-0 property. The default is Auto.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial as a binary double-precision row vector, a binary Galois row vector that represents the coefficients of the generator polynomial in order of descending powers, or as a polynomial character vector. The length of the generator polynomial requires a value of CodewordLength on page 3-0 -MessageLength on page 3-0 +1. This property applies when you set GeneratorPolynomialSource on page 3-0 to Property. The default is 'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1', which is the result of bchgenpoly(15,5,[],'double') and corresponds to a (15,5) code.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to `true` to perform a generator polynomial check the first time you call the `step` method. This check verifies that the specified generator polynomial is valid. For larger codes, disabling the check reduces processing time. As a best practice, perform the check at least once before setting this property to false. This property applies when you set `GeneratorPolynomialSource` on page 3-0 to `Property`. The default is `true`.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as one of `Auto` or `Property`. Set this property to `Auto` to create a primitive polynomial of degree *M*`=ceil(log2(CodewordLength on page 3-0 +1))`. Set it to `Property` to specify a polynomial using the `PrimitivePolynomial` on page 3-0 property. The default is `Auto`

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial of order M, that defines the finite Galois field GF(2). Use a double-precision, binary row vector with the coefficients of the polynomial in order of descending powers or as a polynomial character vector. This property applies when you set the `PrimitivePolynomialSource` on page 3-0 property to `Property`. The default is `'X^4 + X + 1'`, which is the result of `fliplr(de2bi(primpoly(4)))`.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of `None` or `Property`. Set this property to `None` to disable puncturing. Set it to `Property` to decode punctured codewords. This decoding is based on a puncture pattern vector you specify in the `PuncturePattern` on page 3-0 property. The default is `None`.

PuncturePattern

Puncture pattern vector

Specify the pattern that the object uses to puncture the encoded data. Use a double-precision binary column vector of length `CodewordLength` on page 3-0 `-MessageLength` on page 3-0 . Zeros in the puncture pattern vector indicate the position of the parity bits that the object punctures or excludes from each codeword. This property applies when you set `PuncturePatternSource` on page 3-0 to `Property`. The default is `[ones(8,1); zeros(2,1)]`.

Methods

step	Encode data using a BCH encoder

Common to All System Objects	
release	Allow System object property value changes

Examples

Transmit and decode an 8-DPSK-modulated signal, then count errors

```
% The following code transmits a BCH-encoded, 8-DPSK-modulated bit stream
% through an AWGN channel.  Then, the example demodulates, decodes, and counts errors.

enc = comm.BCHEncoder;
mod = comm.DPSKModulator('BitInput',true);
chan = comm.AWGNChannel(...
         'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
demod = comm.DPSKDemodulator('BitOutput',true);
dec = comm.BCHDecoder;
errorRate = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 1], 30, 1);
  encodedData = step(enc, data);
  modSignal = step(mod, encodedData);
  receivedSignal = step(chan, modSignal);
  demodSignal = step(demod, receivedSignal);
  receivedBits = step(dec, demodSignal);
  errorStats = step(errorRate, data, receivedBits);
end

fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.015075
Number of errors = 9
```

Transmit and receive a BPSK-modulated signal

Transmit and receive a BPSK-modulated signal encoded with a shortened BCH code, then count errors.

Specify the codeword, message, and shortened message lengths.

```
N = 255;
K = 239;
S = 63;
```

Create a BCH (255,239) generator polynomial. Use the generator polynomial to create a BCH encoder and decoder pair. The BCH code is based on the AMR standard.

```
gp = bchgenpoly(255,239);
bchEncoder = comm.BCHEncoder(N,K,gp,S);
bchDecoder = comm.BCHDecoder(N,K,gp,S);
```

Create an error rate counter.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Main processing loop.

```
for counter = 1:20
  data = randi([0 1],630,1);                 % Generate binary data
  encodedData = bchEncoder(data);            % BCH encode data
  modSignal = pskmod(encodedData,2);         % BPSK modulate
  receivedSignal = awgn(modSignal,5);        % Pass through AWGN channel
  demodSignal = pskdemod(receivedSignal,2);  % BSPK demodulate
  receivedBits = bchDecoder(demodSignal);    % BCH decode data
  errorStats = errorRate(data,receivedBits); % Compute error statistics
end
```

Display the error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
  errorStats(1), errorStats(2))
```

```
Error rate = 0.000318
Number of errors = 4
```

Shorten a BCH Code

Shorten a (31,26) BCH code to an (11,6) BCH code and use it to encode and decode random binary data.

Create a BCH encoder and decoder pair for a (31,26) code. Specify the generator polynomial, $x^5 + x^2 + 1$, and a shortened message length of 6.

```
enc = comm.BCHEncoder(31,26,'x5+x2+1',6);
dec = comm.BCHDecoder(31,26,'x5+x2+1',6);
```

Encode and decode random binary data and verify that the decoded bit stream matches the original data.

```
x = randi([0 1],60,1);
y = step(enc,x);
z = step(dec,y);
isequal(x,z)
```

```
ans = logical
   1
```

Selected Bibliography

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

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

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`bchenc` | `bchgenpoly` | `comm.BCHDecoder` | `comm.RSEncoder` | `primpoly`

step

System object: `comm.BCHEncoder`
Package: `comm`

Encode data using a BCH encoder

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` encodes input binary data, `X`, using a (`CodewordLength`,`MessageLength`) BCH encoder with the corresponding narrow-sense generator polynomial and returns the result in vector `Y`. The input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.BitToInteger

Package: comm

(To be removed) Convert vector of bits to vector of integers

Note will be removed in a future release. Use bi2de instead. For more information, see “Compatibility Considerations”.

Description

The BitToInteger object maps groups of bits in the input vector to integers in the output vector.

To map bits to integers:

1. Define and set up your bit to integer object. See “Construction” on page 3-104.
2. Call step to map groups of bits in the input vector to integers in the output vector according to the properties of comm.BitToInteger. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.BitToInteger creates a bit-to-integer converter System object, H, that maps a vector of bits to a corresponding vector of integer values.

H = comm.BitToInteger(Name,Value) creates a bit-to-integer converter object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.BitToInteger(NUMBITS,Name,Value) creates a bit-to-integer converter System object, H This object has the BitsPerInteger on page 3-0 property set to NUMBITS and the other specified properties set to the specified values.

Properties

BitsPerInteger

Number of bits per integer

Specify the number of input bits that the object maps to each output integer. You can set this property to a scalar integer between 1 and 32. The default is 3.

MSBFirst

Assume first bit of input bit words is most significant bit

Set this property to true to indicate that the first bit of the input bit words is the most significant bit (MSB). The default is `true`. You can set this property to `false` to indicate that the first bit of the input bit words is the least significant bit (LSB).

SignedIntegerOutput

Output signed integers

Set this property to true to generate signed integer outputs. The default is `false`. You can set this property to `false` to generate unsigned integer outputs.

When you set this property to `false`, the output values are integers between `0` and (2^{N}) –1. In this case, *N* is the value you specified in the `BitsPerInteger` on page 3-0 property.

When you set this property to `true`, the output values are integers between -($2^{(N-1)}$) and ($2^{(N -1)}$) –1.

OutputDataType

Data type of output

Specify the output data type. The default is `Full precision`.

When you set the `SignedIntegerOutput` on page 3-0 property to `false`, set this property as one of `Full precision` | `Smallest integer` | `Same as input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`.

When you set this property to `Same as input`, and the input data type is numeric or fixed-point (fi object), the output data has the same type as the input data.

When the input signal is an integer data type, you must have a Fixed-Point Designer™ user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When you set the `SignedIntegerOutput` property to `true`, specify the output data type as one of `Full precision` | `Smallest integer` | `double` | `single` | `int8` | `int16` | `int32`.

When you set this property to `Full precision`, the object determines the output data type based on the input data type. If the input data type is double or single precision, the output data has the same type as the input data. Otherwise, the property determines the output data type in the same way as when you set this property to `Smallest unsigned integer`.

Methods

step	(To be removed) Convert vector of bits to vector of integers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

```
hBitToInt = comm.BitToInteger(4);
```

Generate three 4 bit words.

```
bitData = randi([0 1],3*hBitToInt.BitsPerInteger,1);
intData = step(hBitToInt,bitData)

intData = 3×1

    13
     9
    13
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Bit To Integer Converter block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.BitToInteger` will be removed in a future release. Use `bi2de` instead.

Warns starting in R2020b

Use `bi2de` instead of comm.BitToInteger. Data types as supported by comm.BitToInteger are not inherently supported by functions. This code shows binary to decimal conversion for various data types using this function.

Examples Using bi2de for Various Data Types

```
function compbi2de

% Number of integers
n = randi([1 100]);

% Default
h1 = comm.BitToInteger;
bpi = h1.BitsPerInteger;
x = randi([0,1],n*bpi,1);
y1 = h1(x);
y2 = bi2de(reshape(x,bpi,[])','left-msb');
isequal(y1,y2)

% Right MSB, logical input
h2 = comm.BitToInteger( ...
    'BitsPerInteger',5, ...
    'MSBFirst',false);
bpi = h2.BitsPerInteger;
x = logical(randi([0,1],n*bpi,1));
y1 = h2(x);
y2 = bi2de(reshape(x,bpi,[])','right-msb');
isequal(y1,y2)

% Right MSB, signed output, single input
h3 = comm.BitToInteger( ...
    'BitsPerInteger',8, ...
    'MSBFirst',false, ...
    'SignedIntegerOutput',true);
bpi = h3.BitsPerInteger;
x = randi([0,1],n*bpi,1,'single');
y1 = h3(x);
y2 = bi2de(reshape(x,bpi,[])','right-msb');
N = 2^bpi;
y2 = y2 - (y2>=N/2)*N;
isequal(y1,y2)
end
```

See Also

`bi2de` | `bin2dec` | `de2bi`

Introduced in R2012a

step

System object: `comm.BitToInteger`
Package: `comm`

(To be removed) Convert vector of bits to vector of integers

Note `comm.BitToInteger` will be removed in a future release. Use `bi2de` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` converts binary input, `X`, to corresponding integers, `Y`. The input must be a scalar or a column vector and the data type can be numeric, `numerictype(0,1)`, or logical. The length of input `X` must be an integer multiple of the value you specify in the BitsPerInteger property. The object outputs a column vector with a length equal to length(`X`)/`BitsPerInteger`. When you set the SignedIntegerOutput property to `false`, the object maps each group of bits to an integer between `0` and ($2^{\text{BitsPerInteger}}$)-1. A group of bits contains *N* bits, where *N* is the value of the `BitsPerInteger` property. If you set the `SignedIntegerOutput` property to true, the object maps each group of BitsPerInteger bits to an integer between -($2^{(\text{BitsPerInteger-1})}$) and ($2^{(\text{BitsPerInteger-1})}$)-1.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.BinarySymmetricChannel

Package: `comm`

(Removed) Introduce binary errors

Note has been removed. Use `bsc` instead. For more information, see “Compatibility Considerations”.

Description

The `BinarySymmetricChannel` object introduces binary errors to the signal transmitted through this channel.

To introduce binary errors into the transmitted signal:

1. Define and set up your binary symmetric channel object. See “Construction” on page 3-109.
2. Call `step` to introduce binary errors into the signal transmitted through this channel according to the properties of `comm.ACPR`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.BinarySymmetricChannel` creates a binary symmetric channel System object, `H`, that introduces binary errors to the input signal with a prescribed probability.

`H = comm.BinarySymmetricChannel(Name,Value)` creates a binary symmetric channel object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`ErrorProbability`

Probability of binary error

Specify the probability of a binary error as a scalar with a value between `0` and `1`. The default is `0.05`.

`ErrorVectorOutputPort`

Enable error vector output

When you set this property to `true`, the `step` method outputs an error signal, `ERR`. This error signal, in vector form, indicates where errors were introduced in the input signal, `X`. A value of `1` at the *i*-th element of ERR indicates that an error was introduced at the *i*-th element of X. Set the property to `false` if you do not want the ERR vector at the output of the `step` method. The default is `true`.

`OutputDataType`

Data type of output

Specify output data type as one of `double` | `logical`. The default is `double`.

Methods

step	(Removed) Introduce binary errors

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Add Errors to Binary Input Signal

```
% Add binary errors with a probability of 0.2 to a binary input signal.
p = 0.2;
binSymChan = comm.BinarySymmetricChannel('ErrorProbability',p);
data = randi([0 1],1000,1);
[~,err1] = binSymChan(data);

% Confirm that the number errors is approximately
% equal to the probability value multiplied by the number of symbols.
[sum(err1) p*length(data)];

% Perform using the bsc function
[~,err2] = bsc(data,p);
[sum(err2) p*length(data)];
```

```
ans =

   192   200
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Binary Symmetric Channel block reference page. The object properties correspond to the block parameters, except:This object uses the MATLAB default random stream to generate random numbers. The block uses a random number generator based on the V5 RANDN (Ziggurat) algorithm. An initial seed, set with the **Initial seed** parameter initializes the random number generator. For every system run that contains the block, the block generates the same sequence of random numbers. To generate reproducible numbers using this object, you can reset the MATLAB default random stream using the following code.

```
reset(RandStream.getGlobalStream)
```

For more information, see help for `RandStream`.

Compatibility Considerations

`comm.BinarySymmetricChannel` will be removed

Errors starting in R2020b

comm.BinarySymmetricChannel will be removed in a future release. Use bsc instead.

```
% This code provides examples using various syntaxes
% for the comm.BinarySymmetricChannel object versus the |bsc| function.

%% Simulate BSC with default comm.BinarySymmetricChannel object
data = randi([0 1],1000,2); % generate binary data
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel;
s = rng; % Save default MATLAB random number generator state
outOld = bscObj(data);
% Using bsc function
rng(s)  % Restore saved random number generator state
outNew = bsc(data,0.05);

%% Simulate BSC with object and set ErrorProbability property
data = randi([0 1],1000,2); % generate binary data
p = 0.3; % probability
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel('ErrorProbability',p);
s = rng; % Save default MATLAB random number generator state
outOld = bscObj(data);
% Using bsc function
rng(s)  % Restore saved random number generator state
outNew = bsc(data,p);

%% Simulate BSC with object and set ErrorVectorOutputPort property
data = randi([0 1],1000,2); % generate binary data
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel('ErrorVectorOutputPort', true)
s = rng; % Save default MATLAB random number generator state
[outOld,errOld] = bscObj(data);
% Using bsc function
rng(s)  % Restore saved random number generator state
[outNew,errNew] = bsc(data,0.05);

%% Simulate BSC with object and set OutputDataType property to logical
data = randi([0 1],1000,2); % generate binary data
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel('OutputDataType','logical');
s = rng; % Save default MATLAB random number generator state
outOld = bscObj(data);
% Using bsc function
rng(s)  % Restore saved random number generator state
outNew = logical(bsc(data,0.05));

%% Simulate BSC with object and input is logical
data = logical(randi([0 1],1000,2)); % generate logical data
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel;
s = rng; % Save default MATLAB random number generator state.
outOld = bscObj(data);
% Using bsc function
rng(s)  % Restore saved random number generator state
outNew = bsc(double(data),0.05);

%% Simulate BSC with object and input is integer, 'int8'
% Integer datatype options: int8, uint8, int16, uint16, int32, uint32
```

```
data = randi([0 1],1000,2,'int8'); % generate integer data
% Using comm.BinarySymmetricChannel object
bscObj = comm.BinarySymmetricChannel;
s = rng; % Save default MATLAB random number generator state
outOld = bscObj(int8(data));
% Using bsc function
rng(s) % Restore saved random number generator state
outNew = bsc(double(data),0.05);
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`bsc` | `comm.AWGNChannel`

Introduced in R2012a

step

System object: `comm.BinarySymmetricChannel`
Package: `comm`

(Removed) Introduce binary errors

Note `comm.BinarySymmetricChannel` has been removed. Use `bsc` instead. For more information, see “Compatibility Considerations”.

Syntax

```
Y = step(H,X)
[Y,ERR] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` adds binary errors to the input signal `X` and returns the modified signal, `Y`. The input signal can be a vector or matrix with numeric, logical, or fixed-point (fi objects) data type elements. The step method output, `Y`, has the same dimensions as the input, `X`. If `X` input contains a non-binary value, *V*, the object considers it to be 1 when **abs**(*V*) > 0. This syntax applies when you set the `ErrorVectorOutputPort` property to false.

`[Y,ERR] = step(H,X)` returns the error signal vector, `ERR`. A value of `1` at the *i*-th element of `ERR` indicates that an error was introduced at the *i*-th element of `X`. The outputs, `Y` and `ERR`, have the same dimensions as the input, `X`. This syntax applies when you set the `ErrorVectorOutputPort` property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

blePCAPWriter

PCAP or PCAPNG file writer of BLE LL packets

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `blePCAPWriter` object writes generated and recovered Bluetooth low energy (BLE) link layer (LL) packets to a packet capture (PCAP) or packet capture next generation (PCAPNG) file (`.pcap` or `.pcapng`, respectively).

Creation

Syntax

```
obj = blePCAPWriter
obj = blePCAPWriter(Name,Value)
```

Description

`obj = blePCAPWriter` creates a default BLE PCAP or PCAPNG file writer object that writes BLE LL packets to a PCAP or PCAPNG file, respectively.

`obj = blePCAPWriter(Name,Value)` sets “Properties” on page 3-114 using one or more name-value pairs. Enclose each property name in quotes. For example, `('FileExtension','pcapng')` sets the extension of the file as `.pcapng`.

Properties

Note The `blePCAPWriter` object does not overwrite the existing PCAP or PCAPNG file. During each call of this object, specify a unique PCAP or PCAPNG file name.

`FileName` — Name of the PCAP or PCAPNG file
`'bleCapture'` (default) | character row vector | string scalar

Name of the PCAP or PCAPNG file, specified as a character row vector or a string scalar.

Data Types: `char` | `string`

`ByteOrder` — Byte order
`'little-endian'` (default) | `'big-endian'`

Byte order, specified as `'little-endian'` or `'big-endian'`.

Data Types: char | string

FileExtension — Type of file
'pcap' (default) | 'pcapng'

Type of file, specified as 'pcap' or 'pcapng'.

Data Types: char | string

FileComment — Comment for PCAPNG file
'' (default) | character vector | string scalar

Comment for the PCAPNG file, specified as a character vector or a string scalar.

Data Types: char | string

Interface — Name of interface on which BLE packets are captured
'BLE' (default) | character vector | string scalar

Name of the interface on which BLE packets are captured, specified as a character vector or a string scalar.

Data Types: char | string

PhyHeaderPresent — Flag to indicate presence of PHY header
false or 0 (default) | true or 1

Flag to indicate the presence of physical layer (PHY) header, specified as a logical 1 (true) or 0 (false).

Data Types: logical

PCAPWriter — PCAP or PCAPNG file writer object
pcapWriter object | pcapngWriter object

PCAP or PCAPNG file writer object, specified as pcapWriter or pcapngWriter object.

Object Functions

Specific to This Object

write	Write BLE LL protocol packet data to PCAP or PCAPNG file

Examples

Use BLE PCAP Writer to Write LL Packet to PCAP File

Create a BLE PCAP file writer object, specifying the name of the PCAP file.

```
pcapObj = blePCAPWriter('FileName','writeblepacket');
```

Generate a BLE LL packet.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', ...
    'Data (start fragment/complete)');
```

```
payload = '0E00050014010A001F004000170017000000';
llDataPDU = bleLLDataChannelPDU(cfgLLData,payload);
connAccessAddress = de2bi(hex2dec('E213BC42'),32)';
llpacket = [connAccessAddress; llDataPDU];
```

Write the BLE LL packet to the PCAP file.

```
timestamp = 0; % Number of microseconds
write(pcapObj,llpacket,timestamp,'PacketFormat','bits');
```

Use BLE PCAP Writer to Write BLE LL Packet to PCAPNG File

Create a BLE PCAPNG file writer object, specifying the name and extension of the PCAPNG file.

```
pcapObj = blePCAPWriter('FileName','sampleBLELL', ...
    'FileExtension','pcapng');
```

Generate a BLE LL packet.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', ...
    'Data (start fragment/complete)');
payload = '0E00050014010A001F004000170017000000';
llDataPDU = bleLLDataChannelPDU(cfgLLData,payload);
connAccessAddress = de2bi(hex2dec('E213BC42'),32)';
llpacket = [connAccessAddress; llDataPDU];
```

Write the BLE LL packet to the PCAPNG file.

```
timestamp = 12800000; % Number of microseconds
write(pcapObj,llpacket,timestamp,'PacketFormat','bits');
```

Write BLE LL Packet to PCAPNG File from PCAPNG File Writer

Create a PCAPNG file writer object, specifying the name of the PCAPNG file.

```
pcapObj = pcapngWriter('FileName','sampleBLELL',  ...
    'FileComment','This is a sample file');
```

Create a BLE PCAP file writer object, specifying the PCAPNG file writer and the presence of PHY header.

```
blePCAP = blePCAPWriter('PCAPWriter',pcapObj,'PhyHeaderPresent',true);
```

Generate a BLE LL packet.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig;
cfgLLAdv.PDUType = 'Advertising indication';
cfgLLAdv.AdvertisingData = '020106';
llDataPDU = bleLLAdvertisingChannelPDU(cfgLLAdv);
connAccessAddress = de2bi(hex2dec('E213BC42'),32)';
llpacket = [connAccessAddress;llDataPDU];
```

Write the BLE LL packet to the PCAPNG file.

```
PhyHeaderBytes = [39 10 8 1 10 10 10 10 15 00];
timestamp = 18912345; % Number of microseconds
write(blePCAP,llpacket,timestamp,'PacketFormat','bits', ...
    'PhyHeader',PhyHeaderBytes,'PacketComment','This is the first packet');
```

References

[1] Tuexen, M. “PCAP Next Generation (Pcapng) Capture File Format.” 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`pcapWriter` | `pcapngWriter`

Introduced in R2020b

bleAngleEstimateConfig

Configuration object for BLE angle estimation

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleAngleEstimateConfig` object creates a configuration object for Bluetooth low energy (BLE) angle estimation. You can estimate the angle of arrival (AoA) and angle of departure (AoD) by configuring the properties of this object.

Creation

Syntax

```
cfg = bleAngleEstimateConfig
cfg = bleAngleEstimateConfig(Name,Value)
```

Description

`cfg = bleAngleEstimateConfig` creates a default BLE angle estimation configuration object.

`cfg = bleAngleEstimateConfig(Name,Value)` sets “Properties” on page 3-118 by using one or more name-value pairs. Enclose each property name in quotes. For example, `bleAngleEstimateConfig('SlotDuration',1)` sets the switch and sample slot duration to 1 µs.

Properties

`ArraySize` — Size of an antenna array
4 (default) | positive integer | two-element row vector of positive integers

Size of an antenna array, specified as a positive integer or a two-element row vector of positive integers. Setting this property to a positive integer, specifies a uniform linear array (ULA) antenna array design with array elements on the *y*-axis. Setting this property to a vector, specifies a uniform rectangular array (URA) antenna array design. The vector must be of the form [*r c*], where *r* and *c* denote the number of row elements and column elements in the antenna array, respectively. In this case, the row elements and column elements are present along *y*-axis and *z*-axis, respectively.

Data Types: `double`

`ElementSpacing` — Normalized element spacing with respect to signal wavelength
`0.5` (default) | positive real number less than or equal to 0.5 | two-element row vector of positive real numbers less than or equal to 0.5

Normalized element spacing with respect to signal wavelength, specified as a positive real number less than or equal to 0.5 or a two-element row vector of positive real numbers less than or equal to

0.5. Setting this property to a positive real number, specifies a ULA antenna array design with array elements on the *y*-axis. Setting this property to a vector, specifies a URA antenna array design. The vector must be of the form [*sr sc*], where *sr* and *sc* denote the spacing between row and column elements of the antenna array, respectively. In this case, the rows and columns are present along *y*-axis and *z*-axis, respectively.

Data Types: `double`

`SlotDuration` — Switch and sample slot duration

`2` (default) | `1`

Switch and sample slot duration, specified as `1` or `2`. This value must be expressed in microseconds.

Data Types: `double`

`SwitchingPattern` — Antenna switching pattern

`[1 2 3 4]` (default) | *M*-element row vector

Antenna switching pattern, specified as an *M*-element row vector, where *M* must be in the range [2, 74/“SlotDuration” on page 3-0 +1].

Data Types: `double`

Object Functions

Specific to This Object

getElementPosition	Positions of the antenna array elements
getNumElements	Number of elements in antenna array

Examples

Estimate Single Angle Using Four-Element ULA

Create a default BLE angle estimation configuration object.

```
cfg = bleAngleEstimateConfig;
```

Specify a ULA antenna array design by setting the antenna array size of the configuration object to 4.

```
cfg.ArraySize = 4
```

```
cfg = 
  bleAngleEstimateConfig with properties:

            ArraySize: 4
       ElementSpacing: 0.5000
         SlotDuration: 2
     SwitchingPattern: [1 2 3 4]
```

Estimate Two Angles Using 4-by-4 URA

Create a default BLE angle estimation configuration object.

```
cfg = bleAngleEstimateConfig
```

```
cfg = 
  bleAngleEstimateConfig with properties:

           ArraySize: 4
      ElementSpacing: 0.5000
        SlotDuration: 2
    SwitchingPattern: [1 2 3 4]
```

Specify a URA antenna array design by setting the antenna array size of the configuration object to [4 4].

```
cfg.ArraySize = [4 4];
```

Set the row element spacing and column element spacing to `0.4` and `0.3`, respectively.

```
cfg.ElementSpacing = [0.4 0.3];
```

Set the value of antenna switching pattern.

```
cfg.SwitchingPattern = 1:16
```

```
cfg = 
  bleAngleEstimateConfig with properties:

           ArraySize: [4 4]
      ElementSpacing: [0.4000 0.3000]
        SlotDuration: 2
    SwitchingPattern: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Wooley, Martin. *Bluetooth Direction Finding: A Technical Overview*. Bluetooth Special Interest Group (SIG), Accessed April 6, 2020, https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleAngleEstimate` | `bleIdealReceiver` | `bleWaveformGenerator`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”
“Bluetooth Location and Direction Finding”

Introduced in R2020b

bleATTPDUConfig

Configuration object for BLE ATT PDU

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleATTPDUConfig` creates a configuration object for a Bluetooth low energy (BLE) attribute protocol data unit (ATT PDU). You can configure a BLE ATT PDU using the applicable properties of `bleATTPDUConfig`.

Creation

Syntax

```
cfgATT = bleATTPDUConfig
cfgATT = bleATTPDUConfig(Name,Value)
```

Description

`cfgATT = bleATTPDUConfig` creates a `bleATTPDUConfig` configuration object, `cfgATT`, for a BLE ATT PDU with default values.

`cfgATT = bleATTPDUConfig(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `bleATTPDUConfig('Opcode','Error response')` sets the `Opcode` property of `cfgATT` to `'Error response'`.

Properties

Note For more information about BLE ATT PDU properties and their respective values, see volume 3, part F, sections 3.3 and 3.4 of the Bluetooth Core Specification [2].

`Opcode` — BLE ATT PDU operation code

`'Read request'` (default) | `'MTU request'` | `'Information request'` | `...`

BLE ATT PDU operation code, specified as one of the values in this list. Each valid value describes a BLE ATT PDU operation code.

- `'MTU request'`
- `'Error response'`
- `'Information request'`

- 'Find by type value request'
- 'Read by type request'
- 'Read request'
- 'Read response'
- 'Read blob request'
- 'Read blob response'
- 'Read by group type request'
- 'Write request'
- 'Write response'
- 'Write command'
- 'Prepare write request'
- 'Prepare write response'
- 'Execute write request'
- 'Execute write response'
- 'Handle value notification'
- 'Handle value indication'
- 'Handle value confirmation'
- 'Information response'
- 'Find by type value response'
- 'Read by type response'
- 'Read by group type response'

Data Types: char | string

RequestedOpcode — Opcode of request BLE ATT PDU
'Read request' (default) | character vector | string scalar

Opcode of request BLE ATT PDU, specified as one of the values in this list. Each valid value describes a request BLE ATT PDU (from a peer device) that caused an error.

- 'MTU request'
- 'Information request'
- 'Find by type value request'
- 'Read by type request'
- 'Read request'
- 'Read blob request'
- 'Read by group type request'
- 'Write request'
- 'Prepare to write request'
- 'Execute write request'

Data Types: char | string

Format — Format of information data field

'16 bit' (default) | '128 bit'

Format of information data field, specified as '16 bit' or '128 bit'. This value specifies the format of information data element in the PDU with opcode 'Information Response'.

Data Types: char | string

AttributeHandle — Handle value of attribute

'0001' (default) | character vector of two-octet hexadecimal

Handle value of attribute, specified as the character vector of a two-octet hexadecimal value in the range [0x0001, 0xFFFF]. This value is a unique identifier. The server dynamically assigns this value.

Data Types: char | string

ErrorMessage — Error message corresponding to request BLE ATT PDU

'Invalid handle' (default) | 'Invalid handle' | 'Read not permitted' | ...

Error message corresponding to request BLE ATT PDU, specified as one of the values in this list. Each value indicates the cause of an error corresponding to the request PDU from a peer device.

- 'Invalid handle'
- 'Read not permitted'
- 'Write not permitted'
- 'Invalid PDU'
- 'Insufficient authentication'
- 'Request not supported'
- 'Invalid offset'
- 'Insufficient authorization'
- 'Prepare queue full'
- 'Attribute not found'
- 'Attribute not long'
- 'Insufficient encryption key size'
- 'Invalid attribute value length'
- 'Unlikely error'
- 'Insufficient encryption'
- 'Unsupported group type'
- 'Insufficient resources'

Data Types: char | string

MaxTransmissionUnit — Maximum size of BLE ATT PDU

23 (default) | positive integer

Maximum size of BLE ATT PDU, specified as a positive integer in the range [23,65535]. This value sets the maximum size of the BLE ATT PDU in bytes that a client or server can receive.

Data Types: double

`StartHandle` — Starting handle of handle range

`'0001'` (default) | character vector of two-octet hexadecimal value

Starting handle of handle range, specified as a two-octet hexadecimal value in the range [0x0001,0xFFFF]. This value indicates the handle value of a service or characteristic declaration or the starting handle of a handle range. This value must be less than the `EndHandle`.

Data Types: `char` | `string`

`EndHandle` — Ending handle of handle range

`'FFFF'` (default) | character vector of two-octet hexadecimal value

Ending handle of handle range, specified as a two-octet hexadecimal value in the range [0x0001,0xFFFF]. This value sets the end handle value of a service declaration or a characteristic declaration or the ending handle of a handle range. This value must be greater than the `StartHandle`.

Data Types: `char` | `string`

`AttributeType` — Type of attribute

`'2800' (Primary service)` (default) | four-element or 32-element character vector | 2-octet or 16-octet string scalar

Type of attribute, specified as a four-element or 32-element character vector or a string scalar denoting a two-octet or16-octet hexadecimal value.

Data Types: `char` | `string`

`AttributeValue` — Value of attribute

`' '` (default) | character vector | string scalar | numeric vector of elements in the range [0,255] | `n`-by-2 character array of maximum length `131068`

Value of attribute, specified as one of these values:

- Character vector — This vector represent octets in hexadecimal format.
- String scalar — This scalar represent octets in hexadecimal format.
- Numeric vector of elements in the range [0,255] — This vector represent octets in decimal format. The maximum length of the numeric vector is `65534`.
- `n`-by-2 character array — Each row represent an octet in hexadecimal format. The maximum length of the character array is `131068`.

`AttributeValue` indicates the value of an attribute to be stored in or read from the attribute database. Specify this value in LSB first format.

Data Types: `char` | `string` | `double`

`Offset` — Offset of next octet to be read

`0` (default) | integer in the range [0, 65565]

Offset of next octet to be read, specified as an integer in the range `[0, 65535]`. You can use this value in the BLE ATT PDUs with opcodes `'Read blob request'`, `'Prepare write request'`, and `'Prepare write response'` to identify an attribute value offset in the attribute database.

Data Types: `double`

ExecuteWrite — Execute write flag

'Cancel all prepared writes' (default) | 'Write all pending requests'

Execute write flag, specified as 'Cancel all prepared writes' or 'Write all pending requests'. You can determine the action (discard or write) to be performed when this property is used.

Data Types: char | string

Object Functions

Specific to This Object

bleATTPDU	Generate BLE ATT PDU
bleATTPDUDecode	Decode BLE ATT PDU

Examples

Create BLE ATT PDU Configuration Objects

Create two unique BLE ATT PDU configuration objects: one of type 'Read by type request' and the other of type 'Error response' using default settings and name-value pairs respectively.

Create a BLE ATT PDU configuration object with default settings.

```
cfgATT = bleATTPDUConfig;
```

Set the BLE ATT PDU opcode as 'Read by type request'. View the applicable properties of the opcode 'Read by type request'.

```
cfgATT.Opcode = 'Read by type request'
```

```
cfgATT = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

Create another BLE ATT PDU configuration object, this time using the name-value pairs. Change the BLE ATT PDU opcode to 'Error response'. View the applicable properties of the opcode 'Error response'.

```
cfgATT = bleATTPDUConfig('Opcode','Error response')
```

```
cfgATT = 
  bleATTPDUConfig with properties:

             Opcode: 'Error response'
    RequestedOpcode: 'Read request'
    AttributeHandle: '0001'
       ErrorMessage: 'Invalid handle'
```

End-to-End Workflow of BLE ATT PDU

Create a BLE ATT PDU configuration object. Change the value of opcode to `'Read by type request'`. View the applicable properties of the specified value of opcode.

```
cfgTx = bleATTPDUConfig;
cfgTx.Opcode = 'Read by type request'
```

```
cfgTx = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

Generate a BLE ATT PDU from the corresponding configuration object.

```
attPDU = bleATTPDU(cfgTx);
```

Decode the generated BLE ATT PDU. The returned status indicates decoding is successful. View the applicable properties of the opcode `'Error response'`.

```
[status, cfgRx] = bleATTPDUDecode(attPDU)
```

```
status = 
Success

cfgRx = 
  bleATTPDUConfig with properties:

           Opcode: 'Read by type request'
      StartHandle: '0001'
        EndHandle: 'FFFF'
    AttributeType: '2800'
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleATTPDU` | `bleATTPDUDecode`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2019b

bleChannelSelection

BLE channel index for connection events

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleChannelSelection` System object selects a Bluetooth low energy (BLE) channel index based on the selected algorithm. For more information, see `Algorithm` in property.

To select a BLE channel index:

1. Create the `bleChannelSelection` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
csa = bleChannelSelection
csa = bleChannelSelection(Name,Value)
```

Description

`csa = bleChannelSelection` creates a `blechannelselection` System object, `csa`, to select a BLE channel index for connection events or periodic advertising events.

`csa = bleChannelSelection(Name,Value)`sets properties using one or more name-value pairs. For example, `bleChannelSelection('Algorithm','2')` configures the System object, `csa`, to select a BLE channel index based on `'Algorithm #2'`. Enclose each property name in quotes.

Properties

Note For more information about BLE channel selection properties and their respective values, see volume 6, part B, section 4.5.8 of the Bluetooth Core Specification [2].

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Algorithm` — Type of BLE channel selection algorithm

`1` (default) | `2`

Type of BLE channel selection algorithm, specified as `1` or `2` representing `'Algorithm #1'` or `'Algorithm #2'` respectively. The property uses this algorithm to select the channel index.

Data Types: `double`

`HopIncrement` — Hop increment count

`5` (default) | integer in the range [5, 16]

Hop increment count, specified as an integer in the range [5,16]. This property indicates the hop increment count to be used for hopping between data channels. `'Algorithm #1'` uses this value as an input.

Data Types: `double`

`AccessAddress` — Unique connection address

`'8E89BED6'` (default) | eight-element character vector | string scalar denoting a four-octet hexadecimal value

Unique connection address, specified as an eight-element character vector or a string scalar denoting a four-octet hexadecimal value. This value indicates a unique 32-bit address for the link layer connection between two devices. `'Algorithm #2'` uses this value as an input.

Data Types: `char` | `string`

`UsedChannels` — List of used (good) data channels

row vector containing all the channel indices [0, 36] (default) | vector of integers in the range [0, 36]

List of used (good) data channels, specified as an integer vector with element values in the range [0, 36]. The vector length must be greater than 1. At least two channels must be set as used (good) channels. This value indicates the set of good channels classified by the master.

Data Types: `double`

Usage

Syntax

```
channelIndex = csa()
```

Description

`channelIndex = csa()` selects a channel index based on the algorithm specified by the `Algorithm` property, the list of used data channels specified by the `UsedChannels` property, and other applicable properties dependent on the selected algorithm. The returned channel index, `channelIndex`, is of data type double.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `bleChannelSelection`

clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Select BLE Channel Indices Based on Type of Channel Selection Algorithm

Create a `blechannelselection` System object, `'csa'`, to select a BLE channel index based on the channel selection algorithm, `'Algorithm #1'`. View the corresponding applicable properties.

```
csa = bleChannelSelection
```

```
csa = 
  bleChannelSelection with properties:

       Algorithm: 1
    HopIncrement: 5
    UsedChannels: [1x37 double]
    ChannelIndex: 0
    EventCounter: 0
```

Set the values of `'HopIncrement'` to `7` and `'UsedChannels'` to `[0,2,24,6,10,14,26,30,34,36]`. View the corresponding properties.

```
csa.HopIncrement = 7;
csa.UsedChannels = [0, 2, 24, 6, 10, 14, 26, 30, 34, 36]
```

```
csa = 
  bleChannelSelection with properties:

       Algorithm: 1
    HopIncrement: 7
    UsedChannels: [0 2 6 10 14 24 26 30 34 36]
    ChannelIndex: 0
    EventCounter: 0
```

Select a BLE channel index from the corresponding system object using `'Algorithm #1'`.

```
channelIndex = csa()
```

```
channelIndex = 30
```

Create another blechannelselection System object, 'csa', this time to select a BLE channel index by specifying the type of channel selection algorithm as 'Algorithm #2'. View the corresponding applicable properties.

```
csa2 = bleChannelSelection("Algorithm",2);
```

Change the values of 'AccessAddress' to '8E89BED6' and 'UsedChannels' to [9,10,21,22,23,33,34,35,36]. View the corresponding properties.

```
csa2.AccessAddress = '8E89BED6';
csa2.UsedChannels = [9, 10, 21, 22, 23, 33, 34, 35, 36]
```

```
csa2 = 
  bleChannelSelection with properties:

        Algorithm: 2
    AccessAddress: '8E89BED6'
     UsedChannels: [9 10 21 22 23 33 34 35 36]
     ChannelIndex: 0
     EventCounter: 0
```

Select a BLE channel index from the corresponding system object using 'Algorithm #2'.

```
channelIndex2 = csa2()
```

```
channelIndex2 = 9
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
bleLLDataChannelPDU | bleLLDataChannelPDUDecode

Objects
bleLLDataChannelPDUConfig | bleLLControlPDUConfig

Topics
“BLE Channel Selection Algorithms”
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLControlPDUConfig

Configuration object for BLE LL control PDU payload configuration

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleLLControlPDUConfig` creates a configuration object for Bluetooth low energy (BLE) link layer (LL) control protocol data unit (PDU) payload configuration by using the default and specified values. You can configure a BLE LL control PDU payload configuration using the applicable properties of `bleLLControlPDUConfig`.

Creation

Syntax

```
cfgControl = bleLLControlPDUConfig
cfgControl = bleLLControlPDUConfig(Name,Value)
```

Description

`cfgControl = bleLLControlPDUConfig` creates a configuration object, `cfgControl`, for a BLE LL control PDU payload configuration using default values.

`cfgControl = bleLLControlPDUConfig(Name,Value)`sets the properties using one or more name-value pairs. Enclose each property name in quotes. For example, `bleLLControlPDUConfig('Opcode','Version indication')` configures `cfgControl` with the operation code as `'Version indication'`

Properties

Note For more information about BLE LL control PDU properties and their respective values, see volume 6, part B, section 2.4 of the Bluetooth Core Specification [2].

Opcode — BLE LL control PDU payload configuration operation code
`'Connection update indication'` (default) | `'Channel map indication'` | `'Terminate indication'` | `'Unknown response'` | `'Version indication'` | `'Reject indication'`

BLE LL control PDU payload configuration operation code, specified as one of the values in this list. Each valid value describes a BLE LL control PDU operation code.

- `'Connection update indication'`

- 'Channel map indication'
- 'Terminate indication'
- 'Unknown response'
- 'Version indication'
- 'Reject indication'

Data Types: char | string

WindowSize — Transmit window size
1 (default) | nonnegative integer

Transmit window size, specified as a nonnegative integer in the range [1, Mws], where Mws is the lesser of 8 and (ConnectionInterval-1). This property indicates the window size within which the master transmits a data packet and the slave listens for a data packet after the connection is established. Each unit is taken as 1.25 ms so that the window size (WindowSize×1.25) is in the range of 1.25 ms to the minimum of (10 ms, ((ConnectionInterval×1.25) - 1.25) ms).

Data Types: double

ConnectionInterval — Connection interval
6 (7.5 ms) (default) | integer in the range [6, 3200]

Connection interval, specified as an integer in the range [6, 3200]. This property indicates the interval between the start of two consecutive connection events. Each unit is taken as 1.25 ms so that the connection interval (ConnectionInterval×1.25) is in the range of 7.5 ms to 4.0 s.

Data Types: double

SlaveLatency — Slave latency
0 (default) | nonnegative integer

Slave latency, specified as a nonnegative integer in the range [0, Msl], where Msl is the lesser of 499 and ((ConnectionTimeout×10)/((ConnectionInterval×1.25)×2))-1. This property indicates the number of connection events that a slave can ignore.

Data Types: double

ConnectionTimeout — Connection supervision timeout
10 (default) | nonnegative integer

Connection supervision timeout, specified as a nonnegative integer in the range [Mct, 3200], where Mct is the larger of 10 and ((1+SlaveLatency)*(ConnectionInterval×1.25)×2)/10. If the slave does not receive a valid packet within this time, this property indicates the connection timeout. Each unit is taken as 10 ms so that the connection timeout (ConnectionInterval×10) is in the range of 100 ms to 32.0 s.

Data Types: double

Instant — Connection event instant
0 (default) | integer in the range [0, 65535]

Connection event instant, specified as an integer in the range [0, 65535]. This property indicates the event count at which specific action must occur, for instance, using updated connection parameters.

Data Types: double

`UsedChannels` — List of used data channels

`row vector containing the channel indices between [0:36]` (default) | integer vector with element values in the range [0, 36]

List of used data channels, specified as an integer vector with element values in the range [0, 36]. The vector length must be greater than 1. At least two channels must be set as used (good) channels. This property indicates the set of good channels classified by the master.

Data Types: `double`

`ErrorCode` — Connection termination error code

`'Success'` (default) | `'Unknown connection identifier'` | `'Hardware failure'` | `...`

Connection termination error code, specified as one of the values in this list. Each valid value describes the error description informing the remote device why the connection is about to be terminated.

- `'Success'`
- `'Unknown connection identifier'`
- `'Hardware failure'`
- `'Memory capacity exceeded'`
- `'Connection timeout'`
- `'Connection limit exceeded'`
- `'Connection already exists'`
- `'Command disallowed'`
- `'Connection accept timeout exceeded'`
- `'Connection rejected due to limited resources'`
- `'Invalid LL parameters'`
- `'Connection rejected due to unacceptable BD_ADDR'`
- `'Unspecified error'`
- `'Unsupported LL parameter value'`
- `'Role change not allowed'`
- `'LL response timeout'`
- `'LL procedure collision'`
- `'Instant passed'`
- `'Channel classification not supported'`
- `'Extended inquiry response too large'`
- `'Connection rejected due to no suitable channel found'`
- `'Advertising timeout'`
- `'Controller busy'`
- `'Unacceptable connection parameters'`
- `'Connection failed to be established'`
- `'Unknown advertising identifier'`
- `'Limit reached'`

- `'Operation cancelled by host'`

Data Types: `char` | `string`

`UnknownOpcode` — Unrecognized or unsupported operation code
`'00'` (default) | two-element character vector | string scalar denoting one-octet hexadecimal value

Unrecognized or unsupported operation code, specified as the comma-separated pair consisting of `'UnknownOpcode'` and a two-element character vector or string scalar denoting a one-octet hexadecimal value. This property indicates the type of BLE LL control PDU that is not recognized or supported.

Data Types: `char` | `string`

`VersionNumber` — Version number of Bluetooth Core Specification
`'5.0'` (default) | `'4.0'` | `'4.1'` | `'4.2'`

Version number of Bluetooth Core Specification, specified as `'4.0'`, `'4.1'`, `'4.2'` or `'5.0'`. This property indicates the version number of the Bluetooth Core Specification.

Data Types: `string`

`CompanyIdentifier` — Manufacturer ID of Bluetooth controller
`'FFFF'` (default) | four-element character vector | string scalar denoting a two-octet hexadecimal value

Manufacturer ID of Bluetooth controller, specified as a four-element character vector or a string scalar denoting a two-octet hexadecimal value. This property indicates the unique identifier assigned to your organization by Bluetooth Special Interest Group (SIG) [2].

Data Types: `char` | `string`

`SubVersionNumber` — Subversion number of the Bluetooth controller
`'0000'` (default) | four-element character vector | string scalar denoting a two-octet hexadecimal value

Subversion number of the Bluetooth controller, specified as a four-element character vector or string scalar denoting a two-octet hexadecimal value. This property indicates the unique value for each implementation or revision of a Bluetooth controller implementation.

Data Types: `char` | `string`

Object Functions

Specific to This Object

bleLLDataChannelPDU	Generate BLE LL data channel PDU
bleLLDataChannelPDUDecode	Decode BLE LL data channel PDU
bleLLDataChannelPDUConfig	Configuration object for BLE LL data channel PDU

Examples

Create BLE LL Control PDU Configuration Object Using Default Settings

Create a BLE LL control PDU configuration object, `'cfgControl'`, using default settings. View the corresponding applicable properties.

```
cfgControl = bleLLControlPDUConfig
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

                Opcode: 'Connection update indication'
            WindowSize: 1
          WindowOffset: 0
    ConnectionInterval: 6
          SlaveLatency: 0
     ConnectionTimeout: 10
               Instant: 0
```

Change the value of the connection interval to `64`. View the configured properties.

```
cfgControl.ConnectionInterval = 64
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

                Opcode: 'Connection update indication'
            WindowSize: 1
          WindowOffset: 0
    ConnectionInterval: 64
          SlaveLatency: 0
     ConnectionTimeout: 10
               Instant: 0
```

Create BLE LL Control PDU Configuration Object Using Name-Value Pairs

Create two unique BLE LL control PDU configuration objects using name-value pairs: first one of the type `'Terminate indication'` and error code `'Connection timeout'` and second one of the type `'Channel map indication'` with used set of data channels.

Create a BLE LL control PDU configuration object, `'cfgControl'`, by specifying opcode as `'Terminate indication'` and the error code as `'Connection timeout'`. View the applicable properties corresponding to the specified opcode.

```
cfgControl = bleLLControlPDUConfig('Opcode','Terminate indication', ...
    'ErrorCode','Connection Timeout')
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

       Opcode: 'Terminate indication'
    ErrorCode: 'Connection timeout'
```

Create another BLE LL control PDU payload configuration object, `'cfgControl'`, this time by setting the value of opcode as `'Channel map indication'`. Specify the list of used data channels. View the configured properties corresponding to the specified opcode.

```
cfgControl = bleLLControlPDUConfig('Opcode','Channel map indication');
cfgControl.UsedChannels = [0 3 12 16 18 24]
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

          Opcode: 'Channel map indication'
         Instant: 0
    UsedChannels: [0 3 12 16 18 24]
```

Create a BLE LL data channel configuration object, `'cfgLLData'`, by specifying the values of `'LLID'` as `'Control'` and `'ControlConfig'` as `'cfgControl'`. View the properties of the configuration object `'cfgLLData'`.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID','Control');
cfgLLData.ControlConfig = cfgControl
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Control'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: '012345'
        ControlConfig: [1x1 bleLLControlPDUConfig]
```

End-to-End Workflow of BLE LL Control PDU

Create a BLE LL data channel PDU configuration object for a control PDU by using default configuration. View the corresponding default properties.

```
cfgControl = bleLLControlPDUConfig
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

               Opcode: 'Connection update indication'
           WindowSize: 1
         WindowOffset: 0
   ConnectionInterval: 6
         SlaveLatency: 0
    ConnectionTimeout: 10
              Instant: 0
```

Generate a BLE LL data channel PDU using `'cfgTx'` by specifying the value of link layer identifier, `'LLID'`, as `'Control'` and `'ControlConfig'` as `'cfgControl'`. Specify the cyclic redundancy check (CRC) value for the configuration object `'cfgTx'`. View the properties of `'cfgTx'`.

```
cfgTx = bleLLDataChannelPDUConfig('LLID', 'Control', ...
    'ControlConfig',cfgControl);
cfgTx.CRCInitialization = 'E23456'

cfgTx = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Control'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: 'E23456'
        ControlConfig: [1x1 bleLLControlPDUConfig]
```

Generate a BLE LL control PDU from `'cfgTx'`.

```
pdu = bleLLDataChannelPDU(cfgTx);
```

Decode the generated BLE LL control PDU by initializing the CRC value. The returned status indicates decoding is successful. View the values of `'status'`, `'cfgRx'` and `'llPayload'`.

```
crcInit = 'E23456'; % Received during associaton
[status, cfgRx, llPayload] = bleLLDataChannelPDUDecode(pdu, crcInit)

status = 
Success

cfgRx = 
  bleLLDataChannelPDUConfig with properties:

                 LLID: 'Control'
                 NESN: 0
       SequenceNumber: 0
             MoreData: 0
    CRCInitialization: '012345'
        ControlConfig: [1x1 bleLLControlPDUConfig]


llPayload =

  1x0 empty char array
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleLLDataChannelPDU` | `bleLLDataChannelPDUDecode`

Objects

`bleLLDataChannelPDUConfig`

Topics

"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2019b

bleL2CAPFrameConfig

Configuration object for BLE L2CAP frame

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleL2CAPFrameConfig` creates a configuration object for Bluetooth low energy (BLE) logical link control and adaptation protocol (L2CAP) signaling frame or data frame using the default and specified values. You can configure a BLE L2CAP signaling frame or data frame using the applicable properties of `bleL2CAPFrameConfig`.

Creation

Syntax

```
cfgL2CAP = bleL2CAPFrameConfig
cfgL2CAP = bleL2CAPFrameConfig(Name,Value)
```

Description

`cfgL2CAP = bleL2CAPFrameConfig` creates a configuration object, `cfgL2CAP`, for a BLE L2CAP signaling command frame or data frame with default values.

`cfgL2CAP = bleL2CAPFrameConfig(Name,Value)` specifies properties using one or more name-value pairs. Enclose each property name in quotes. For example, `bleL2CAPFrameConfig('CommandType','Command reject')` configures `cfgL2CAP` with the type of signaling command frame as `'Command reject'`.

Properties

Note For more information about BLE L2CAP frame properties and their respective values, see volume 3, part A, section 3 of the Bluetooth Core Specification [2].

`ChannelIdentifier` — Identifier for logical channel endpoint
`'0005'` (default) | four-element character vector | string scalar denoting a two-octet hexadecimal value

Identifier for a logical channel endpoint, specified as a four-element character vector or a string scalar denoting a two-octet hexadecimal value. The `'ChannelIdentifier'` denotes the local name representing a logical channel endpoint. This property is used to identify the command and data frames. Command frames use `'0005'` as the `'ChannelIdentifier'`. L2CAP B-frames use fixed

'ChannelIdentifier', '0004' for attribute protocol (ATT) and '0006' for security manager protocol (SMP).

Data Types: char | string

CommandType — Signaling command type
'Credit Based Connection request' (default) | 'Command reject' | 'Disconnection request' | ...

Signaling command type, specified as a character vector or a string scalar. You can specify CommandType as one of these values:

- 'Command reject'
- 'Disconnection request'
- 'Disconnection response'
- 'Connection Parameter Update request'
- 'Connection Parameter Update response'
- 'Credit Based Connection request'
- 'Credit Based Connection response'
- 'Flow Control Credit'

This property is applicable only when the value of ChannelIdentifier is set to '0005' (signaling channel identifier).

Data Types: char | string

SignalIdentifier — Identifier for request-response frame exchange
'01' (default) | two-element character vector | string scalar denoting one-octet hexadecimal value

Identifier for a request-response frame exchange, specified as a two-element character vector or string scalar denoting a one-octet hexadecimal value. The requesting device sets the value of this property and the responding device uses the same value in its response. The value of this property cannot be set to '00'.

Data Types: char | string

CommandRejectReason — Reason for rejecting received signaling command frame
'Command not understood' (default) | 'Signaling MTU exceeded' | 'Invalid CID in request'

Reason for rejecting the received signaling command frame, specified as a character vector or a string scalar. You can specify CommandRejectReason as one of these values:

- 'Command not understood'
- 'Signaling MTU exceeded'
- 'Invalid CID in request'

This property specifies the reason for rejecting a signaling command frame.

Data Types: char | string

SourceChannelIdentifier — Source logical channel endpoint
'0040' (default) | four-element character vector | string scalar denoting two-octet hexadecimal value

Source logical channel endpoint, specified as a four-element character vector or string scalar denoting a two-octet hexadecimal value. This property specifies the source channel endpoint from which the request is sent or a response is received. When the channel is created using credit-based connection procedure, data packets flowing to the sender of the request are sent to the `SourceChannelIdentifier`.

Data Types: `char` | `string`

`DestinationChannelIdentifier` — Destination logical channel endpoint

`'0040'` (default) | four-element character vector | string scalar denoting two-octet hexadecimal value

Destination logical channel endpoint, specified as a four-element character vector or string scalar denoting a two-octet hexadecimal value. This property specifies the destination channel endpoint from which the request is sent or a response is received. When the channel is created using credit-based connection procedure, data packets flowing to the destination of the request are sent to the `DestinationChannelIdentifier`.

Data Types: `char` | `string`

`ConnectionIntervalRange` — Connection interval range

`[6,3200]` (default) | two-element numeric vector specified as [*MIN*, *MAX*]

Connection interval range, specified as a two-element numeric vector in the form of [*MIN*, *MAX*]. *MIN* and *MAX* specify the minimum and the maximum value of the `ConnectionIntervalRange` respectively. You can specify the value of *MIN* and *MAX* in the range [6, 3200]. *MIN* must be less than or equal to *MAX*. Each unit for *MIN* or *MAX* is taken as 1.25 ms so the resultant value is in the range [7.5 ms, 4.0 s].

Data Types: `double`

`ConnectionTimeout` — Connection supervision timeout

`10 (100 ms)` (default) | integer in the range [*Mct*, 3200]

Connection supervision timeout, specified as an integer in the range [*Mct*, 3200], where *Mct* is the larger of 10 and ((1+`SlaveLatency`)×(`ConnectionInterval`×1.25)×2)/10. This property indicates the timeout for a connection if no valid packet is received within this time. Each unit is taken as 10 ms so that the resultant connection timeout, (`ConnectionInterval`×10), is in the range [100 ms, 32.0s].

Data Types: `double`

`SlaveLatency` — Number of link layer connection events a slave can ignore

`0` (default) | integer in the range [0, *Msl*]

Number of link layer connection events a slave can ignore, specified as an integer in the range [0, *Msl*], where *Msl* is the lesser of 499 and ((`ConnectionTimeout`×10)/((`ConnectionInterval`×1.25) ×2))-1. This property indicates the number of connection events that a slave can ignore.

Data Types: `double`

`ParameterUpdateResult` — Result of connection parameters update

`'Accepted'` (default) | `'Rejected'`

Result of the connection parameters update, specified as `'Accepted'` or `'Rejected'`. This property indicates the response to the `'Connection Parameter Update Request'` value of the property `CommandType` and specifies the result after updating the connection parameters.

Data Types: `char` | `string`

`LEPSM` — LE protocol or service multiplexer

`'001F'` (default) | four-element character vector | string scalar denoting two-octet hexadecimal value

LE protocol or service multiplexer, specified as a four-element character vector or a string scalar denoting a two-octet hexadecimal value. The value of this property is a unique number specified by the Special Interest Group (SIG) for each protocol. The SIG assigns the value of this property within the range [0x0001, 0x007F] for a set of existing protocols. The SIG dynamically assigns the value of this property in the range [0x0080, 0x00FF] to the implemented protocols.

Data Types: `char` | `string`

`MaxTransmissionUnit` — Maximum service data unit (SDU) size

`23` (default) | integer in the range of [23, 65,535]

Maximum service data unit (SDU) size, specified as an integer in the range of [23, 65,535] octets. This property specifies the maximum acceptable SDU size for the upper-layer entity.

Data Types: `double`

`MaxPDUPayloadSize` — Maximum protocol data unit (PDU) payload size

`23` (default) | integer in the range of [23, 65,535]

Maximum protocol data unit (PDU) payload size, specified as an integer in the range of [23, 65,535] octets. This property specifies the maximum acceptable payload data for the L2CAP layer entity.

Data Types: `double`

`Credits` — Number of LE-frames peer device can send or receive

`1` (default) | integer in the range of [0, 65,535]

Number of LE-frames peer device can send or receive, specified as an integer in the range of [0, 65,535] octets. This property indicates the number of LE-frames that the peer device can send or receive. If the value of `CommandType` property is set to `'Flow control credit'`, then this property cannot be set to `0`.

Data Types: `double`

`ConnectionResult` — Result of credit-based connection procedure

`'Successful'` (default) | `'LEPSM not supported'` | `'No resources available'` | `...`

Result of the credit-based connection procedure, specified as a character vector or a string scalar. You can specify `ConnectionResult` as one of these values:

- `'Successful'`
- `'LEPSM not supported'`
- `'No resources available'`
- `'Insufficient authentication'`
- `'Insufficient authorization'`
- `'Insufficient encryption key size'`
- `'Insufficient encryption'`
- `'Invalid Source CID'`

- `'Source CID already allocated'`
- `'Unacceptable parameters'`

This property indicates the outcome of the connection request.

Data Types: `char` | `string`

Object Functions

Specific to This Object

bleL2CAPFrame	Generate BLE L2CAP frame
bleL2CAPFrameDecode	Decode BLE L2CAP frame

Examples

Create BLE L2CAP Frame Configuration Object Using Default Settings

Create a BLE L2CAP frame configuration object, `'cfgL2CAP'`, using default properties. View the corresponding applicable properties. This configuration object generates a BLE L2CAP signaling frame of type `'Credit based connection request'`.

```
cfgL2CAP = bleL2CAPFrameConfig
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Credit based connection request'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 1
```

Set the value of credits to `10`. View the corresponding properties of `'cfgL2CAP'`.

```
cfgL2CAP.Credits            = 10
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

          ChannelIdentifier: '0005'
                CommandType: 'Credit based connection request'
           SignalIdentifier: '01'
    SourceChannelIdentifier: '0040'
                      LEPSM: '001F'
        MaxTransmissionUnit: 23
          MaxPDUPayloadSize: 23
                    Credits: 10
```

Create BLE L2CAP Configuration Object Using Name-Value pairs

Create a BLE L2CAP frame configuration object, `'cfgL2CAP'`, by setting the value of channel identifier as `'0004'`using name-value pairs. View the corresponding applicable properties. This configuration object generates a BLE L2CAP data frame (B-frame).

```
cfgL2CAP = bleL2CAPFrameConfig('ChannelIdentifier','0004')
```

```
cfgL2CAP = 
  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'
```

End-to-End Workflow of BLE L2CAP Frames

Create a BLE L2CAP configuration object, `'bleL2CAPFrameConfig'`, to generate a BLE L2CAP data frame (B-frame). Set the value of channel identifier as`'0004'` and view the corresponding applicable properties.

```
cfgTx = bleL2CAPFrameConfig('ChannelIdentifier','0004')
```

```
cfgTx = 
  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'
```

Generate a BLE L2CAP data frame (B-frame) with service data unit (SDU) from ATT specified as `'0A0100'`.

```
l2capFrame = bleL2CAPFrame(cfgTx,"0A0100")
```

```
l2capFrame = 7x2 char array
    '03'
    '00'
    '04'
    '00'
    '0A'
    '01'
    '00'
```

Decode the generated BLE L2CAP data frame (B-frame). The returned status indicates decoding was successful. View the output of `'status'`, `'cfgRx'` and `'SDU'`.

```
[status, cfgRx, SDU] = bleL2CAPFrameDecode(l2capFrame)
```

```
status = 
Success

cfgRx = 
  bleL2CAPFrameConfig with properties:

    ChannelIdentifier: '0004'
```

```
SDU = 3x2 char array
    '0A'
    '01'
    '00'
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleL2CAPFrame` | `bleL2CAPFrameDecode`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLDataChannelPDUConfig

Configuration object for BLE LL data channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleLLDataChannelPDUConfig` creates a configuration object for a Bluetooth low energy (BLE) link layer (LL) data channel protocol data unit (PDU) using the default and specified values. You can configure a BLE LL data PDU and a BLE LL control PDU using the applicable properties of `bleLLDataChannelPDUConfig`.

Creation

Syntax

```
cfgLLData = bleLLDataChannelPDUConfig
cfgLLData = bleLLDataChannelPDUConfig(Name,Value)
```

Description

`cfgLLData = bleLLDataChannelPDUConfig` creates a configuration object, `cfgLLData`, for a BLE LL data channel PDU with default values.

`cfgLLData = bleLLDataChannelPDUConfig(Name,Value)` sets properties by using one or more name-value pairs. Enclose each property name in quotes. For example, `bleLLDataChannelPDUConfig('LLID','Data (start fragment/complete)')` configures `cfgLLData` with the type of BLE LL data channel PDU as a data PDU.

Properties

Note For more information about BLE LL data channel PDU properties and their respective values, see volume 6, part B, section 2.4 of the Bluetooth Core Specification [2].

`LLID` — Link layer identifier

`'Data (continuation fragment/empty)'` (default) | `'Data (continuation fragment/empty)'` | `'Control'`

Link layer identifier, specified as a character vector or a string scalar to describe the type of a BLE LL data channel PDU. You can specify `'LLID'` as one of these values:

- `'Data (continuation fragment/empty)'`

- `'Data (start fragment/complete)'`
- `'Control'`

Data Types: `char` | `string`

`NESN` — Next expected sequence number
`0` (default) | `1`

Next expected sequence number, specified as `0` or `1`. This property indicates either an acknowledgment for the last received packet or a request for resending the last transmitted packet from the peer.

Data Types: `logical` | `double`

`SequenceNumber` — Transmitting packet sequence number
`0` (default) | `1`

Transmitting packet sequence number, specified as `0` or `1`. This property indicates the sequence number of the packet being transmitted.

Data Types: `logical` | `double`

`MoreData` — Indication for more data
`false` or `0` (default) | `true` or `1`

Indication for more data, specified as a numeric or a logical value of `1` (`true`) or `0` (`false`). A `true` or `1` value indicates that the sender has more data to send.

Data Types: `logical`

`CRCInitialization` — Cyclic redundancy check initialization value
`'012345'` (default) | six-element character vector | string scalar denoting three-octet hexadecimal value

Cyclic redundancy check initialization value, specified as a six-element character vector or a string scalar denoting a three-octet hexadecimal value.

Data Types: `char` | `string`

`ControlConfig` — BLE LL control PDU payload configuration object
`bleLLControlPDUConfig` object

BLE LL control PDU payload configuration object, specified as a `bleLLControlPDUConfig` object. The different fields of this value are configured using the settings of `bleLLControlPDUConfig`. The default value of this property is an object of type `bleLLControlPDUConfig` with all properties set to their default values. This property is applicable for generating and decoding a BLE LL control PDU.

Object Functions

Specific to This Object

bleLLDataChannelPDU	Generate BLE LL data channel PDU
bleLLDataChannelPDUDecode	Decode BLE LL data channel PDU
bleLLControlPDUConfig	Configuration object for BLE LL control PDU payload configuration

Examples

Create BLE LL Data Channel PDU Configuration Object for Data PDU

Create a BLE LL data channel PDU configuration object for a data PDU and view the corresponding default properties.

```
cfgLLData = bleLLDataChannelPDUConfig
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (continuation fragment/empty)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
   CRCInitialization: '012345'
```

Specify the value of LLID as `'Data (start fragment/complete)'` and view the corresponding properties.

```
cfgLLData.LLID = 'Data (start fragment/complete)'
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (start fragment/complete)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
   CRCInitialization: '012345'
```

Create BLE LL Data Channel PDU Configuration Object for Control PDU

Create two unique BLE LL data channel PDU configuration objects for a control PDU of type `'Channel map indication'` and `'Version indication'` using name-value pairs.

Create a BLE LL control PDU payload configuration object, `'cfgControl'`, for a control PDU with opcode `'Channel map indication'`. View the applicable properties corresponding to the specified opcode.

```
cfgControl = bleLLControlPDUConfig('Opcode','Channel map indication')
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

          Opcode: 'Channel map indication'
         Instant: 0
    UsedChannels: [1x37 double]
```

Create a BLE LL data channel configuration object by specifying the values of `'LLID'` as `'Control'` and `'ControlConfig'` as `'cfgControl'`. View the properties of `'cfgLLData'`.

```
 cfgLLData = bleLLDataChannelPDUConfig('LLID','Control', ...
    'ControlConfig',cfgControl)
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

               LLID: 'Control'
               NESN: 0
     SequenceNumber: 0
           MoreData: 0
  CRCInitialization: '012345'
      ControlConfig: [1x1 bleLLControlPDUConfig]
```

Create another BLE LL data channel configuration object, `'cfgControl'` for a control PDU, this time specifying the opcode as `'Version indication'`. View the applicable properties of `'cfgControl'` corresponding to the specified opcode.

```
cfgControl.Opcode = 'Version indication';
cfgControl.SubVersionNumber = '008E'
```

```
cfgControl = 
  bleLLControlPDUConfig with properties:

             Opcode: 'Version indication'
      VersionNumber: '5.0'
  CompanyIdentifier: 'FFFF'
   SubVersionNumber: '008E'
```

Create a BLE LL data channel configuration object, `'cfgLLData'`, by specifying the values of `'ControlConfig'` as `'cfgControl'` and view the applicable properties.

```
cfgLLData.ControlConfig = cfgControl
```

```
cfgLLData = 
  bleLLDataChannelPDUConfig with properties:

               LLID: 'Control'
               NESN: 0
     SequenceNumber: 0
           MoreData: 0
  CRCInitialization: '012345'
      ControlConfig: [1x1 bleLLControlPDUConfig]
```

End-to-End Workflow of BLE LL Data Channel PDU

Create a BLE LL data channel PDU configuration object, `'cfgTx'`, for a data PDU using default configuration. View the corresponding default properties.

```
cfgTx = bleLLDataChannelPDUConfig
cfgTx =
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (continuation fragment/empty)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
   CRCInitialization: '012345'
```

Initialize the CRC value corresponding to the configuration object `'cfgTx'`.

```
cfgTx.CRCInitialization = '123456';
```

Generate a BLE LL data channel PDU by using the data PDU configuration object, `'cfgTx'` and the upper-layer payload `'030004000A0100'`.

```
pdu = bleLLDataChannelPDU(cfgTx,'030004000A0100');
```

Decode the generated BLE LL data channel PDU by initializing the CRC value. The returned status indicates decoding is successful. View the values of `'status'`, `'cfgRx'` and `'llPayload'`.

```
crcInit = '123456';
[status, cfgRx, llPayload] = bleLLDataChannelPDUDecode(pdu, crcInit)
```

```
status =
Success

cfgRx =
  bleLLDataChannelPDUConfig with properties:

                LLID: 'Data (continuation fragment/empty)'
                NESN: 0
      SequenceNumber: 0
            MoreData: 0
   CRCInitialization: '012345'


llPayload = 7x2 char array
    '03'
    '00'
    '04'
    '00'
    '0A'
    '01'
    '00'
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleLLDataChannelPDU` | `bleLLDataChannelPDUDecode`

Objects

`bleLLControlPDUConfig`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bleLLAdvertisingChannelPDUConfig

Configuration object for BLE LL advertising channel PDU

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleLLAdvertisingChannelPDUConfig` creates a configuration object for Bluetooth low energy (BLE) link layer (LL) advertising channel protocol data unit (PDU) using the default and specified values. You can configure a BLE LL advertising channel PDU using the applicable properties of `bleLLAdvertisingChannelPDUConfig`.

Creation

Syntax

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig
cfgLLAdv = bleLLAdvertisingChannelPDUConfig(Name,Value)
```

Description

`cfgLLAdv = bleLLAdvertisingChannelPDUConfig` creates a configuration object, `cfgLLAdv`, for a BLE LL advertising channel PDU with default values.

`cfgLLAdv = bleLLAdvertisingChannelPDUConfig(Name,Value)` creates a BLE LL advertising channel PDU configuration object by specifying properties using one or more name-value pairs. Enclose each property name in quotation marks. For example, `bleLLAdvertisingChannelPDUConfig('PDUType','Scan response')` configures `cfgLLAdv` with the type of BLE LL advertising channel PDU as `'Scan response'`.

Properties

Note For more information about BLE LL advertising channel PDU properties and their respective values, see volume 6, part B, section 2.3 of the Bluetooth Core Specification [2].

PDUType — BLE LL advertising channel PDU type
`'Advertising indication'` (default) | `'Advertising direct indication'` | `'Advertising non connectable indication'` | ...

BLE LL advertising channel PDU type, specified as a character vector or a string scalar to describe the type of a BLE LL advertising channel PDU. You can specify `'PDUType'` as one of these values:

- `'Advertising indication'`
- `'Advertising direct indication'`
- `'Advertising non connectable indication'`
- `'Scan request'`
- `'Scan response'`
- `'Connection indication'`
- `'Advertising scannable indication'`

Data Types: `char` | `string`

`ChannelSelection` — Channel selection algorithm
`'Algorithm1'` (default) | `'Algorithm2'`

Channel selection algorithm, specified as `'Algorithm1'` or `'Algorithm2'` to indicate the type of algorithm used for hopping between channels.

Data Types: `char` | `string`

`AdvertiserAddressType` — Advertiser device address type
`'Random'` (default) | `'Public'`

Advertiser device address type, specified as `'Random'` or `'Public'` to indicate the type of advertiser address in the packet.

Data Types: `char` | `string`

`AdvertiserAddress` — Advertiser device address
`'0123456789AB'` (default) | 12-element character vector | string scalar denoting a six-octet hexadecimal value

Advertiser device address, specified as a 12-element character vector or a string scalar denoting a six-octet hexadecimal value to indicate the advertiser device address.

Data Types: `char` | `string`

`TargetAddressType` — Target device address type
`'Random'` (default) | `'Public'`

Target device address type, specified as `'Random'` or `'Public'` to indicate the type of target device address when a directed advertisement packet is transmitted.

Data Types: `char` | `string`

`TargetAddress` — Target device address
`'0123456789CD'` (default) | 12-element character vector | string scalar denoting a six-octet hexadecimal value

Target device address, specified as a 12-element character vector or a string scalar denoting a six-octet hexadecimal value to indicate the target device address when a directed advertisement packet is transmitted.

Data Types: `char` | `string`

`ScannerAddressType` — Scanner device address type
`'Random'` (default) | `'Public'`

Scanner device address type, specified as `'Random'` or `'Public'` to indicate the type of scanner device address when a scan request packet is transmitted.

Data Types: `char` | `string`

`ScannerAddress` — Scanner device address

`'0123456789CD'` (default) | 12-element character vector | string scalar denoting a six-octet hexadecimal value

Scanner device address, specified as a 12-element character vector or a string scalar denoting a six-octet hexadecimal value to indicate the scanner device address when a scan request packet is transmitted.

Data Types: `char` | `string`

`InitiatorAddressType` — Initiator device address type

`'Random'` (default) | `'Public'`

Initiator device address type, specified as `'Random'` or `'Public'` to indicate the type of initiator device address when a connection indication packet is transmitted.

Data Types: `char` | `string`

`InitiatorAddress` — Initiator device address

`'0123456789CD'` (default) | 12-element character vector | string scalar denoting a six-octet hexadecimal value

Initiator device address, specified as a 12-element character vector or a string scalar denoting a six-octet hexadecimal value to indicate the initiator device address when a connection indication packet is transmitted.

Data Types: `char` | `string`

`AdvertisingData` — Advertising data

`'020106'` (default) | character vector | string scalar | numeric vector of elements in the range [0,31] | `n`-by-2 character array

Advertising data, specified as one of these values:

- Character vector — This vector represent octets in hexadecimal format.
- String scalar — This scalar represent octets in hexadecimal format.
- Numeric vector of elements in the range [0,31] — This vector represent octets in decimal format.
- `n`-by-2 character array — Each row represent an octet in hexadecimal format.

This property indicates the advertising data that the device sends out in an advertisement packet.

Data Types: `char` | `string` | `double`

`ScanResponseData` — Scan response data

`'020106'` (default) | character vector | string scalar | numeric vector of elements in the range [0,31] | `n`-by-2 character array

Scan response data, specified as one of these values:

- Character vector — This vector represent octets in hexadecimal format.

- String scalar — This scalar represent octets in hexadecimal format.
- Numeric vector of elements in the range [0,31] — This vector represent octets in decimal format.
- n-by-2 character array — Each row represent an octet in hexadecimal format.

This property indicates the scan response data that the device sends out in a scan response packet (when scan request is received).

Data Types: `char` | `string` | `double`

`AccessAddress` — Unique connection address
`'01234567'` (default) | eight-element character vector or a string scalar

Unique connection address, specified as an eight-element character vector or a string scalar. This property indicates a unique 32-bit address generated by the LL for the new connection or for a periodic advertisement between two devices.

Data Types: `char` | `string`

`CRCInitialization` — CRC initialization value
`'012345'` (default) | six-element character vector | three-octet hexadecimal value

CRC initialization value, specified as a six-element character vector or a string scalar denoting a three-octet hexadecimal value. This property is used as initialization value for the CRC calculation.

Data Types: `char` | `string`

`WindowSize` — Transmit window size
`1` (default) | nonnegative integer

Transmit window size, specified as a nonnegative integer in the range [1, Mws], where Mws is the lesser of 8 and `ConnectionInterval`-1. This property indicates the window size within which the master transmits the data packet and slave listens for the data packet after connection establishment. Each unit is taken as 1.25 ms so that the window size (`WindowSize`×1.25) is in the range of 1.25 ms to the minimum of (10 ms, ((`ConnectionInterval`×1.25) - 1.25) ms).

Data Types: `double`

`WindowOffset` — Transmit window offset
`0` (default) | nonnegative integer

Transmit window offset, specified as a nonnegative integer in the range [0, Mwo], where Mwo is the lesser of 3200 and `ConnectionInterval`. This property indicates the window offset after which the transmit window starts. Each unit is taken as 1.25 ms so that the resultant window offset (`WindowOffset`×1.25) is in the range of 0 ms to (`ConnectionInterval`×1.25) ms.

Data Types: `double`

`ConnectionInterval` — Connection interval
`6 (7.5 ms)` (default) | positive integer

Connection interval, specified as a positive integer in the range [6, 3200]. This property indicates the interval between the start of two consecutive connection events. Each unit is taken as 1.25 ms so that the resultant connection interval (`ConnectionInterval`×1.25) is in the range of 7.5 ms to 4.0 s.

Data Types: `double`

`SlaveLatency` — Slave latency

`0` (default) | nonnegative integer

Slave latency, specified as a nonnegative integer in the range [0, Msl], where Msl is the lesser of 499 and ((`ConnectionInterval`×10)/((`ConnectionInterval`×1.25)×2))-1. This property indicates the number of connection events that a slave can skip listening for packets from the master.

Data Types: `double`

`ConnectionTimeout` — Connection supervision timeout

`10` (default) | positive integer

Connection supervision timeout, specified as a positive integer in the range [Mct, 3200], where Mct is the larger of 10 and ((1+`SlaveLatency`)×(`ConnectionInterval`×1.25)×2)/10. This property indicates the timeout for a connection if no valid packet is received within this time. Each unit is taken as 10 ms so that the resultant connection timeout (`ConnectionInterval`×10) is in the range of 100 ms to 32.0 s.

Data Types: `double`

`UsedChannels` — List of used data channels

`row vector containing the channel indices between [0:36]` (default) | integer vector with element values in the range [0, 36]

List of used data channels, specified as an integer vector with element values in the range [0, 36]. The vector length must be greater than 1. At least two channels must be set as used (good) channels. This property indicates the set of good channels classified by the master.

Data Types: `double`

`HopIncrement` — Hop increment count

`5` (default) | positive integer

Hop increment count, specified as an integer in the range [5, 16]. This property indicates the hop increment count to be used for hopping between data channels.

Data Types: `double`

`SleepClockAccuracy` — Master sleep clock accuracy

`'251 to 500 ppm'` (default) | `'151 to 250 ppm'` | `'101 to 150 ppm'` | `...`

Master sleep clock accuracy, specified as a character vector or a string scalar indicating the worst case master sleep clock accuracy. You can specify `SleepClockAccuracy` as one of these values:

- `'251 to 500 ppm'`
- `'151 to 250 ppm'`
- `'101 to 150 ppm'`
- `'76 to 100 ppm'`
- `'51 to 75 ppm'`
- `'31 to 50 ppm'`
- `'21 to 30 ppm'`
- `'0 to 20 ppm'`

Data Types: `char` | `string`

Object Functions

Specific to This Object

bleLLAdvertisingChannelPDU	Generate BLE LL advertising channel PDU
bleLLAdvertisingChannelPDUDecode	Decode BLE LL advertising channel PDU

Examples

Create BLE LL Advertising Channel PDU Configuration Object Using Default Settings

Create a BLE LL advertising channel configuration object and view the corresponding default properties.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                 PDUType: 'Advertising indication'
        ChannelSelection: 'Algorithm1'
   AdvertiserAddressType: 'Random'
       AdvertiserAddress: '0123456789AB'
         AdvertisingData: [3x2 char]
```

Create BLE LL Advertising Channel PDU Configuration Object Using Name-Value Pairs

Create two unique BLE LL advertising channel configuration objects of type `'Scan response'` and `'Connection indication'` using name-value pairs.

Create a BLE LL advertising channel PDU configuration object, `'cfgLLAdv'`, by setting the values of PDU type as `'Scan response'`, advertiser address as `'1234567890AB'`, and scan response data as `'020106020AD3'` using name-value pairs. View the corresponding properties.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig(...
    'PDUType','Scan response', ...
    'AdvertiserAddress','1234567890AB', ...
    'ScanResponseData','020106020AD3')
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                 PDUType: 'Scan response'
        ChannelSelection: 'Algorithm1'
   AdvertiserAddressType: 'Random'
       AdvertiserAddress: '1234567890AB'
        ScanResponseData: [6x2 char]
```

Create another BLE LL advertising channel PDU configuration object, `'cfgLLAdv'`, and change the type of PDU to `'Connection indication'`. Set the values of the connection interval as `64` and the set of data channels as `[0 4 12 16 18 24 25]`. View the corresponding properties.

```
cfgLLAdv = bleLLAdvertisingChannelPDUConfig('PDUType', ...
    'Connection indication');
cfgLLAdv.ConnectionInterval = 64;
cfgLLAdv.UsedChannels       = [0 4 12 16 18 24 25]
```

```
cfgLLAdv = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Connection indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
     InitiatorAddressType: 'Random'
         InitiatorAddress: '0123456789CD'
            AccessAddress: '01234567'
        CRCInitialization: '012345'
               WindowSize: 1
             WindowOffset: 0
       ConnectionInterval: 64
             SlaveLatency: 0
        ConnectionTimeout: 10
             UsedChannels: [0 4 12 16 18 24 25]
             HopIncrement: 5
       SleepClockAccuracy: '251 to 500 ppm'
```

End-to-End Workflow of BLE LL Advertising Channel PDU

Create a BLE LL advertising channel PDU configuration object, `'cfgTx'`. Set the values of PDU type as `'Connection indication'`, the connection interval as `8`, and the set of data channels as `[0 4 12 16 18 24 25]`.

```
cfgTx = bleLLAdvertisingChannelPDUConfig('PDUType', ...
    'Connection indication');
cfgTx.ConnectionInterval = 8;
cfgTx.UsedChannels       = [0 4 12 16 18 24 25]
```

```
cfgTx = 
  bleLLAdvertisingChannelPDUConfig with properties:

                  PDUType: 'Connection indication'
         ChannelSelection: 'Algorithm1'
    AdvertiserAddressType: 'Random'
        AdvertiserAddress: '0123456789AB'
     InitiatorAddressType: 'Random'
         InitiatorAddress: '0123456789CD'
            AccessAddress: '01234567'
        CRCInitialization: '012345'
               WindowSize: 1
             WindowOffset: 0
       ConnectionInterval: 8
             SlaveLatency: 0
        ConnectionTimeout: 10
             UsedChannels: [0 4 12 16 18 24 25]
             HopIncrement: 5
       SleepClockAccuracy: '251 to 500 ppm'
```

Generate a BLE LL advertising channel PDU by using the corresponding configuration object.

```
pdu = bleLLAdvertisingChannelPDU(cfgTx);
```

Decode the generated BLE LL advertising channel PDU. The returned status indicates decoding is successful. View the output of `'status'` and `'cfgRx'`.

```
[status, cfgRx] = bleLLAdvertisingChannelPDUDecode(pdu)
```

```
status = 
Success

cfgRx = 
  bleLLAdvertisingChannelPDUConfig with properties:

                PDUType: 'Connection indication'
       ChannelSelection: 'Algorithm1'
   AdvertiserAddressType: 'Random'
       AdvertiserAddress: '0123456789AB'
    InitiatorAddressType: 'Random'
        InitiatorAddress: '0123456789CD'
           AccessAddress: '01234567'
        CRCInitialization: '012345'
              WindowSize: 1
            WindowOffset: 0
      ConnectionInterval: 8
            SlaveLatency: 0
       ConnectionTimeout: 10
            UsedChannels: [0 4 12 16 18 24 25]
            HopIncrement: 5
      SleepClockAccuracy: '251 to 500 ppm'
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleLLAdvertisingChannelPDU` | `bleLLAdvertisingChannelPDUDecode`

Topics

"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2019b

bleGAPDataBlockConfig

Configuration object for BLE GAP data block

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bleGAPDataBlockConfig` creates a configuration object for a Bluetooth low energy (BLE) generic access profile (GAP) data block of the type advertising data (AD) or scan response data (SRD). You can configure a BLE GAP data block using the applicable properties of `bleGAPDataBlockConfig`.

Creation

Syntax

```
cfgGAP = bleGAPDataBlockConfig
cfgGAP = bleGAPDataBlockConfig(Name,Value)
```

Description

`cfgGAP = bleGAPDataBlockConfig` creates a configuration object, `cfgGAP`, for a BLE GAP data block of the type AD or SRD with default values.

`cfgGAP = bleGAPDataBlockConfig(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `bleGAPDataBlockConfig('AdvertisingDataTypes','Tx power level')` configures `cfgGAP` with block data advertising data type as `'Tx power level'`.

Properties

Note For more information about BLE GAP data block properties and their respective values, see vol 3, part C, section 4 of the Bluetooth Core Specification [2].

`AdvertisingDataTypes` — Block data advertising data types
`'Flags'` (default) | `'UUIDs'` | `'Local name'` | `'Tx power level'` | `'Connection interval range'` | `'Advertising interval'`

Block data advertising data types, specified as a character vector or a string scalar or a cell array, containing the list of advertising data types for BLE GAP data block. You can specify `AdvertisingDataTypes` as one of these values:

- 'Flags'
- 'UUIDs'
- 'Local name'
- 'Tx power level'
- 'Connection interval range'
- 'Advertising interval'

Data Types: char | string | cell

LEDiscoverability — LE discoverable mode
'General' (default) | 'None' | 'Limited' | 'Limited and general'

LE discoverable mode, specified as a character vector or a string scalar, describing the LE discoverable mode of the device. You can specify LEDiscoverability as one of these values:

- 'None'
- 'General'
- 'Limited'
- 'Limited and general'

Data Types: char | string

BREDR — Basic rate (BR) or enhanced data rate (EDR) support
false (default) | true

Basic rate (BR) or enhanced data rate (EDR) support, specified as true or false. A true value indicates that BR or EDR is supported. This is used when AdvertisingDataTypes is set to 'Flags'.

Data Types: logical

LE — Simultaneous LE and BR or EDR support
'None' (default) | 'Host' | 'Controller' | 'Host and controller'

Simultaneous LE and BR/EDR support, specified as a character vector or a string scalar. You can specify 'LE' as one of these values:

- 'None'
- 'Host'
- 'Controller'
- 'Host and controller'

This property is applicable only when BREDR is set to true. Set this property to 'Host' or 'Controller' to specify simultaneous LE and BR or EDR support at the host or the controller respectively. Set this property to 'Host and Controller' for simultaneous LE and BR or EDR support at the host and the controller.

Data Types: char | string

LocalNameShortening — Enable shortening of local name
false (default) | true

Local name shortened used, specified as a scalar logical value of `false` or `true`. Set this value to `true` value indicates that the name of the device is shortened.

Data Types: `logical`

`LocalName` — UTF-8 encoded user-friendly descriptive name
`'Bluetooth'` (default) | character vector or a string scalar consisting of UTF-8 characters

UTF-8 encoded user-friendly descriptive name, specified as a character vector or a string scalar consisting of UTF-8 characters. This property specifies the local name assigned to the device.

Data Types: `char` | `string`

`IdentifierType` — Type of 16-bit service or service class identifiers
`'Incomplete'` (default) | `'Complete'`

Type of 16-bit service or service class identifiers, specified as `'Incomplete'` or `'Complete'`. If this value is `'Incomplete'`, then the list of 16-bit service or service class identifiers list is incomplete.

Data Types: `char` | `string`

`Identifiers` — List of 16-bit service or service class identifiers
`'' (empty)` (default) | *n*-by-4 character array

List of 16-bit service or service class identifiers, specified as an *n*-by-4 character array. The value of *n* must be in the range [0, 127]. Each row in the *n*-by-4 character array is represented as a four-element character vector or string scalar denoting a two-octet (16-bit) hexadecimal value of a service or service class universally unique identifier (UUID). These UUIDs are assigned by the Bluetooth Special Interest Group (SIG).

Data Types: `char`

`AdvertisingInterval` — Advertising interval
`32` (default) | integer in the range [32, 65,535]

Advertising interval, specified as an integer in the range [32, 65,535]. This property denotes the interval between the start of two consecutive advertising events. Incremental units are 0.625 ms steps, so the resultant range for [32, 65,535] is [20, 40.959375].

Data Types: `double`

`TxPowerLevel` — Packet transmit power level
`0 (dBm)` (default) | integer in the range [–127, 127]

Packet transmit power level, in dBm, specified as an integer in the range [–127, 127]. This property calculates the pathloss as *pathloss = Tx Power Level – RSSI*, where *RSSI* is the received signal strength indicator.

Data Types: `double`

`ConnectionIntervalRange` — Connection interval range
`[6,3200]` (default) | two-element numeric vector [MIN, MAX]

Connection interval range, specified as a two-element numeric vector [*MIN*, *MAX*], where *MIN* and *MAX* must be in the range [6,3200]. *MIN* and *MAX* specify the minimum and maximum value for the connection interval respectively. *MIN* must be less than or equal to *MAX*. Incremental units are 1.25 ms steps, so that the resultant range for [6, 3200] is [7.5, 4.0].

Data Types: double

Object Functions

Specific to This Object

bleGAPDataBlock	Generate BLE GAP data block
bleGAPDataBlockDecode	Decode BLE GAP data block

Examples

Create BLE GAP AD Block Configuration Object Using Default Settings

Create two unique BLE GAP AD configuration objects: one with AD types 'Flags' and 'Tx power level' and the other with AD type 'Flags' and simultaneous LE and BR or EDR support at the host.

Create a BLE GAP AD block configuration object using default settings. Set the values of AD types as 'Flags' and 'Tx power level', LE discoverability as 'Limited' and Tx power level as 45. View the properties of the corresponding configuration object.

```
cfgGAP = bleGAPDataBlockConfig;
cfgGAP.AdvertisingDataTypes = {'Flags';'Tx power level'};
cfgGAP.LEDiscoverability = 'Limited';
cfgGAP.TxPowerLevel = 45
```

```
cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
       LEDiscoverability: 'Limited'
                   BREDR: 0
            TxPowerLevel: 45
```

Create another BLE GAP AD block configuration object using default settings, this time with AD type 'Flags' and having simultaneous support for LE and BR/EDR. Set the values of LE discoverability as 'Limited', enable BR or EDR support as true and enable simultaneous support for LE and BR or EDR as 'Host'. View the properties of the corresponding configuration object.

```
cfgGAP = bleGAPDataBlockConfig;
cfgGAP.LEDiscoverability = 'Limited and general';
cfgGAP.BREDR = true;
cfgGAP.LE = 'Host'
```

```
cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {'Flags'}
       LEDiscoverability: 'Limited and general'
                   BREDR: 1
                      LE: 'Host'
```

Create BLE GAP AD Block Configuration Object Using Name-Value Pairs

Create a configuration object for a BLE GAP AD block using name-value pairs. Set the values of AD types as `'Advertising interval'` and `'Local name'`, advertising interval as `48`, local name as `'MathWorks'` and local name shortening as `true`. View the properties of the corresponding configuration object.

```
cfgGAP = bleGAPDataBlockConfig('AdvertisingDataTypes', ...
    {'Advertising interval', ...
    'Local name'});
cfgGAP.AdvertisingInterval = 48;
cfgGAP.LocalName = 'MathWorks';
cfgGAP.LocalNameShortening = true
```

```
cfgGAP = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'MathWorks'
     LocalNameShortening: 1
     AdvertisingInterval: 48
```

End-to-End Workflow of BLE GAP AD Blocks

Create a configuration object for a BLE GAP AD block using name-value pairs. Set the values of AD types as `'Advertising interval'` and `'Local name'`, advertising interval as `48`, local name as `'MathWorks'`, and local name shortening as `true`. View the properties of the corresponding configuration object.

```
cfgTx = bleGAPDataBlockConfig('AdvertisingDataTypes',{'Advertising interval','Local name'});
cfgTx.AdvertisingInterval = 48;
cfgTx.LocalName = 'MathWorks';
cfgTx.LocalNameShortening = true
```

```
cfgTx = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'MathWorks'
     LocalNameShortening: 1
     AdvertisingInterval: 48
```

Create a BLE GAP AD block using the configuration object `'cfgTx'`.

```
dataBlock = bleGAPDataBlock(cfgTx);
```

Decode the generated BLE GAP AD block. The returned status indicates decoding was successful. View the output of `'status'` and `'cfgRx'`.

```
[status, cfgRx] = bleGAPDataBlockDecode(dataBlock)
```

```
status = 
Success

cfgRx = 
  bleGAPDataBlockConfig with properties:

    AdvertisingDataTypes: {2x1 cell}
               LocalName: 'MathWorks'
     LocalNameShortening: 1
     AdvertisingInterval: 48
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Bluetooth Special Interest Group (SIG). "Supplement to the Bluetooth Core Specification." CSS Version 7 (2016).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`bleGAPDataBlock` | `bleGAPDataBlockDecode`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2019b

bluetoothFrequencyHop

Bluetooth BR/EDR channel index for frequency hopping

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bluetoothFrequencyHop` object creates a Bluetooth basic rate/enhanced data rate (BR/EDR) channel index for frequency hopping. This object generates the hopping sequence used in inquiry, paging, and connection procedures.

Creation

Syntax

```
FH = bluetoothFrequencyHop
FH = bluetoothFrequencyHop(Name,Value)
```

Description

`FH = bluetoothFrequencyHop` creates a default Bluetooth BR/EDR channel index object for frequency hopping.

`FH = bluetoothFrequencyHop(Name,Value)` sets “Properties” on page 3-170 by using one or more name-value pairs. Enclose each property name in quotes. For example, `bluetoothFrequencyHop('SequenceType','Page')` sets the frequency hopping sequence type to `Page`.

Properties

`DeviceAddress` — Bluetooth BR/EDR device address
`'9E8B33'` (default) | 12-element character vector | string scalar denoting a 6-octet hexadecimal value

Bluetooth BR/EDR device address, specified as a 12-element character vector or a string scalar denoting a 6-octet hexadecimal value. This property indicates the Bluetooth BR/EDR device address given as an input to the hop selection kernel. This property ignores all consecutive 0s starting from the most significant bit (MSB). This table maps the value of this property to different physical channels.

Type of Physical Channel	Value of `DeviceAddress` Property
Basic	Address of Master

Type of Physical Channel	Value of DeviceAddress Property
Page scan	Address of the scanning device
Inquiry	General inquiry access code (GIAC)

The default value of this property denotes the lower address part (LAP) of GIAC.

Data Types: char | string

SequenceType — Frequency hopping sequence type

'Inquiry' (default) | 'Page scan' | 'Inquiry scan' | 'Page' | ...

Frequency hopping sequence type, specified as one of these values:

- 'Page scan'
- 'Inquiry scan'
- 'Page'
- 'Inquiry'
- 'Master page response'
- 'Slave page response'
- 'Inquiry response'
- 'Connection basic'
- 'Connection adaptive'
- 'Interlaced page scan'
- 'Interlaced inquiry scan'

Data Types: char | string

InterlaceOffset — Offset for available frequencies in inquiry and paging procedures

16 (default) | integer in the range [0, 31]

Offset for available frequencies in inquiry and paging procedures, specified as an integer in the range [0, 31].

Dependencies

To enable this property, set the “SequenceType” on page 3-0 property to 'Interlaced page scan' or 'Interlaced inquiry scan'.

Data Types: double

KNudge — Offset to compute control signal (X)

0 (default) | even integer

Offset to compute control signal (X), specified as an even integer. This property specifies the additional offset added to the clock bits.

Dependencies

To enable this property, set the SequencyType property to 'Page' or 'Inquiry'.

Data Types: double

`KOffset` — Offset to switch between A-train and B-train

24 (default) | 8

Offset to switch between A-train and B-train, specified as 24 (for A-train) or 8 (for B-train). To switch between the trains, this property specifies the offset added to the clock bits.

Dependencies

To enable this property, set the `SequencyType` property to `'Page'` or `'Inquiry'`.

Data Types: `double`

`Counter` — Counter for Master or Slave page response sequence

`0` (default) | nonnegative integer

Counter for Master or Slave page response sequence, specified as a nonnegative integer. This property is incremented at every Master transmission slot.

Dependencies

To enable this property, set the `SequencyType` property to `'Slave page response'`, `'Master page response'`, or `'Inquiry response'`.

Data Types: `double`

`UsedChannels` — List of used channels

row vector containing all 79 channel indices (default) | vector of integers in the range [0, 78]

List of used channels, specified as a vector of integers in the range [0, 78]. The vector must contain at least 20 elements.

Dependencies

To enable this property, set the `SequencyType` property to `'Connection adaptive'`.

Data Types: `double`

Object Functions

Specific to This Object

nextHop Select Bluetooth BR/EDR channel index to hop for next frequency

Examples

Select Bluetooth BR/EDR Channel Index for Connection Basic Frequency Hopping Sequence

Create a Bluetooth BR/EDR channel index object for frequency hopping.

```
fh = bluetoothFrequencyHop;
```

Specify the frequency hopping sequence type as connection basic.

```
fh.SequenceType = 'Connection basic';
```

Specify a clock value and Bluetooth BR/EDR device address.

```
inputClock = '2C'; % 28-bit
fh.DeviceAddress = '2A96EF25'
```

```
fh = 
  bluetoothFrequencyHop with properties:

      DeviceAddress: '2A96EF25'
       SequenceType: 'Connection basic'
    InterlaceOffset: 16
             KNudge: 0
            KOffset: 24
            Counter: 0
       UsedChannels: [1x79 double]
```

Select a Bluetooth BR/EDR channel index to hop for next frequency.

```
[channelIndex, X] = nextHop(fh,inputClock)
```

```
channelIndex = 27
```

```
X = 11
```

Select Bluetooth BR/EDR Channel Index for Connection Adaptive Frequency Hopping Sequence

Create a Bluetooth BR/EDR channel index object for frequency hopping, specifying frequency hopping sequence type, Bluetooth BR/EDR device address, and used channels.

```
fh = bluetoothFrequencyHop('SequenceType','Connection adaptive', ...
    'DeviceAddress','2A96EF25','UsedChannels',22:78)
```

```
fh = 
  bluetoothFrequencyHop with properties:

      DeviceAddress: '2A96EF25'
       SequenceType: 'Connection adaptive'
    InterlaceOffset: 16
             KNudge: 0
            KOffset: 24
            Counter: 0
       UsedChannels: [1x57 double]
```

Specify a clock value.

```
inputClock = '12C'; % 28-bit
```

Select a Bluetooth BR/EDR channel index to hop for next frequency.

```
[channelIndex, X] = nextHop(fh, inputClock)
```

```
channelIndex = 65
```

```
X = 11
```

Select Bluetooth BR/EDR Channel Index for Page Frequency Hopping Sequence

Create a Bluetooth BR/EDR channel index object for frequency hopping.

```
fh = bluetoothFrequencyHop;
```

Specify the frequency hopping sequence type as page.

```
fh.SequenceType = 'Page';
```

Specify a clock value, offset to select frequencies in A-train, and Bluetooth BR/EDR device address.

```
inputClock = 44; % 28-bit
fh.KOffset = 24;
fh.DeviceAddress = '2A96EF25'
```

```
fh = 
  bluetoothFrequencyHop with properties:

      DeviceAddress: '2A96EF25'
       SequenceType: 'Page'
    InterlaceOffset: 16
             KNudge: 0
            KOffset: 24
            Counter: 0
       UsedChannels: [1x79 double]
```

Select a Bluetooth BR/EDR channel index to hop for next frequency.

```
[channelIndex, X] = nextHop(fh, inputClock)
```

```
channelIndex = 15
```

```
X = 30
```

Select Bluetooth BR/EDR Channel Index for Slave Page Response Frequency Hopping Sequence

Create a Bluetooth BR/EDR channel index object for frequency hopping.

```
fh = bluetoothFrequencyHop;
```

Specify the frequency hopping sequence type as Slave page response.

```
fh.SequenceType = 'Slave page response';
```

Specify a clock value, counter for Slave page response, and Bluetooth BR/EDR device address.

```
frozenClock = '2A'; % 28-bit
fh.Counter = 1;
fh.DeviceAddress = '2A96EF25'
```

```
fh = 
  bluetoothFrequencyHop with properties:
      DeviceAddress: '2A96EF25'
       SequenceType: 'Slave page response'
    InterlaceOffset: 16
             KNudge: 0
            KOffset: 24
            Counter: 1
       UsedChannels: [1x79 double]
```

Select a Bluetooth BR/EDR channel index to hop for next frequency.

```
[channelIndex, X] = nextHop(fh, frozenClock)
```

```
channelIndex = 28
```

```
X = 1
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`bleChannelSelection`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2020b

pcapWriter

PCAP file writer of protocol packets

Description

The `pcapWriter` object writes generated and recovered protocol packets to a packet capture (PCAP) file (`.pcap`).

You can write these packet types to a PCAP file:

- Generated and recovered Bluetooth low energy (BLE) link layer (LL) packets (requires Communications Toolbox Library for the Bluetooth Protocol)
- Generated and recovered 5G NR protocol packets (requires 5G Toolbox™)
- Generated and recovered WLAN protocol packets (requires WLAN Toolbox)

Creation

Syntax

```
pcapObj = pcapWriter
pcapObj = pcapWriter(Name,Value)
```

Description

`pcapObj = pcapWriter` creates a default PCAP file writer object.

`pcapObj = pcapWriter(Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `'ByteOrder','big-endian'` specifies the byte order as big-endian.

Properties

Note The `pcapWriter` object does not overwrite the existing PCAP file. During each call of this object, specify a unique PCAP file name.

`FileName` — Name of the PCAP file
`'capture'` (default) | character row vector | string scalar

Name of the PCAP file, specified as a character row vector or a string scalar.

Data Types: `char` | `string`

`ByteOrder` — Byte order
`'little-endian'` (default) | `'big-endian'`

Byte order, specified as `'little-endian'` or `'big-endian'`.

Data Types: `char` | `string`

Object Functions

Specific to This Object

write	Write protocol packet data to PCAP or PCAPNG file
writeGlobalHeader	Write global header to PCAP file

Examples

Write BLE Link Layer Packet to PCAP File

Create a PCAP file writer object, specifying the name of the PCAP file. Specify the BLE link type.

```
pcapObj = pcapWriter('FileName','writeBLE');
bleLinkType = 251;
```

Write a global header to the PCAP file.

```
writeGlobalHeader(pcapObj,bleLinkType);
```

Specify a BLE LL packet.

```
llpacket = '42BC13E206120E00050014010A001F0040001700170000007D47C0';
```

Write the BLE LL packet to the PCAP file.

```
timestamp = 129100; % Number of microseconds
write(pcapObj,llpacket,timestamp);
```

References

[1] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[2] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`pcapngWriter`

Introduced in R2020b

pcapngWriter

PCAPNG file writer of protocol packets

Description

The `pcapngWriter` object writes generated and recovered protocol packets to a packet capture next generation (PCAPNG) file (`.pcapng`).

You can write these packet types to a PCAPNG file:

- Generated and recovered Bluetooth low energy (BLE) link layer (LL) packets (requires Communications Toolbox Library for the Bluetooth Protocol)
- Generated and recovered 5G NR protocol packets (requires 5G Toolbox)
- Generated and recovered WLAN protocol packets (requires WLAN Toolbox)

Creation

Syntax

```
pcapngObj = pcapngWriter
pcapngObj = pcapngWriter(Name,Value)
```

Description

`pcapngObj = pcapngWriter` creates a default PCAPNG file writer object.

`pcapngObj = pcapngWriter(Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `'ByteOrder','big-endian'` specifies the byte order as big-endian.

Properties

Note The `pcapngWriter` object does not overwrite the existing PCAPNG file. During each call of this object, specify a unique PCAPNG file name.

`FileName` — Name of the PCAPNG file
`'capture'` (default) | character row vector | string scalar

Name of the PCAPNG file, specified as a character row vector or a string scalar.

Data Types: `char` | `string`

`ByteOrder` — Byte order
`'little-endian'` (default) | `'big-endian'`

Byte order, specified as `'little-endian'` or `'big-endian'`.

Data Types: `char` | `string`

`FileComment` — Comment for PCAPNG file

`' '` (default) | character vector | string scalar

Comment for the PCAPNG file, specified as a character vector or a string scalar.

Data Types: `char` | `string`

Object Functions

Specific to This Object

write	Write protocol packet data to PCAP or PCAPNG file
writeCustomBlock	Write custom block to PCAPNG file
writeInterfaceDescriptionBlock	Write interface description block to PCAPNG file

Examples

Write BLE Link Layer Packet to PCAPNG File

Create a PCAPNG file writer object, specifying the name of the PCAPNG file. Specify the BLE link type.

```
pcapngObj = pcapngWriter('FileName','BLELLCapture');
```

Write an interface description block for BLE.

```
interfaceName = 'BLE interface';
bleLinkType = 251;
interfaceId = writeInterfaceDescriptionBlock(pcapngObj,bleLinkType, ...
    interfaceName);
```

Specify a BLE LL packet.

```
llpacket = '42BC13E206120E00050014010A001F0040001700170000007D47C0';
```

Write the BLE LL packet to the PCAPNG format file.

```
timestamp = 0; % Number of microseconds
packetComment = 'This is a BLE packet';
write(pcapngObj,llpacket,timestamp,interfaceId,'PacketComment', ...
    packetComment);
```

References

[1] Tuexen, M. “PCAP Next Generation (Pcapng) Capture File Format.” 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`pcapWriter`

Introduced in R2020b

comm.BlockDeinterleaver

Package: `comm`

(To be removed) Deinterleave input symbols using permutation vector

Compatibility

comm.BlockDeinterleaver will be removed in a future release. Use `deintrlv` instead. For more information, see "Compatibility Considerations" on page 3-183.

Description

The `BlockDeinterleaver` object, which can process variable-sized signals, rearranges the elements of its input vector without repeating or omitting any elements. The input can be real or complex.

To deinterleave the input vector:

1. Define and set up your block deinterleaver object. See "Construction" on page 3-181.
2. Call `step` to rearrange the elements of the input vector according to the properties of `comm.BlockDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.BlockDeinterleaver` creates a block deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the block interleaver System object.

`H = comm.BlockDeinterleaver(Name,Value)` creates object, `H`, with the specified property set to the specified value.

Properties

`PermutationVectorSource`

Permutation vector source

Specify the source of the permutation vector as either `Property` or `Input port`. The default value is `Property`.

`PermutationVector`

Permutation vector

Specify the mapping used to permute the input symbol as a column vector of integers. The default is `[5;4;3;2;1]`. The mapping is a column vector of integers where the number of elements is equal to the length, *N*, of the input to the step method. Each element must be an integer, between 1 and *N*, with no repeated values. The `PermutationVector` property is available only when the `PermutationVectorSource` property is set to `Property`.

Methods

step (To be removed) Deinterleave input symbols using permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Block Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver([3 4 1 2]');
```

```
Warning: COMM.BLOCKINTERLEAVER will be removed in a future release. Use INTRLV instead. See <a hr
```

```
deinterleaver = comm.BlockDeinterleaver([3 4 1 2]');
```

```
Warning: COMM.BLOCKDEINTERLEAVER will be removed in a future release. Use DEINTRLV instead. See <
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data, intData, deIntData]
```

```
ans = 4×3

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver(permVec);
```

```
Warning: COMM.BLOCKINTERLEAVER will be removed in a future release. Use INTRLV instead. See <a h
```

```
deinterleaver = comm.BlockDeinterleaver(permVec);
```

```
Warning: COMM.BLOCKDEINTERLEAVER will be removed in a future release. Use DEINTRLV instead. See <
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Block Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.BlockDeinterleaver` will be removed

Not recommended starting in R2019b

comm.BlockDeinterleaver will be removed in a future release. Use `deintrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`deintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.BlockDeinterleaver`
Package: `comm`

(To be removed) Deinterleave input symbols using permutation vector

Compatibility

step will be removed in a future release. Use `deintrlv` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a block interleaver. The step method forms the output, `Y`, based on the mapping specified by the PermutationVector property as **Output**(`PermutationVector`(*k*))=**Input**(*k*), for *k* = `1`:*N*, where *N* is the length of the permutation vector. The input `X` must be a column vector of the same length, *N*. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.BlockInterleaver

Package: `comm`

(To be removed) Permute input symbols using permutation vector

Compatibility

comm.BlockInterleaver will be removed in a future release. Use `intrlv` instead. For more information, see “Compatibility Considerations” on page 3-187.

Description

The `BlockInterleaver` object permutes the symbols in the input signal. Internally, it uses a set of shift registers, each with its own delay value. This object processes variable-size signals.

To interleave the input signal:

1. Define and set up your block interleaver object. See “Construction” on page 3-185.
2. Call `step` to reorder the input symbols according to the properties of `comm.BlockInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.BlockInterleaver` creates a block interleaver System object, `H` This object permutes the symbols in the input signal based on a permutation vector.

`H = comm.BlockInterleaver(Name,Value)` creates object, `H`, with specified property set to the specified value.

Properties

`PermutationVectorSource`

Permutation vector source

Specify the source of the permutation vector as either `Property` or `Input port`. The default value is `Property`.

`PermutationVector`

Permutation vector

Specify the mapping used to permute the input symbols as an integer column vector. The default is `[5;4;3;2;1]`. The number of elements of the permutation vector property must equal the length of the input vector. The `PermutationVector` property indicates the indices, in order, of the input elements that form the output vector. The relationship **Output**(*k*)=**Input**(`PermutationVector`(*k*)) describes this order. Each integer, *k*, must be between `1` and *N*, where *N* is the number of elements in the permutation vector. The elements in the `PermutationVector` property must be integers between `1` and *N* with no repetitions. The `PermutationVector` property is available only when the `PermutationVectorSource` property is set to `Property`.

Methods

step	(To be removed) Permute input symbols using a permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Block Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver([3 4 1 2]');
```

```
Warning: COMM.BLOCKINTERLEAVER will be removed in a future release. Use INTRLV instead. See <a hr
```

```
deinterleaver = comm.BlockDeinterleaver([3 4 1 2]');
```

```
Warning: COMM.BLOCKDEINTERLEAVER will be removed in a future release. Use DEINTRLV instead. See <
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data, intData, deIntData]
```

```
ans = 4×3

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.BlockInterleaver(permVec);
```

```
Warning: COMM.BLOCKINTERLEAVER will be removed in a future release. Use INTRLV instead. See <a h
```

```
deinterleaver = comm.BlockDeinterleaver(permVec);
```

```
Warning: COMM.BLOCKDEINTERLEAVER will be removed in a future release. Use DEINTRLV instead. See <
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Block Interleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.BlockInterleaver` will be removed

Not recommended starting in R2019b

comm.BlockInterleaver will be removed in a future release. Use `intrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`deintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.BlockInterleaver`
Package: `comm`

(To be removed) Permute input symbols using a permutation vector

Compatibility

step will be removed in a future release. Use `intrlv` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The step method forms the output `Y`, based on the mapping defined by the PermutationVector property as **Output**(*k*)=**Input**(`PermutationVector`(*k*)), for *k* = `1`:*N*, where *N* is the length of the `PermutationVector` property. The input `X` must be a column vector of length *N*. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.BPSKDemodulator

Package: `comm`

Demodulate using BPSK method

Description

The `BPSKDemodulator` object demodulates a signal that was modulated using the binary phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a binary phase shift signal:

1. Define and set up your BPSK demodulator object. See “Construction” on page 3-189.
2. Call `step` to demodulate a signal according to the properties of `comm.BPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.BPSKDemodulator` creates a demodulator System object, `H`, that demodulates the input signal using the binary phase shift keying (BPSK) method.

`H = comm.BPSKDemodulator(Name,Value)` creates a BPSK demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.BPSKDemodulator(PHASE,Name,Value)` creates a BPSK demodulator object, `H`, with the `PhaseOffset` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

`PhaseOffset`

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a finite, real scalar. The default is `0`.

`DecisionMethod`

Demodulation decision method

Specify the decision method the object uses as one of `Hard decision` | `Log-likelihood ratio` | `Approximate log-likelihood ratio`. The default is `Hard decision`.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of Property | Input port. The default is Property. This property applies when you set the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a nonzero, real scalar. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations can yield Inf or -Inf. This variance occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite precision arithmetic. As a best practice in such cases, use approximate LLR because this option's algorithm does not compute exponentials. This property applies when you set the VarianceSource on page 3-0 property to Property. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as one of Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical. The default is Full precision. This property applies only when you set the DecisionMethod on page 3-0 property to Hard decision. Thus, when you set the OutputDataType on page 3-0 property to Full precision, and the input data type is single or double precision, the output data has the same data type as the input. If the input data is of a fixed-point type, then the output data type behaves as if you had set the OutputDataType property to Smallest unsigned integer. If you set the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data type is the same as that of the input. In this case, that data type can only be single or double precision.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer or Full precision mode.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as one of Same word length as input | Custom. The default is Same word length as input. This property applies when you set the DecisionMethod on page 3-0 property to Hard decision. The object uses the derotate factor in the computations only when certain conditions exist. The step method input must be of a fixed-point type, and the PhaseOffset on page 3-0 property must have a value that is not a multiple of π/2.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled, numerictype object with a Signedness of Auto. The default is numerictype([],16). This property applies when you set the

DecisionMethod on page 3-0 property to Hard decision and the DerotateFactorDataType on page 3-0 property to Custom.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using BPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate BPSK Signal and Calculate Errors

Generate a BPSK signal, pass it through an AWGN channel, demodulate the signal, and compute the error statistics.

Create BPSK modulator and demodulator System objects.

```
bpskModulator = comm.BPSKModulator;
bpskDemodulator = comm.BPSKDemodulator;
```

Create an error rate calculator System object.

```
errorRate = comm.ErrorRate;
```

Generate 50-bit random data frames, apply BPSK modulation, pass the signal through an AWGN channel, demodulate the received data, and compile the error statistics.

```
for counter = 1:100
    % Transmit a 50-symbol frame
    txData = randi([0 1],50,1);            % Generate data
    modSig = bpskModulator(txData);        % Modulate
    rxSig = awgn(modSig,5);                % Pass through AWGN
    rxData = bpskDemodulator(rxSig);       % Demodulate
    errorStats = errorRate(txData,rxData); % Collect error stats
end
```

Display the cumulative error statistics.

```
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 0.005600
Number of errors = 28
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the BPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BPSKModulator` | `comm.PSKDemodulator`

Introduced in R2012a

constellation

System object: `comm.BPSKDemodulator`
Package: `comm`

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Reference Signal Constellation for BPSK Demodulator

Create BPSK Demodulator System object™ and calculate its reference constellation.

Create a `comm.BPSKDemodulator` System object.

```
h = comm.BPSKDemodulator;
```

Calculate and display the reference signal constellation by calling the `constellation` function.

```
refC = constellation(h)
```

```
refC = 2×1 complex

   1.0000 + 0.0000i
  -1.0000 + 0.0000i
```

Plot BPSK Demodulator Reference Signal Constellation

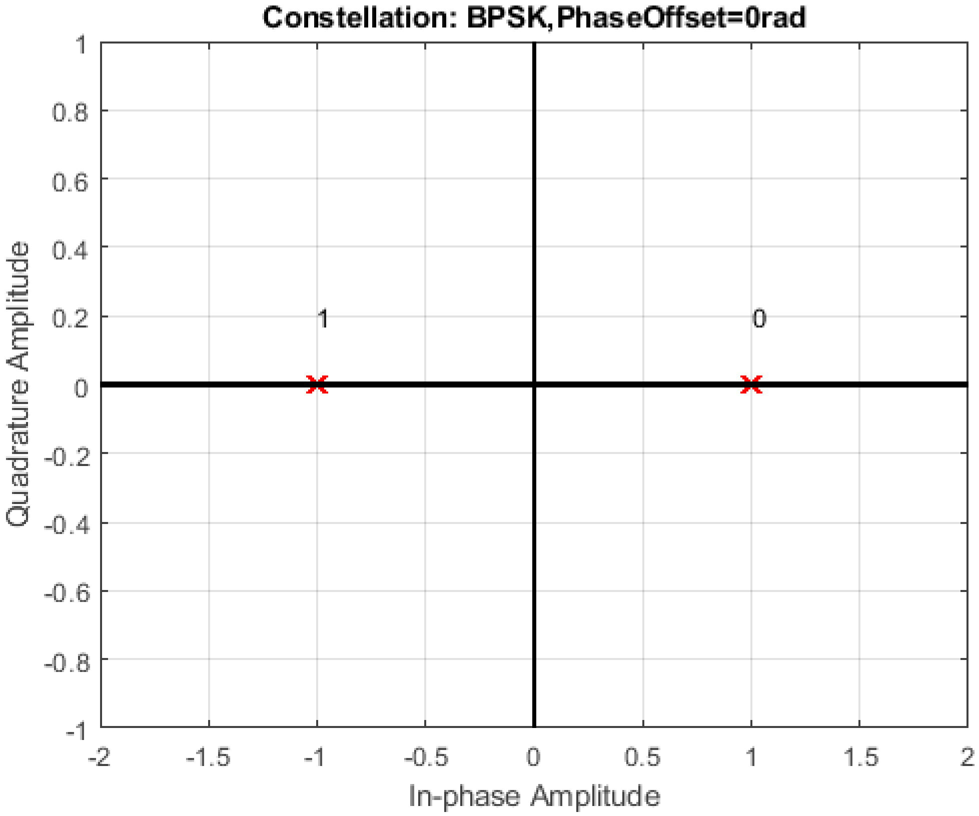
Create a BPSK Demodulator System object™ and then plot the reference signal constellation.

Create a `comm.BPSKDemodulator` System object.

```
h = comm.BPSKDemodulator;
```

Plot the reference constellation by calling the `constellation` function.

```
constellation(h)
```


Constellation: BPSK,PhaseOffset=0rad
Quadrature Amplitude
In-phase Amplitude
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
0

step

System object: `comm.BPSKDemodulator`
Package: `comm`

Demodulate using BPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the BPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `DecisionMethod` property to `Hard decision`, the data type of the input can also be signed integer, or signed fixed point (fi objects).

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `DecisionMethod` property to `Log-likelihood ratio` or `Approximate log-likelihood ratio` and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

bluetoothPhyConfig

Configuration object for Bluetooth BR/EDR PHY

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bluetoothPhyConfig` object creates a configuration object for the Bluetooth physical layer (PHY) using default and specified values. You can configure the Bluetooth PHY by using the applicable properties of the `bluetoothPhyConfig` object.

Creation

Syntax

```
cfg = bluetoothPhyConfig
cfg = bluetoothPhyConfig(Name,Value)
```

Description

`cfg = bluetoothPhyConfig` creates a configuration object, `cfg`, for Bluetooth PHY with default property values.

`cfg = bluetoothPhyConfig(Name,Value)` sets “Properties” on page 3-196 by using one or more name-value pairs. Enclose each property name in quotes. For example, `bluetoothPhyConfig('Mode','EDR3M')` sets the physical layer transmission mode to 3 Mbps.

Properties

Note For more information about Bluetooth BR/EDR waveform generator properties and their respective values, see volume 2, part B, sections 6 and 7 of the Bluetooth Core Specification [2].

`Mode` — PHY transmission mode
`'BR'` (default) | `'EDR2M'` | `'EDR3M'`

PHY transmission mode, specified as `'BR'`, `'EDR2M'` or `'EDR3M'`. This value indicates the type of Bluetooth BR/EDR waveform.

Data Types: `char` | `string`

`DeviceAddress` — Bluetooth device address
`'0123456789AB'` (default) | 12-element character vector | string scalar denoting 6-octet hexadecimal value

Bluetooth device address, specified as a 12-element character vector or a string scalar denoting a 6-octet hexadecimal value.

Data Types: `char` | `string`

`ModulationIndex` — Modulation Index
`0.32` (default) | scalar in the range [0.28, 0.35]

Modulation index, specified as a scalar in the range [0.28, 0.35]. This property is the modulation index that the object uses when performing Gaussian frequency shift keying (GFSK) modulation or demodulation.

Data Types: `double`

`SamplesPerSymbol` — Samples per symbol
8 (default) | positive integer

Samples per symbol, specified as a positive integer. This value is used for GFSK modulation and demodulation.

Data Types: `double`

`WhitenStatus` — Data whiten status
`'On'` (default) | `'Off'`

Data whiten status, specified as `'On'` or `'Off'`. Set this value to `'On'` for the object to perform whitening on header and payload bits.

Data Types: `char` | `string`

`WhitenInitialization` — Whiten initialization
`[1; 1; 1; 1; 1; 1; 1]` (default) | 7-bit binary column vector

Whiten initialization, specified as a 7-bit binary column vector.

Dependencies

To enable this property, set the “WhitenStatus” on page 3-0 property to `'On'`.

Data Types: `double`

Object Functions

Specific to This Object

bluetoothIdealReceiver	Ideal receiver for Bluetooth BR/EDR PHY waveform

Examples

Create Bluetooth BR/EDR PHY Configuration Objects

Create two unique Bluetooth BR/EDR PHY configuration objects: one for synchronous connection oriented (SCO) logical transport and the other for connectionless slave broadcast (CSB) logical transport.

Create a Bluetooth BR/EDR PHY configuration object for an SCO logical transport. For SCO logical transport, the PHY transmission mode must be set to basic rate (BR).

```
cfg = bluetoothPhyConfig
```

```
cfg = 
  bluetoothPhyConfig with properties:

                  Mode: 'BR'
         DeviceAddress: '0123456789AB'
       ModulationIndex: 0.3200
      SamplesPerSymbol: 8
          WhitenStatus: 'On'
    WhitenInitialization: [7x1 double]
```

Create another Bluetooth BR/EDR PHY configuration object for a CSB logical transport by disabling the whiten status.

```
cfg = bluetoothPhyConfig('WhitenStatus','Off')
```

```
cfg = 
  bluetoothPhyConfig with properties:

              Mode: 'BR'
     DeviceAddress: '0123456789AB'
   ModulationIndex: 0.3200
  SamplesPerSymbol: 8
      WhitenStatus: 'Off'
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Properties must be given as: coder.Constant().

See Also

Functions

bleIdealReceiver | bleWaveformGenerator | bluetoothIdealReceiver | bluetoothWaveformGenerator

Objects
`bluetoothWaveformConfig`

Topics
"Bluetooth Protocol Stack"
"Bluetooth Packet Structure"

Introduced in R2020a

bluetoothWaveformConfig

Configuration object for Bluetooth BR/EDR waveform generator

Note **Download Required:** To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Description

The `bluetoothWaveformConfig` object creates a configuration object for the Bluetooth BR/EDR waveform generator using default and specified values. You can configure the Bluetooth waveform generator by using the applicable properties of the `bluetoothWaveformConfig` object.

Creation

Syntax

```
cfg = bluetoothWaveformConfig
cfg = bluetoothWaveformConfig(Name,Value)
```

Description

`cfg = bluetoothWaveformConfig` creates a configuration object, `cfg`, for a Bluetooth BR/EDR waveform generator with default property values.

`cfg = bluetoothWaveformConfig(Name,Value)` sets “Properties” on page 3-200 by using one or more name-value pairs. Enclose each property name in quotes. For example, `bluetoothWaveformConfig('Mode','EDR3M')` sets the physical layer transmission mode to 3 Mbps.

Properties

Note For more information about Bluetooth BR/EDR waveform generator properties and their respective values, see volume 2, part B, sections 6 and 7 of the Bluetooth Core Specification [2].

`Mode` — PHY transmission mode
`'BR'` (default) | `'EDR2M'` | `'EDR3M'`

PHY transmission mode, specified as `'BR'`, `'EDR2M'` or `'EDR3M'`. This value indicates the type of Bluetooth BR/EDR waveform.

Data Types: `char` | `string`

`PacketType` — Packet type
`'FHS'` (default) | `'ID'` | `'NULL'` | `'POLL'` | `...`

Packet type, specified as one of these values:

- `'ID'`
- `'NULL'`
- `'POLL'`
- `'FHS'`
- `'DM1'`
- `'HV1'`
- `'HV2'`
- `'HV3'`
- `'DV'`
- `'EV3'`
- `'EV4'`
- `'EV5'`
- `'DH1'`
- `'AUX1'`
- `'DM3'`
- `'DH3'`
- `'DM5'`
- `'DH5'`
- `'2-EV3'`
- `'3-EV3'`
- `'2-EV5'`
- `'3-EV5'`
- `'2-DH1'`
- `'3-DH1'`
- `'2-DH3'`
- `'3-DH3'`
- `'2-DH5'`
- `'3-DH5'`

Data Types: `char` | `string`

`PayloadLength` — Payload length of packet
18 (default) | integer in the range [0, *X*]

Payload length of packet, specified as an integer in the range [0, *X*], where *X* depends on the "PacketType" on page 3-0 property. This value sets the number of bytes to be processed in a packet.

Dependencies

To enable this property, set the value of packet type to `'DM1'`, `'DH1'`, `'DM3'`, `'DH3'`, `'DM5'`, `'DH5'`, `'AUX1'`, or `'DV'`.

Data Types: `double`

`DeviceAddress` — Bluetooth device address

`'0123456789AB'` (default) | 12-element character vector | string scalar denoting 6-octet hexadecimal value

Bluetooth device address, specified as a 12-element character vector or string scalar denoting a 6-octet hexadecimal value.

Data Types: `char` | `string`

`LogicalTransportAddress` — Logical transport address for packet

`[0; 0; 1]` (default) | 3-bit binary column vector

Logical transport address for the packet, specified as a 3-bit binary column vector. This property indicates the active destination slave for a packet in the master-to-slave transmission slot.

Data Types: `double`

`HeaderControlBits` — Header control information

`[1; 1; 1]` (default) | 3-bit binary column vector

Header control information, specified as a 3-bit binary column vector. This property indicates the header control information consisting of these three fields:

Field	Size of Field	Field Information Indicates
FLOW	1 bit	Flow control information
ARQN	1 bit	Acknowledgment for successful reception of a CRC packet
SEQN	1 bit	Sequencing scheme to order the packet stream

Data Types: `double`

`ModulationIndex` — Modulation Index

`0.32` (default) | scalar in the range [0.28, 0.35]

Modulation index, specified as a scalar in the range [0.28, 0.35]. This property is the modulation index that the object uses when performing Gaussian frequency shift keying (GFSK) modulation or demodulation.

Data Types: `double`

`SamplesPerSymbol` — Samples per symbol

`8` (default) | positive integer

Samples per symbol, specified as a positive integer. This value is used for GFSK modulation and demodulation.

Data Types: `double`

`WhitenStatus` — Data whiten status

`'On'` (default) | `'Off'`

Data whiten status, specified as `'On'` or `'Off'`. Set this value to `'On'` for the object to perform whitening on the header and payload bits.

Data Types: `char` | `string`

`WhitenInitialization` — Whiten initialization
`[1; 1; 1; 1; 1; 1; 1]` (default) | 7-bit binary column vector

Whiten initialization, specified as a 7-bit binary column vector.

Dependencies

To enable this property, set the "WhitenStatus" on page 3-0 property to `'On'`.

Data Types: `double`

`LLID` — Logical link identifier
`[1; 1]` (default) | 2-bit binary column vector

Logical link identifier, specified as a 2-bit binary column vector.

Dependencies

To enable this property, set the value of "PacketType" on page 3-0 to `'DM1'`, `'DH1'`, `'DM3'`, `'DH3'`, `'DM5'`, `'DH5'`, `'AUX1'`, and `'DV'`.

Data Types: `double`

`FlowIndicator` — Logical channel flow control indicator
`true` (default) | `false`

Logical channel flow control indicator, specified as `true` or `false`.

Dependencies

To enable this property, set the value of packet type to `'DM1'`, `'DH1'`, `'DM3'`, `'DH3'`, `'DM5'`, `'DH5'`, `'AUX1'`, and `'DV'`.

Data Types: `logical`

Object Functions

Specific to This Object

getPayloadLength	Payload length in bytes for Bluetooth BR/EDR format configuration
getPhyConfigProperties	Updated configuration properties of Bluetooth BR/EDR PHY configuration object
bluetoothWaveformGenerator	Waveform generator for Bluetooth BR/EDR PHY
bluetoothIdealReceiver	Ideal receiver for Bluetooth BR/EDR PHY waveform

Examples

Create Bluetooth BR/EDR Waveform Configuration Objects

Create three unique Bluetooth BR/EDR waveform configuration objects for synchronous connection oriented (SCO), connectionless slave broadcast (CSB), and asynchronous connectionless (ACL) logical transports.

Create a Bluetooth BR/EDR waveform configuration object for an SCO logical transport by specifying packet type as HV1. For SCO logical transport, the PHY transmission mode must be basic rate (BR).

```
cfg = bluetoothWaveformConfig;
cfg.PacketType = 'HV1'
```

```
cfg = 
  bluetoothWaveformConfig with properties:

                       Mode: 'BR'
                 PacketType: 'HV1'
              DeviceAddress: '0123456789AB'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
            ModulationIndex: 0.3200
           SamplesPerSymbol: 8
               WhitenStatus: 'On'
       WhitenInitialization: [7x1 double]
```

Create a second Bluetooth BR/EDR waveform configuration object for a CSB logical transport by specifying the packet type as DH1 and disabling the whiten status.

```
cfg = bluetoothWaveformConfig('PacketType','DH1','WhitenStatus','Off')
```

```
cfg = 
  bluetoothWaveformConfig with properties:

                       Mode: 'BR'
                 PacketType: 'DH1'
              PayloadLength: 18
              DeviceAddress: '0123456789AB'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
            ModulationIndex: 0.3200
           SamplesPerSymbol: 8
               WhitenStatus: 'Off'
                       LLID: [2x1 double]
              FlowIndicator: 1
```

Create a third Bluetooth BR/EDR waveform configuration object for an ACL logical transport with enhanced data rate mode. Set the value of packet type to DH3, and payload length to 184 bytes.

```
cfg = bluetoothWaveformConfig;
cfg.Mode = 'EDR2M';
cfg.PacketType = 'DH3';
cfg.PayloadLength = 184  % in bytes
```

```
cfg = 
  bluetoothWaveformConfig with properties:

                       Mode: 'EDR2M'
                 PacketType: 'DH3'
              PayloadLength: 184
              DeviceAddress: '0123456789AB'
    LogicalTransportAddress: [3x1 double]
          HeaderControlBits: [3x1 double]
```

```
         ModulationIndex: 0.3200
        SamplesPerSymbol: 8
            WhitenStatus: 'On'
    WhitenInitialization: [7x1 double]
                    LLID: [2x1 double]
           FlowIndicator: 1
```

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

Properties must be given as: coder.Constant().

See Also

Functions

`bleIdealReceiver` | `bleWaveformGenerator` | `bluetoothIdealReceiver` | `bluetoothWaveformGenerator`

Objects

`bluetoothPhyConfig`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2020a

comm.IQImbalanceCompensator

Package: `comm`

Compensate for I/Q imbalance

Description

The `IQImbalanceCompensator` System object compensates for the imbalance between the in-phase and quadrature components of a modulated signal.

To compensate for I/Q imbalance:

1. Define and set up the `IQImbalanceCompensator` object. See “Construction” on page 3-206.
2. Call `step` to compensate for the I/Q imbalance according to the properties of `comm.IQImbalanceCompensator`. The behavior of `step` is specific to each object in the toolbox.

The adaptive algorithm inherent to the I/Q imbalance compensator is compatible with M-PSK, M-QAM, and OFDM modulation schemes, where *M>2*.

Note The output of the compensator might be scaled and rotated, that is, multiplied by a complex number, relative to the reference constellation. In practice, this is not an issue as receivers correct for this prior to demodulation through the use of channel estimation.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.IQImbalanceCompensator` creates a compensator System object, `H`, that compensates for the imbalance between the in-phase and quadrature components of the input signal.

`H = comm.IQImbalanceCompensator(Name,Value)` creates an I/Q imbalance compensator object, `H`, with each specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`CoefficientSource`

Source of compensator coefficients

Specify either `Estimated from input signal` or `Input port`. If the `CoefficientSource` property is set to `Estimated from input signal`, the compensator uses an adaptive algorithm to estimate the compensator coefficient from the input signal. If the `CoefficientSource` property is set to `Input port`, all other properties are disabled and the compensator coefficients must be provided to the `step` function as an input argument. The default value is `Estimated from input signal`. This property is nontunable.

InitialCoefficent

Initial coefficient used to compensate for I/Q imbalance

The initial coefficient is a complex scalar that can be either single or double precision. The default value is `0+0i`. This property is nontunable.

StepSizeSource

Source of step size for coefficient adaptation

Specify either `Property` or `Input port`. If `StepSizeSource` is set to `Property`, you specify the step size through the `StepSize` property. Otherwise, the step size is provided to the `step` function as an input argument. The default value is `Property`. This property is nontunable.

StepSize

Adaptation step size

Specifies the step size used by the algorithm in estimating the I/Q imbalance. This property is accessible only when `StepSizeSource` is set to `Property`. The default value is `1e-5`. This property is tunable.

AdaptInputPort

Creates input port to control compensator coefficient adaptation

When this logical property is `true`, an input port is created to enable or disable coefficient adaptation. If `AdaptInputPort` is `false`, the coefficients update after each output sample. The default value is `false`. This property is nontunable.

CoefficientOutputPort

Create port to output compensator coefficients

When this logical property is `true`, the I/Q imbalance compensator coefficients are made available through an output argument of the `step` function. The default value is `false`. This property is nontunable.

Methods

step	Compensate I/Q Imbalance
reset	Reset states of the `IQImbalanceCompensator` System object

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Remove I/Q Imbalance from a QPSK Signal

Mitigate the impacts of amplitude and phase imbalance on a QPSK modulated signal by using the comm.IQImbalanceCompensator System object?.

Create a constellation diagram object. Specify name-value pairs to ensure that the constellation diagram displays only the last 100 data symbols.

```
constDiagram = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',100);
```

Create an I/Q imbalance compensator.

```
iqImbComp = comm.IQImbalanceCompensator;
```

Generate random data symbols and apply QPSK modulation.

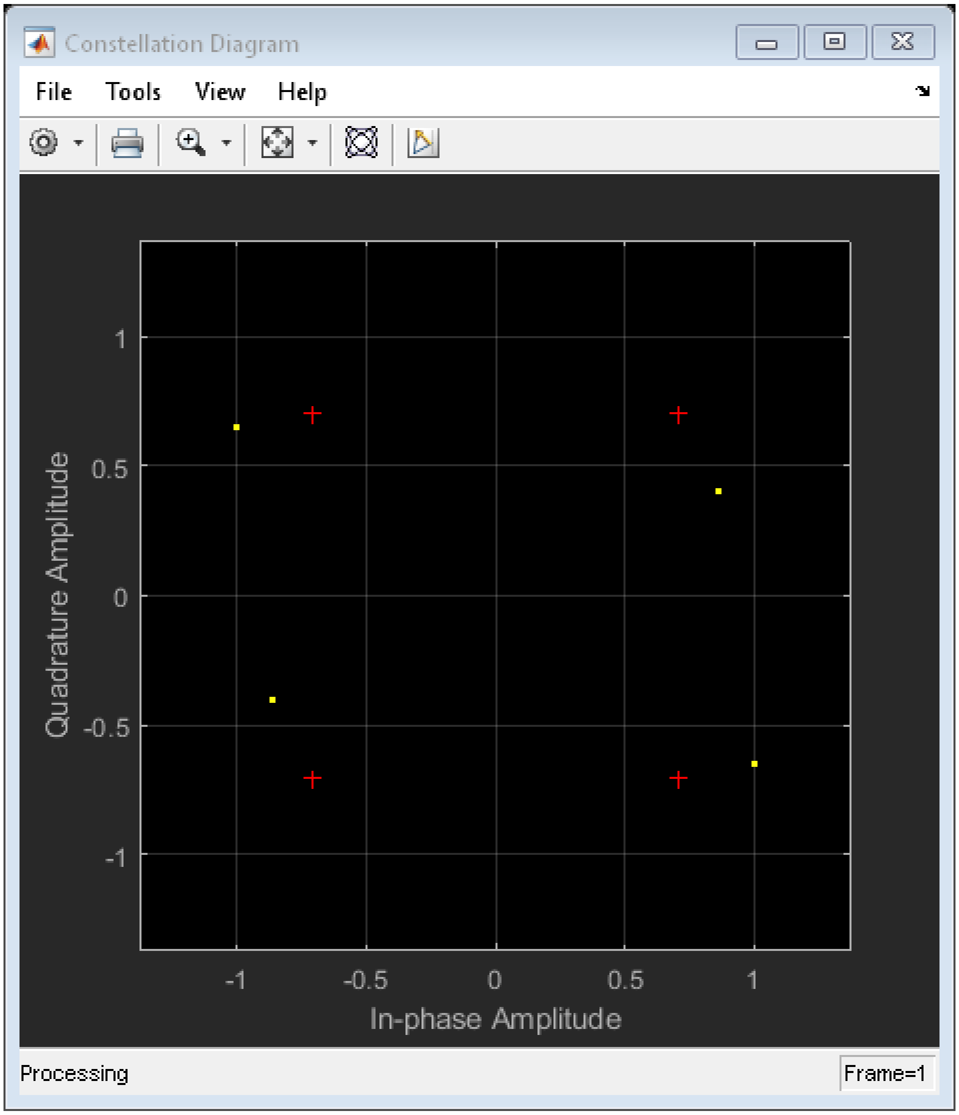
```
data = randi([0 3],1e7,1);
txSig = pskmod(data,4,pi/4);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

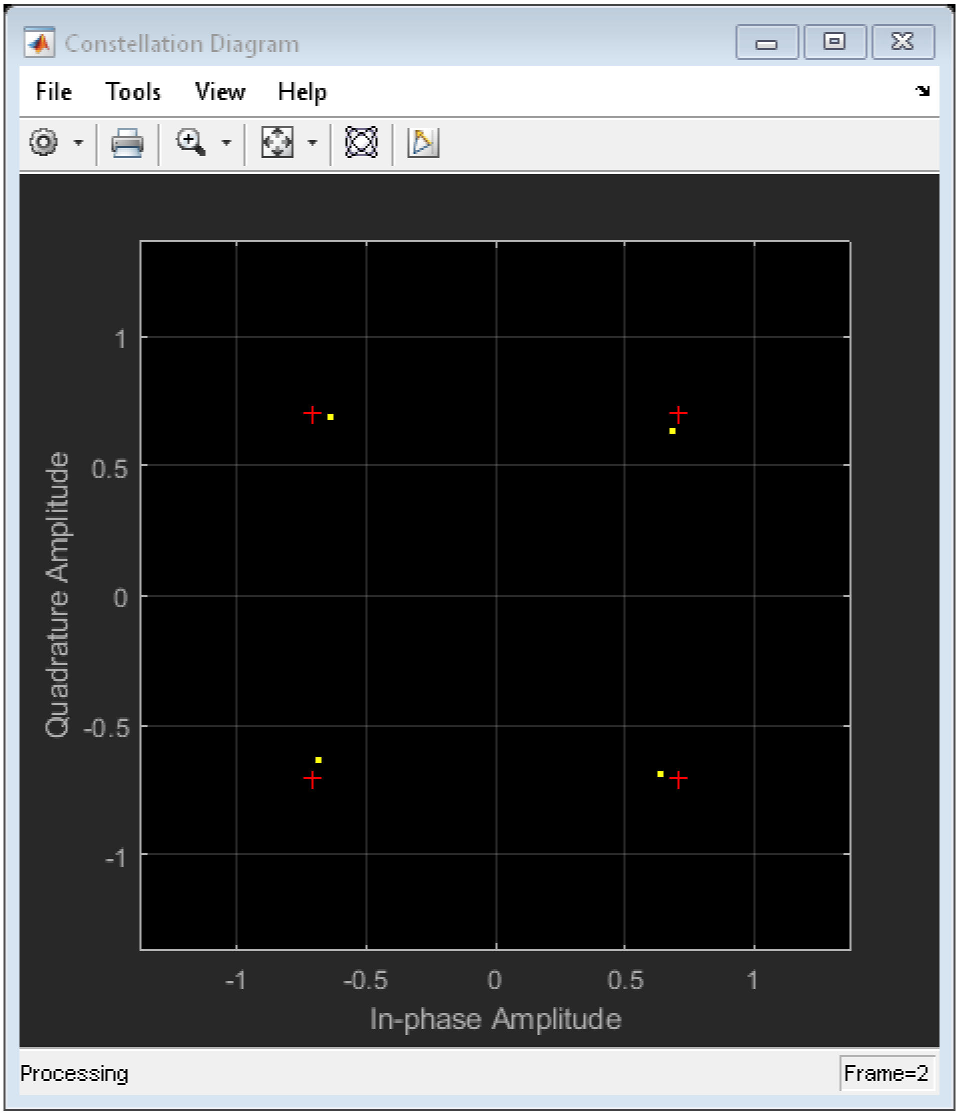
Plot the constellation diagram of the received signal. Observe that the received signal experienced an amplitude and phase shift.

```
constDiagram(rxSig)
```



Apply the I/Q compensation algorithm and view the constellation. The compensated signal constellation is nearly aligned with the reference constellation.

```
compSig = iqImbComp(rxSig);
constDiagram(compSig)
```



Remove I/Q Imbalance from an 8-PSK Signal using External Coefficients

Compensate for an amplitude and phase imbalance on an 8-PSK signal by using the `comm.IQImbalanceCompensator` System object™ with external coefficients.

Create 8-PSK modulator and constellation diagram System objects. Use name-value pairs to ensure that the constellation diagram displays only the last 100 data symbols and to provide the reference constellation.

```
hMod = comm.PSKModulator(8);
refC = constellation(hMod);
hScope = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',100, ...
    'ReferenceConstellation',refC);
```

Create an I/Q imbalance compensator object with an input port for the algorithm coefficients.

```
hIQComp = comm.IQImbalanceCompensator('CoefficientSource','Input port');
```

Generate random data symbols and apply 8-PSK modulation.

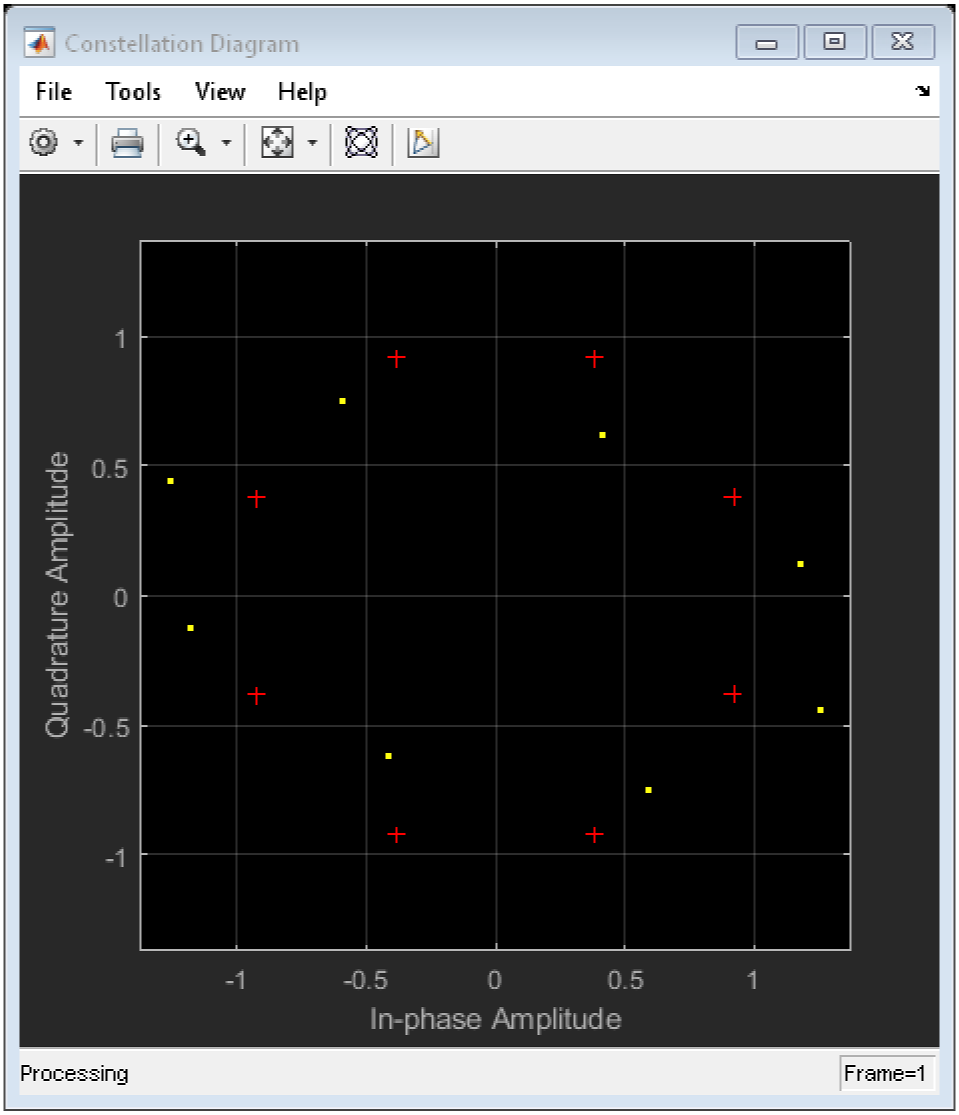
```
data = randi([0 7],1000,1);
txSig = step(hMod,data);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the constellation diagram of the received signal. Observe that the received signal experienced an amplitude and phase shift.

```
step(hScope,rxSig);
```

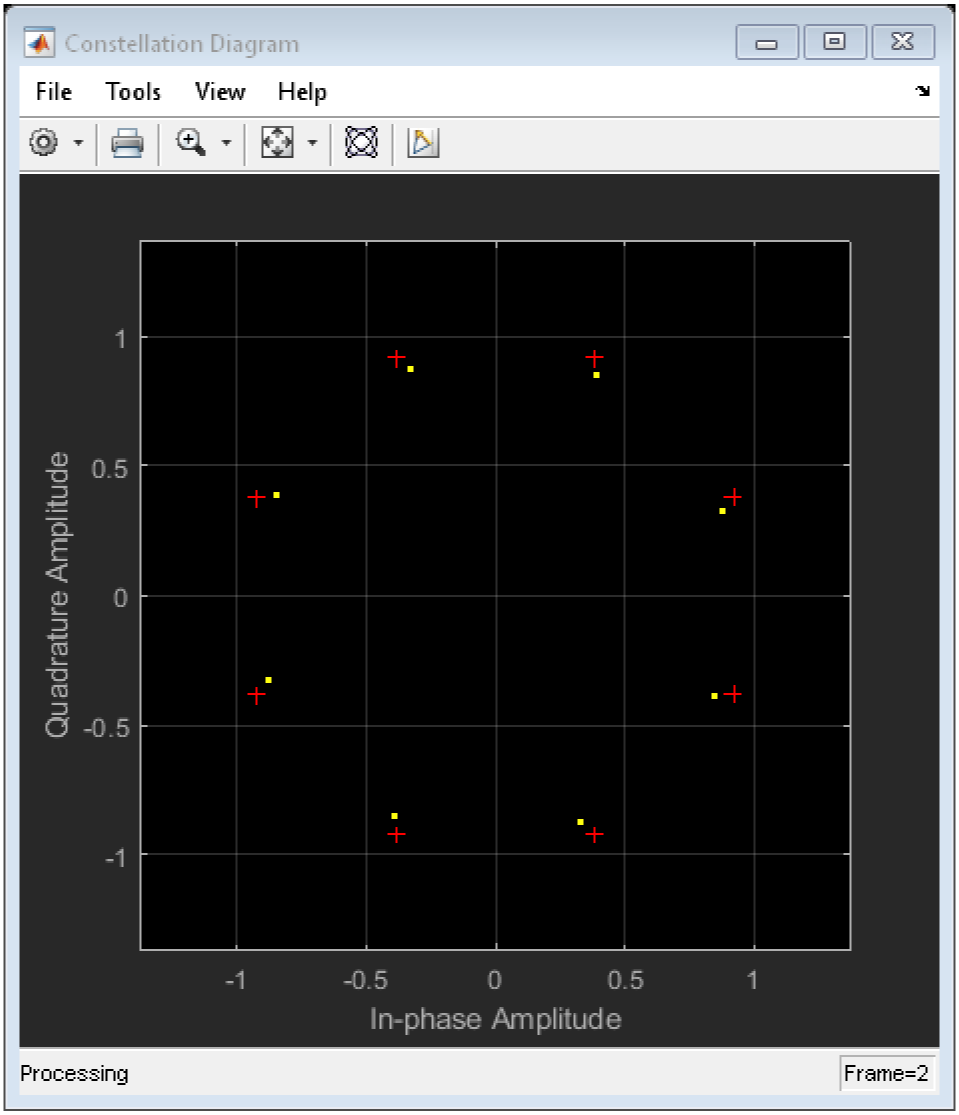


Use the `iqimbal2coef` function to determine the compensation coefficient given the amplitude and phase imbalance.

```
compCoef = iqimbal2coef(ampImb,phImb);
```

Apply the compensation coefficient to the received signal using the `step` function of the `comm.IQImbalanceCompensator` object and view the resultant constellation. You can see that the compensated signal constellation is now nearly aligned with the reference constellation.

```
compSig = step(hIQComp,rxSig,compCoef);
step(hScope,compSig)
```



Remove I/Q Imbalance from a QAM Signal

Remove an I/Q imbalance from a 64-QAM signal and to make the estimated coefficients externally available while setting the algorithm step size from an input port.

Create a constellation diagram object. Use name-value pairs to ensure that the constellation diagram displays only the last 256 data symbols, set the axes limits, and specify the reference constellation.

```
M = 64;
refC = qammod(0:M-1,M);
constDiagram = comm.ConstellationDiagram(...
    'SymbolsToDisplaySource','Property', ...
    'SymbolsToDisplay',256, ...
    'XLimits',[-10 10],'YLimits',[-10 10], ...
    'ReferenceConstellation',refC);
```

Create an I/Q imbalance compensator System object in which the step size is specified as an input argument and the estimated coefficients are made available through an output port.

```
iqImbComp = comm.IQImbalanceCompensator('StepSizeSource','Input port', ...
    'CoefficientOutputPort',true);
```

Generate random data symbols and apply 64-QAM modulation.

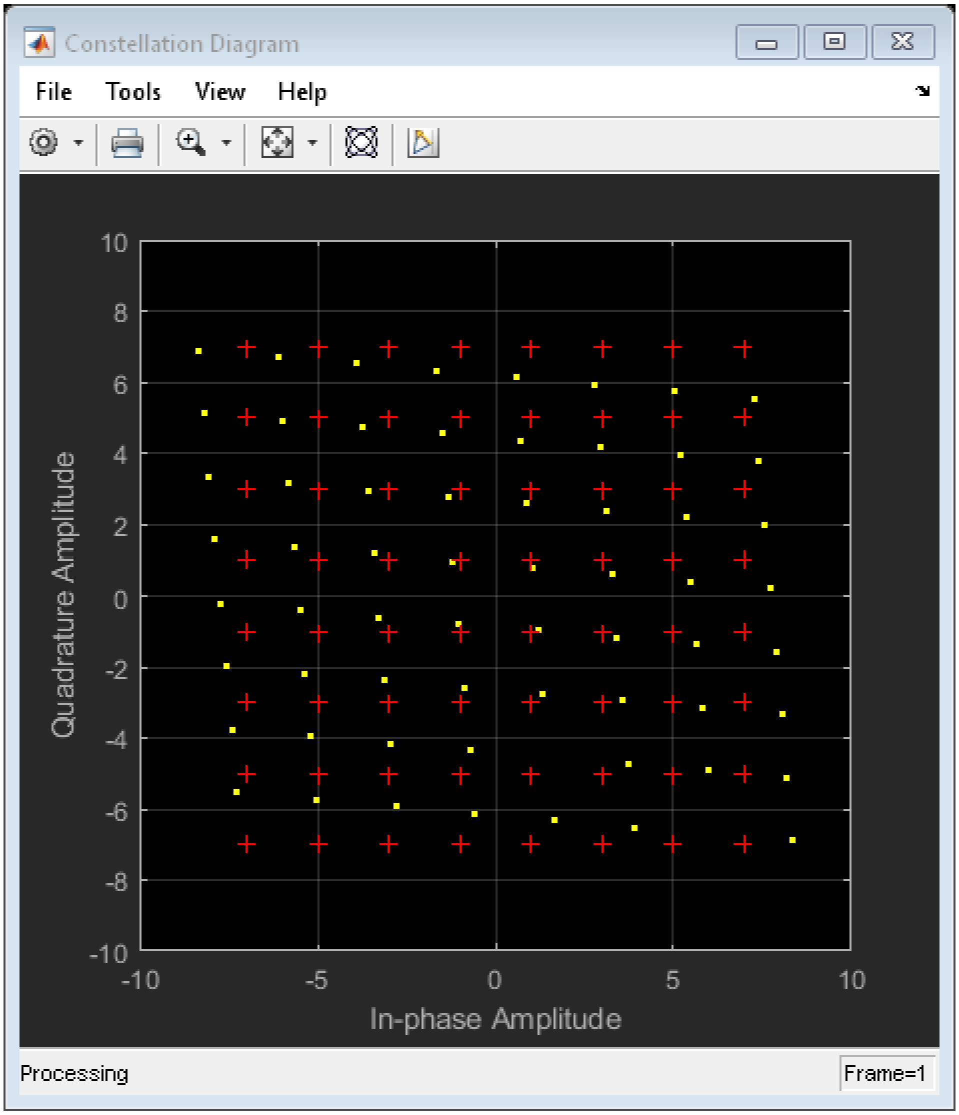
```
nSym = 25000;
data = randi([0 M-1],nSym,1);
txSig = qammod(data,M);
```

Apply amplitude and phase imbalance to the transmitted signal.

```
ampImb = 2; % dB
phImb = 10; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Plot the constellation diagram of the received signal.

```
constDiagram(rxSig);
```

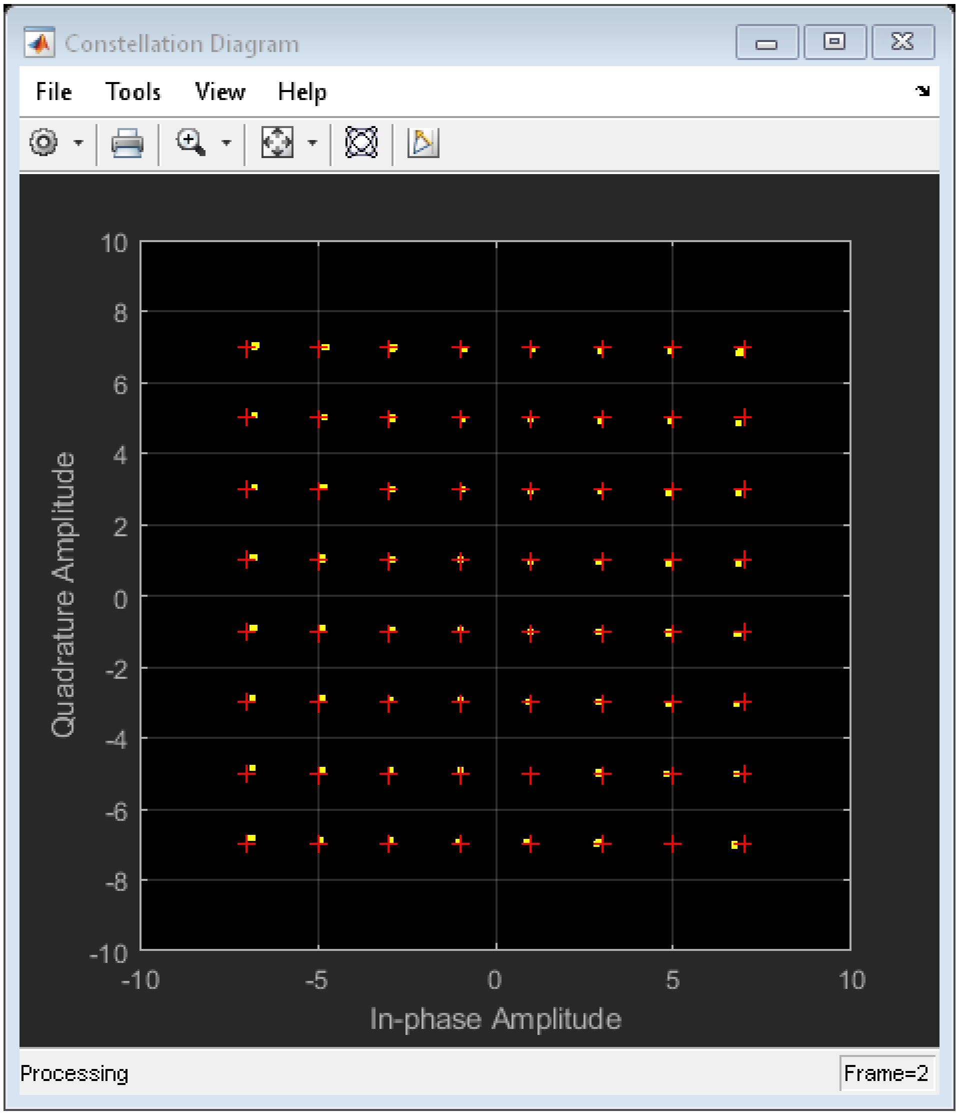


Specify the step size parameter for the I/Q imbalance compensator.

```
stepSize = 1e-5;
```

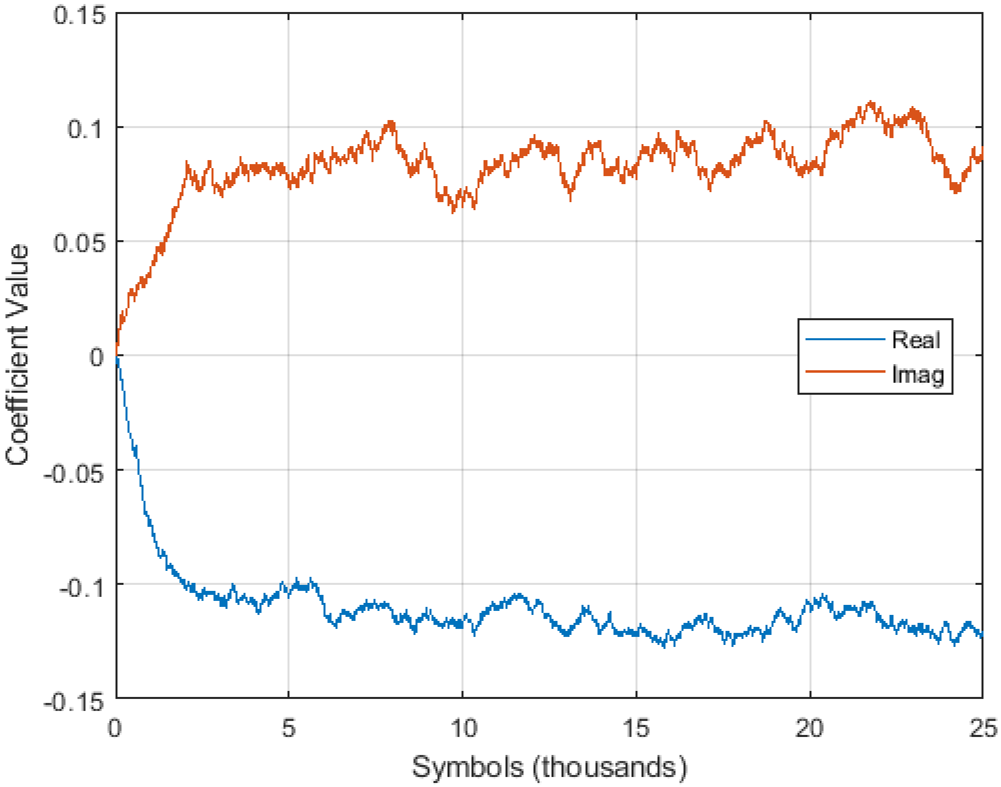
Compensate for the I/Q imbalance while setting the step size via an input argument. You can see that the compensated signal constellation is now nearly aligned with the reference constellation.

```
[compSig,estCoef] = iqImbComp(rxSig,stepSize);
constDiagram(compSig)
```



Plot the real and imaginary values of the estimated coefficients. You can see that they reach a steady-state solution.

```
plot((1:nSym)'/1000,[real(estCoef),imag(estCoef)])
grid
xlabel('Symbols (thousands)')
ylabel('Coefficient Value')
legend('Real','Imag','location','best')
```



Control Adaptation Algorithm for I/Q Imbalance Compensator

Control the adaptation algorithm of the I/Q imbalance compensator using an external argument.

Apply QPSK modulation to random data symbols.

```
data = randi([0 3],600,1);
txSig = pskmod(data,4,pi/4,'gray');
```

Create an I/Q imbalance compensator in which the adaptation algorithm is controlled through an input port, the step size is specified through the `StepSize` property, and the estimated coefficients are made available through an output port.

```
iqImbComp = comm.IQImbalanceCompensator('AdaptInputPort',true, ...
    'StepSize',0.001,'CoefficientOutputPort',true);
```

Apply amplitude and phase imbalance to the transmitted signal.

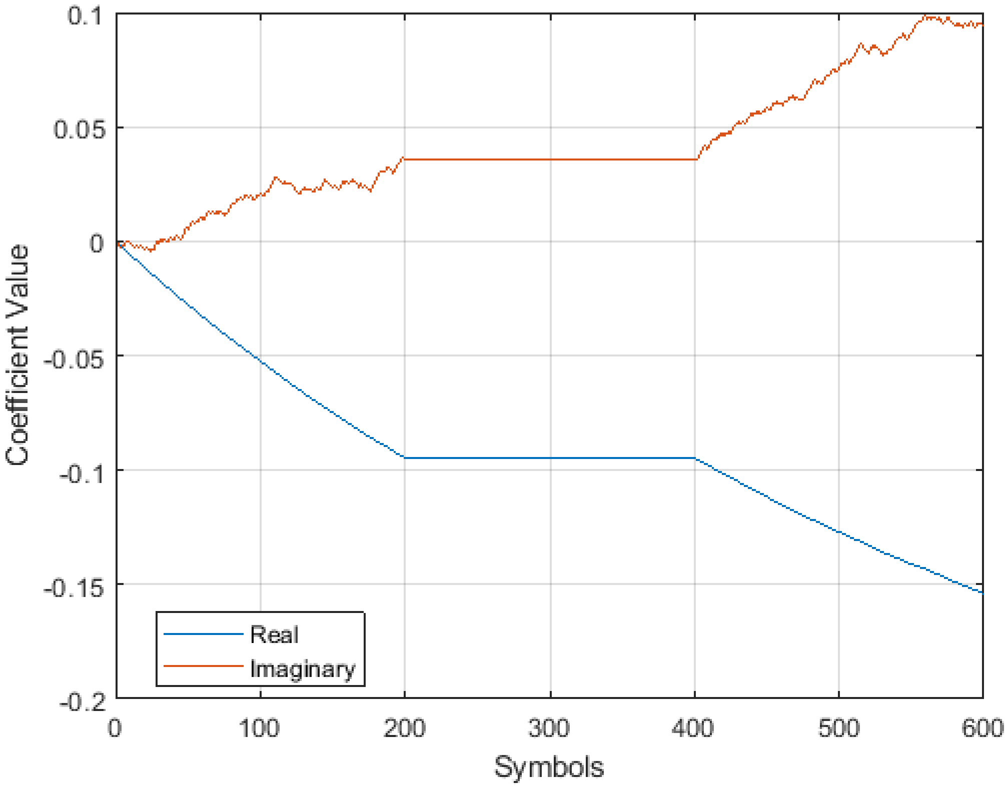
```
ampImb = 5; % dB
phImb = 15; % deg
gainI = 10.^(0.5*ampImb/20);
gainQ = 10.^(-0.5*ampImb/20);
imbI = real(txSig)*gainI*exp(-0.5i*phImb*pi/180);
imbQ = imag(txSig)*gainQ*exp(1i*(pi/2 + 0.5*phImb*pi/180));
rxSig = imbI + imbQ;
```

Break the compensation operation into three segments in which the compensator is enabled for the first 200 symbols, disabled for the next 200 symbols, and enabled for the last 200 symbols. Save the coefficient data in three vectors.

```
[~,estCoef1] = iqImbComp(rxSig(1:200),true);
[~,estCoef2] = iqImbComp(rxSig(201:400),false);
[~,estCoef3] = iqImbComp(rxSig(401:600),true);
```

Concatenate the complex algorithm coefficients and plot their real and imaginary parts.

```
estCoef = [estCoef1; estCoef2; estCoef3];
plot((1:600)',[real(estCoef) imag(estCoef)])
grid
xlabel('Symbols')
ylabel('Coefficient Value')
legend('Real','Imaginary','location','best')
```

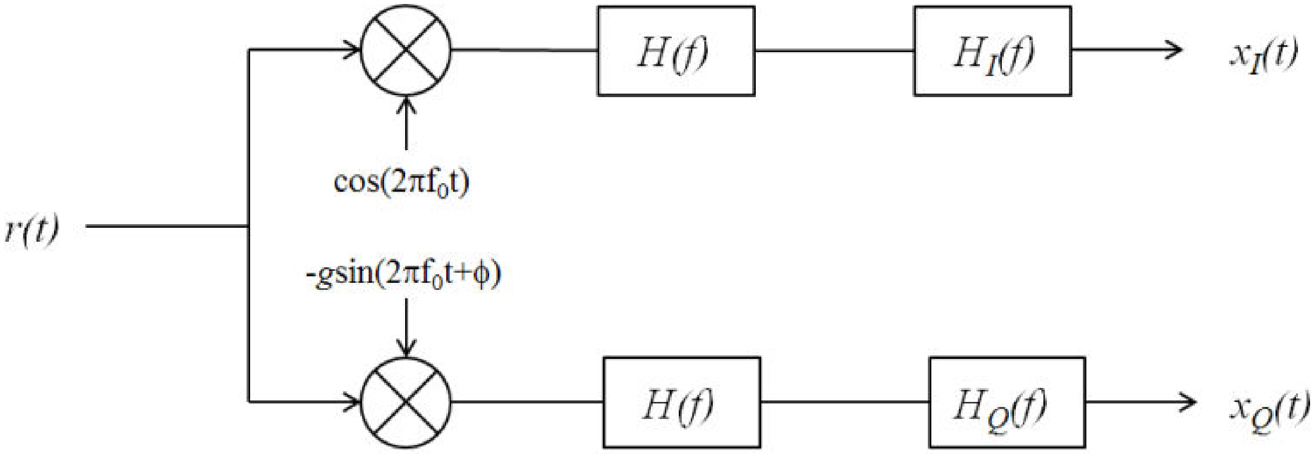


Observe that the coefficients do not adapt during the time in which the compensator is disabled.

Algorithms

One of the major impairments affecting direct conversion receivers is the imbalance between the received signal’s in-phase and quadrature components. Rather than improving the front-end, analog hardware, it is more cost effective to tolerate a certain level of I/Q imbalance and then implement compensation methods. A circularity-based blind compensation algorithm is used as the basis for the I/Q Imbalance Compensator.

A generalized I/Q imbalance model is shown, where g is the amplitude imbalance and ϕ is the phase imbalance (ideally, $g = 1$ and $\phi = 0$). In the figure, $H(f)$ is the nominal frequency response of the branches due to, for example, lowpass filters. $H_I(f)$ and $H_Q(f)$ represent the portions of the in-phase and quadrature amplitude and phase responses that differ from the nominal response. With perfect matching, $H_I(f) = H_Q(f) = 1$.



Let $z(t)$ be the ideal baseband equivalent signal of the received signal, $r(t)$, where its Fourier transform is denoted as $Z(f)$. Given the generalized I/Q imbalance model, the Fourier transform of the imbalanced signal, $x(t) = x_I(t) + x_Q(t)$, is

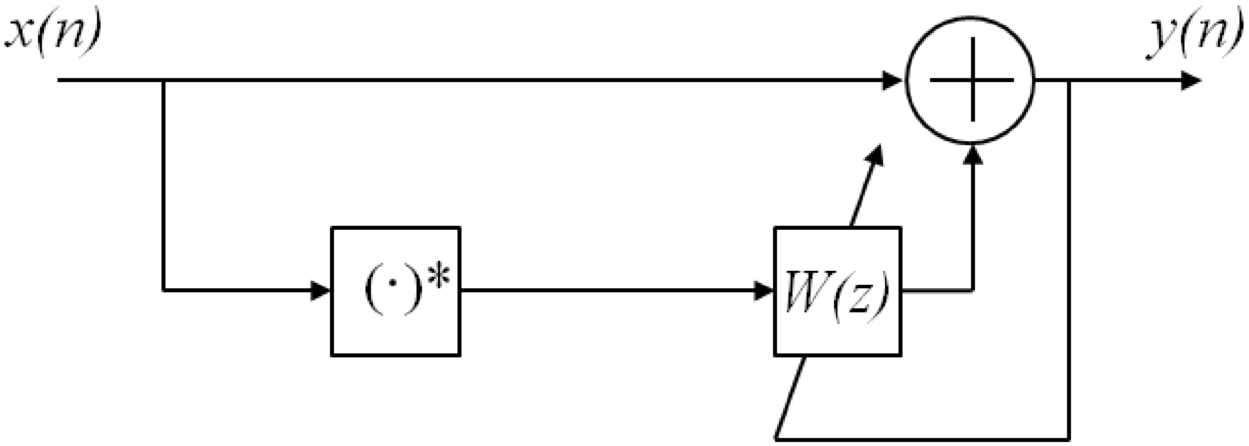
$$X(f) = G_1(f)Z(f) + G_2(f)Z^*(-f)$$

where $G_1(f)$ and $G_2(f)$ are the direct and conjugate components of the I/Q imbalance. These components are defined as

$$G_1(f) = \left[H_I(f) + H_Q(f)g\exp(-j\phi)\right]/2$$
$$G_2(f) = \left[H_I(f) + H_Q(f)g\exp(j\phi)\right]/2$$

Applying the inverse Fourier transform to $X(f)$, the signal model becomes $x(t) = g_1(t) * z(t) + g_2(t) * z^*(t)$.

This suggests the compensator structure as shown in which discrete-time notation is used to express the variables. The compensated signal is expressed as y(n) = $x(n) + wx^*(n)$.



A simple algorithm of the form

$$\begin{cases} y(n) = x(n) + w(n)x^*(n) \\ w(n+1) = w(n) - My^2(n) \end{cases}$$

is used to determine the weights, because it ensures that the output is “proper”, that is, $E[y^2(n)] = 0$ [1]. The initial value of w is determined by the `InitialCoefficient` property, which has a default value of `0 + 0i`. M is the step size, as specified in the `StepSize` property.

Selected Bibliography

[1] Anttila, L., M. Valkama, and M. Renfors. “Blind compensation of frequency-selective I/Q imbalances in quadrature radio receivers: Circularity-based approach”, *Proc. IEEE ICASSP*, pp.III-245–248, 2007.

[2] Kiayani, A., L. Anttila, Y. Zou, and M. Valkama, “Advanced Receiver Design for Mitigating Multiple RF Impairments in OFDM Systems: Algorithms and RF Measurements”, *Journal of Electrical and Computer Engineering*, Vol. 2012.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

I/Q Imbalance Compensator | `iqcoef2imbal` | `iqimbal2coef`

Introduced in R2014b

step

System object: `comm.IQImbalanceCompensator`
Package: `comm`

Compensate I/Q Imbalance

Syntax

```
Y = step(H,X)
Y = step(H,X,COEF)
Y = step(H,X,STEPSIZE)
Y = step(H,...,ADAPT)
[Y,ESTCOEF] = step(H,X)
[Y,ESTCOEF] = step(H,X,STEPSIZE)
[Y,ESTCOEF] = step(H,X,STEPSIZE,ADAPT)
[Y,ESTCOEF] = step(H,X,ADAPT)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the I/Q imbalance in the input signal, `X`, and returns a compensated signal, `Y`. The input `X` can take real or complex values and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` has the same properties as `X`.

`Y = step(H,X,COEF)` accepts input coefficients, `COEF`, instead of generating them internally. This syntax applies when the CoefficientSource property of `H` is set to `Input port`. The input coefficients, `COEF`, are complex and can be either double or single precision. `COEF` has the same dimensions as `X`. At each time instant, `COEF` is a complex scalar.

`Y = step(H,X,STEPSIZE)` accepts a step size input, `STEPSIZE`. This syntax applies when the StepSizeSource property of `H` is set to `Input port`. The step size is a real scalar supporting either double or single precision.

`Y = step(H,...,ADAPT)` accepts a control signal, `ADAPT`, to enable or disable coefficient updates. This syntax applies when the AdaptInputPort property of `H` is `true`. The adaptation control signal is a logical scalar.

`[Y,ESTCOEF] = step(H,X)` outputs the estimated coefficients, `ESTCOEF`, when the CoefficientOutputPort property of `H` is `true`. `ESTCOEF` has the same data properties and dimensionality as the input signal, `X`.

`[Y,ESTCOEF] = step(H,X,STEPSIZE)` outputs the estimated coefficients, `ESTCOEF`, and accepts a step size input, `STEPSIZE`. This syntax applies when the properties of `H` are set so that CoefficientOutputPort is `true` and StepSizeSource is `Input port`.

[Y,ESTCOEF] = step(H,X,STEPSIZE,ADAPT) outputs the estimated coefficients, ESTCOEF, and accepts a step size input,STEPSIZE, and a control signal input, ADAPT. This syntax applies when the properties of H are set so that CoefficientOutputPort is true, StepSizeSource is Input port, and AdaptInputPort is true.

[Y,ESTCOEF] = step(H,X,ADAPT) outputs the estimated coefficients, ESTCOEF, and accepts a control signal input, ADAPT. This syntax applies when the properties of H are set so that CoefficientOutputPort is true and AdaptInputPort is true.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

reset

System object: `comm.IQImbalanceCompensator`
Package: `comm`

Reset states of the `IQImbalanceCompensator` System object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `IQImbalanceCompensator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

comm.BPSKModulator

Package: `comm`

Modulate using BPSK method

Description

The `BPSKModulator` object modulates using the binary phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a binary phase shift signal:

1. Create the `comm.BPSKModulator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
bpskmod = comm.BPSKModulator
bpskmod = comm.BPSKModulator(Name,Value)
bpskmod = comm.BPSKModulator(phase,Name,Value)
```

Description

`bpskmod = comm.BPSKModulator` creates a modulator System object `bpskmod`, that modulates the input signal using the binary phase shift keying (BPSK) method.

`bpskmod = comm.BPSKModulator(Name,Value)` creates a BPSK modulator object `bpskmod`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`bpskmod = comm.BPSKModulator(phase,Name,Value)` creates a BPSK modulator object `bpskmod`. The object's `PhaseOffset` property is set to `phase`, and the other specified properties are set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`PhaseOffset` — Phase of zeroth point of constellation
`0` (default) | finite real scalar

Phase of zeroth point of constellation in radians, specified as a finite real scalar.

`OutputDataType` — Output datatype
`double` (default) | `single` | `Custom`

Output datatype, specified as either `double`, `single` or `Custom`.

Fixed-Point Properties

`CustomOutputDataType` — Fixed-point data type of output
`numerictype([],16)` (default) | `numerictype` object

Fixed-point data type of output, specified as a `numerictype` object with a signedness of Auto.

Dependencies

This property applies when you set the `OutputDataType` property to `'Custom'`.

Usage

Syntax

```
waveform = bpskmod(data)
```

Description

`waveform = bpskmod(data)` applies BPSK modulation to the input data and returns the modulated BPSK baseband signal.

Input Arguments

`data` — Input data
column vector | matrix

Input data, specified as a column vector or matrix.

Data Types: `double`

Output Arguments

`waveform` — BPSK Modulated baseband signal
column vector | matrix

BPSK Modulated baseband signal, returned as a column vector or matrix of the same size as the input signal.

Data Types: `double`
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.BPSKModulator`

constellation	Calculate or plot ideal signal constellation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

BPSK Data Scatter Plot

This example creates binary data, modulates it, and then displays the data using a scatter plot.

Create binary data symbols

```
data = randi([0 1],100,1);
```

Create a BPSK modulator System object

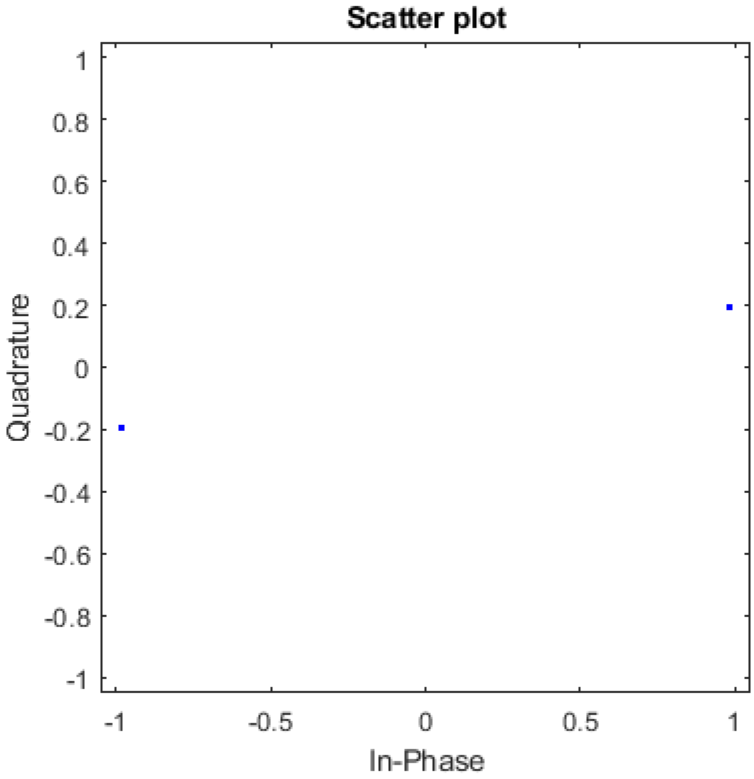
```
bpskModulator = comm.BPSKModulator;
```

Change the phase offset to pi/16

```
bpskModulator.PhaseOffset = pi/16;
```

Modulate and plot the data

```
modData = bpskModulator(data);
scatterplot(modData)
```



Algorithms

Phase modulation is a linear baseband modulation technique in which the message modulates the phase of a constant amplitude signal. Binary Phase Shift Keying (BPSK) is a two phase modulation scheme, where the 0’s and 1’s in a binary message are represented by two different phase states in the carrier signal

$$s_n(t) = \sqrt{\frac{2E_b}{T_b}}\cos(2\pi f_c t + \pi(1 - n));\ n \in \left\{0, 1\right\},$$

for $(n - 1)T_b \le t \le nT_b,\ \text{n} = 1, 2,\ 3, \ldots$ where:

- $\phi_n = \pi m,\ m\in\{0,1\}$.
- E_b is the energy per bit.
- T_b is the bit duration.
- f_c is the carrier frequency.

In MATLAB, the baseband representation of a BPSK signal is

$$s_n(t) = \cos(\pi n);\ n \in \{0, 1\}\ .$$

The BPSK signal has two phases: 0 and π. The probability of a bit error in an AWGN channel is

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right),$$

where N_0 is the noise power spectral density.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.BPSKDemodulator` | `comm.PSKModulator`

Blocks

BPSK Modulator Baseband

Introduced in R2012a

constellation

Package: `comm`

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(bpskmod)
constellation(bpskmod)
```

Description

`y = constellation(bpskmod)` returns the numerical values of the constellation.

`constellation(bpskmod)` generates a constellation plot for the object.

Examples

Calculate BPSK Modulator Reference Constellation

Create a BPSK Modulator System object™ and calculate the reference constellation values.

Create a `comm.BPSKModulator` System object.

```
h = comm.BPSKModulator;
```

Calculate and display the reference constellation values by calling the `constellation` function.

```
refC = constellation(h)
```

```
refC = 2×1 complex

   1.0000 + 0.0000i
  -1.0000 + 0.0000i
```

Plot BPSK Modulator Reference Constellation

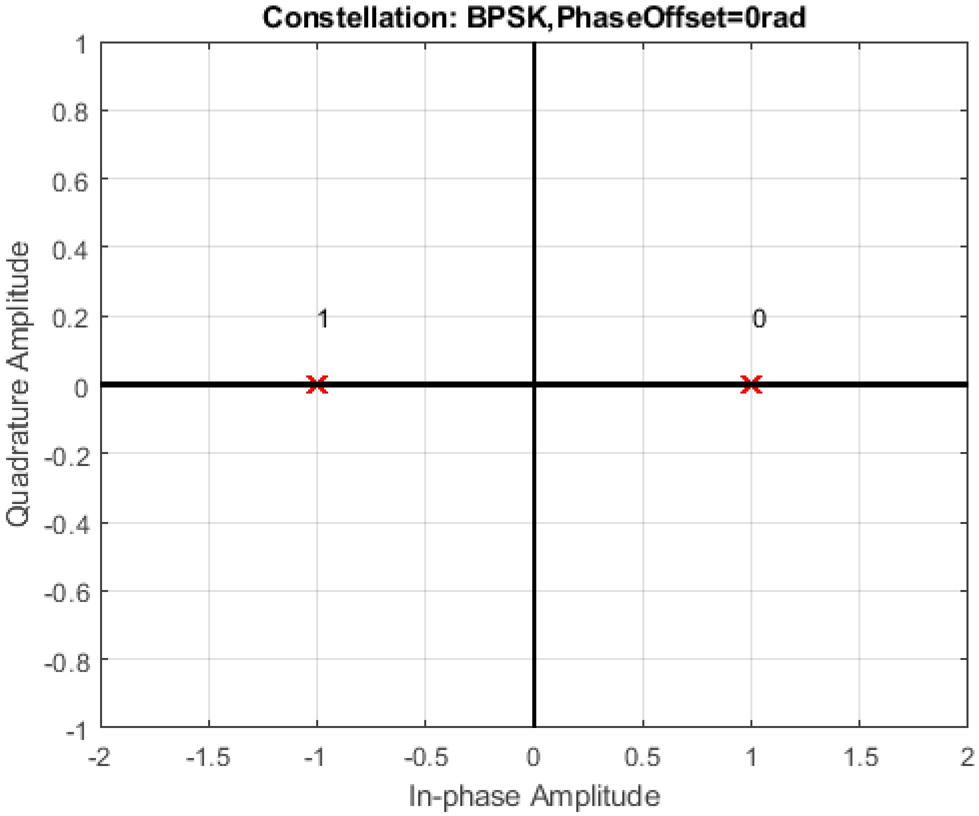
Create a BPSK Modulator System object™ and plot the reference constellation.

Create a `comm.BPSKModulator` System object.

```
bpsk = comm.BPSKModulator;
```

Plot the reference constellation by calling the `constellation` function.

```
constellation(bpsk)
```



Input Arguments

bpskmod — BPSK Modulator
System object

BPSK Modulator, specified as a `comm.BPSKModulator` System object.

See Also

Objects
`comm.BPSKModulator`

Introduced in R2012a

step

System object: `comm.BPSKModulator`
Package: `comm`

Modulate using BPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the BPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be a column vector of bits. The data type of the input can be numeric, logical, or unsigned fixed point of word length `1` (fi object).

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

comm.OFDMModulator

Package: `comm`

Modulate signal using OFDM method

Description

The `OFDMModulator` object modulates a signal using the orthogonal frequency division modulation method. The output is a baseband representation of the modulated signal.

To modulate a signal using OFDM:

1. Create the `comm.OFDMModulator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
hMod = comm.OFDMModulator
hMod = comm.OFDMModulator(Name,Value)
hMod = comm.OFDMModulator(hDemod)
```

Description

`hMod = comm.OFDMModulator` creates an OFDM modulator System object.

`hMod = comm.OFDMModulator(Name,Value)` specifies “Properties” on page 3-233 using one of more name-value pair arguments. Enclose each property name in quotes. For example, `comm.OFDMModulator('NumSymbols',8)` specifies eight OFDM symbols in the time-frequency grid.

`hMod = comm.OFDMModulator(hDemod)` sets the OFDM modulator system object properties based on the specified OFDM demodulator system object `comm.OFDMDemodulator`.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`FFTLength` — Number of FFT points
`64` (default) | positive integer

Number of Fast Fourier Transform (FFT) points, specified as a positive integer. The length of the FFT, N_{FFT}, must be greater than or equal to 8 and is equivalent to the number of subcarriers.

Data Types: `double`

`NumGuardBandCarriers` — Number of subcarriers to the left and right guard bands

`[6;5]` (default) | two-element column vector of integers

Number of subcarriers allocated to the left and right guard bands, specified as a two-element column vector of integers. The number of subcarriers must fall within $[0, \lfloor N_{\mathrm{FFT}}/2\rfloor - 1]$. This vector has the form $[N_{\mathrm{leftG}}, N_{\mathrm{rightG}}]$, where N_{leftG} and N_{rightG} specify the left and right guard bands, respectively.

Data Types: `double`

`InsertDCNull` — Option to insert DC null

`false` or `0` (default) | `true` or `1`

Option to insert DC null, specified as a numeric or logical `0` (`false`) or `1` (`true`). The DC subcarrier is the center of the frequency band and has the index value:

- (`FFTLength` / 2) + 1 when `FFTLength` is even
- (`FFTLength` + 1) / 2 when `FFTLength` is odd

`PilotInputPort` — Option to specify pilot input

`false` or `0` (default) | `true` or `1`

Option to specify pilot input, specified as a numeric or logical `0` (`false`) or `1` (`true`). If this property is `1` (`true`), you can assign individual subcarriers for pilot transmission. If this property is `0` (`false`), pilot information is assumed to be embedded in the input data.

`PilotCarrierIndices` — Pilot subcarrier indices

`[12; 26; 40; 54]` (default) | column vector

Pilot subcarrier indices, specified as a column vector. If the PilotCarrierIndices property is set to `1` (`true`), you can specify the indices of the pilot subcarriers. You can assign the indices to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the property vary. Valid pilot indices fall in the range

$$[N_{\mathrm{leftG}} + 1,\ N_{\mathrm{FFT}}/2] \cup \left[N_{\mathrm{FFT}}/2 + 2,\ N_{\mathrm{FFT}} - N_{\mathrm{rightG}}\right],$$

where the index value cannot exceed the number of subcarriers. When the pilot indices are the same for every symbol and transmit antenna, the property has dimensions N_{pilot}-by-1. When the pilot indices vary across symbols, the property has dimensions N_{pilot}-by-N_{sym}. If you transmit only one symbol but multiple transmit antennas, the property has dimensions N_{pilot}-by-1-by-N_{t}., where N_{t}. is the number of transmit antennas. If the indices vary across the number of symbols and transmit antennas, the property has dimensions N_{pilot}-by-N_{sym}-by-N_{t}. If the number of transmit antennas is greater than one, ensure that the indices per symbol must be mutually distinct across antennas to minimize interference.

To enable this property, set the `PilotInputPort` property to `1` (`true`).

`CyclicPrefixLength` — Length of cyclic prefix

`16` (default) | positive integer | row vector

Length of cyclic prefix, specified as a positive integer. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same through all antennas.

Data Types: `double`

`Windowing` — Option to apply raised cosine window between OFDM symbols

`false` or `0` (default) | `true` or `1`

Option to apply raised cosine window between OFDM symbols, specified as `true` or `false`. Windowing is the process in which the OFDM symbol is multiplied by a raised cosine window before transmission to more quickly reduce the power of out-of-band subcarriers. Windowing reduces spectral regrowth.

`WindowLength` — Length of raised cosine window

`1` (default) | positive scalar

Length of raised cosine window, specified as a positive scalar. This value must be less than or equal to the minimum cyclic prefix length. For example, in a configuration of four symbols with cyclic prefix lengths 12, 14, 16, and 18, the window length must be less than or equal to 12.

To enable this property, set the `Windowing` property to `1` (`true`).

`NumSymbols` — Number of OFDM symbols

`1` (default) | positive integer

Number of OFDM symbols in the time-frequency grid, specified as a positive integer.

`NumTransmitAntennnas` — Number of transmit antennas

`1` (default) | positive integer

Number of transmit antennas, used to transmit the OFDM modulated signal, specified as a positive integer.

Usage

Syntax

```
waveform = hMod(insignal)
waveform = hMod(data,pilot)
```

Description

`waveform = hMod(insignal)` applies OFDM modulation the specified baseband signal and returns the modulated OFDM baseband signal.

`waveform = hMod(data,pilot)` assigns the pilot signal, `pilot`, into the frequency subcarriers specified by the PilotCarrierIndices property value of the `hMod` system object. To enable this syntax set the PilotCarrierIndices property to `true`.

Input Arguments

`insignal` — Input baseband signal

matrix | 3-D array

Input baseband signal, specified as a matrix or 3-D array of numeric values. The input baseband signal must be of size N_f-by-N_{sym}-by-N_t. where N_f is the number of frequency subcarriers excluding guard bands and DC null.

Data Types: `double`
Complex Number Support: Yes

`data` — Input data
matrix | 3-D array

Input data, specified as a matrix or 3-D array. The input must be a numeric of size N_d-by-N_{sym}-by-N_t. where N_d is the number of data subcarriers in each symbol. For more information on how N_d is calculated, see the PilotCarrierIndices property.

Data Types: `double`
Complex Number Support: Yes

`pilot` — Pilot signal
3-D array

Pilot signal, specified as a 3-D array of numeric values. The pilot signal must be of size N_{pilot}-by-N_{sym}-by-$\mathrm{N_t}$.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`waveform` — OFDM Modulated baseband signal
2-D array

OFDM Modulated baseband signal, returned as a 2-D array. If the `CyclicPrefixLength` property is a scalar, the output `waveform` is of size $((\mathrm{N_{FFT}}+\mathrm{CP_{len}})*\mathrm{N_{sym}})$-by-$\mathrm{N_t}$. Otherwise, the size is $(\mathrm{N_{FFT}}*\mathrm{N_{sym}}+\sum(\mathrm{CP_{len}}))$-by-$\mathrm{N_t}$.

Data Types: `double`
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.OFDMModulator`

info	Provide dimensioning information for OFDM modulator
showResourceMapping	Show the subcarrier mapping of the OFDM symbols created by the OFDM modulator System object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics

reset	Reset internal states of System object

Examples

Create and Modify OFDM Modulator

Create and display an OFDM modulator System object™ with default property values.

```
hMod = comm.OFDMModulator
```

```
hMod = 
  comm.OFDMModulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 1
     NumTransmitAntennas: 1
```

Modify the number of subcarriers and symbols.

```
hMod.FFTLength = 128;
hMod.NumSymbols = 2;
```

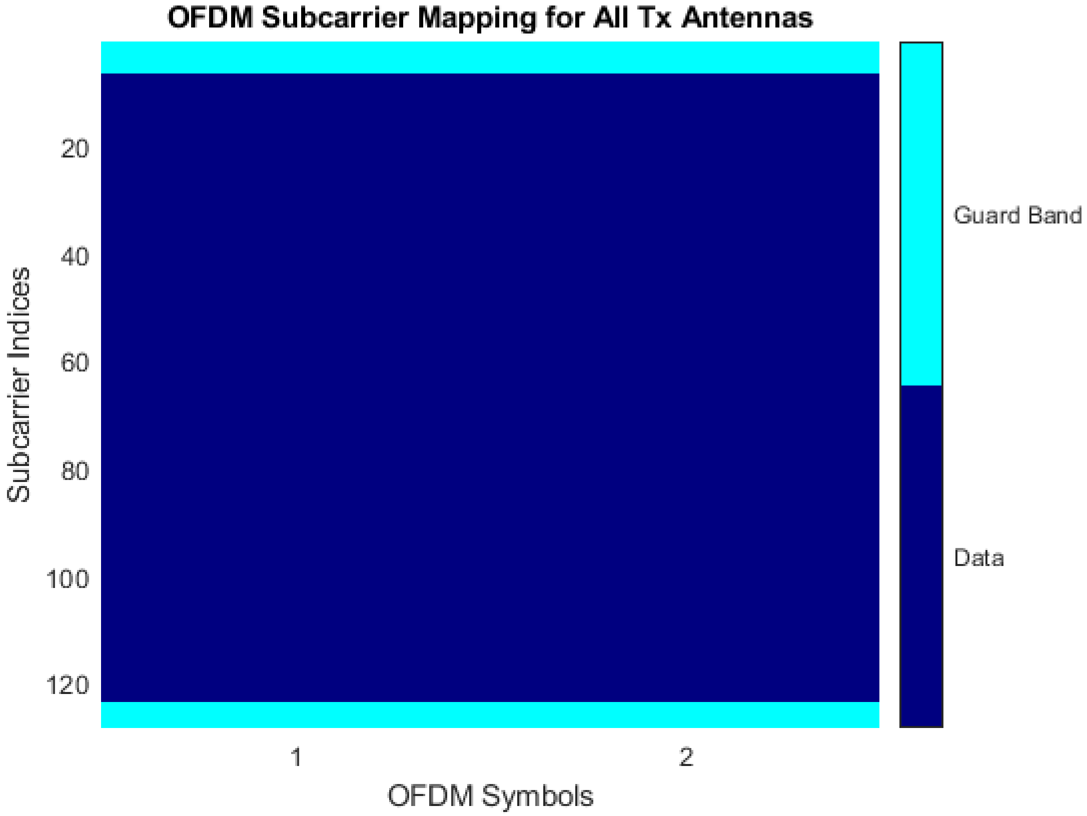
Verify that the number of subcarriers and the number of symbols changed.

```
disp(hMod)
```

```
  comm.OFDMModulator with properties:

               FFTLength: 128
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 2
     NumTransmitAntennas: 1
```

Use the `showResourceMapping` object function to show the mapping of data, pilot, and null subcarriers in the time-frequency space.

```
showResourceMapping(hMod)
```



Create OFDM Modulator from OFDM Demodulator

Create an OFDM demodulator System object™ with default property values. Then, specify pilot indices for a single symbol and two transmit antennas.

Setting the `PilotCarrierIndices` property of the demodulator affects the number of transmit antennas in the OFDM modulator when you use the demodulator in the creation of the modulator. The number of receive antennas in the demodulator is uncorrelated with the number of transmit antennas.

```
ofdmDemod = comm.OFDMDemodulator;
ofdmDemod.PilotOutputPort = true;
ofdmDemod.PilotCarrierIndices = cat(3,[12; 26; 40; 54],[13; 27; 41; 55]);
```

Use the OFDM demodulator to construct the OFDM modulator.

```
ofdmMod = comm.OFDMModulator(ofdmDemod);
```

Display the properties of the OFDM modulatorand demodulator, verifying that the applicable properties match.

```
disp(ofdmMod)
```

```
  comm.OFDMModulator with properties:
```

```
               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
          PilotInputPort: true
     PilotCarrierIndices: [4x1x2 double]
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 1
     NumTransmitAntennas: 2
```

```
disp(ofdmDemod)
```

```
  comm.OFDMDemodulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: true
     PilotCarrierIndices: [4x1x2 double]
      CyclicPrefixLength: 16
              NumSymbols: 1
      NumReceiveAntennas: 1
```

Visualize Time-Frequency Resource Assignments for OFDM Modulator

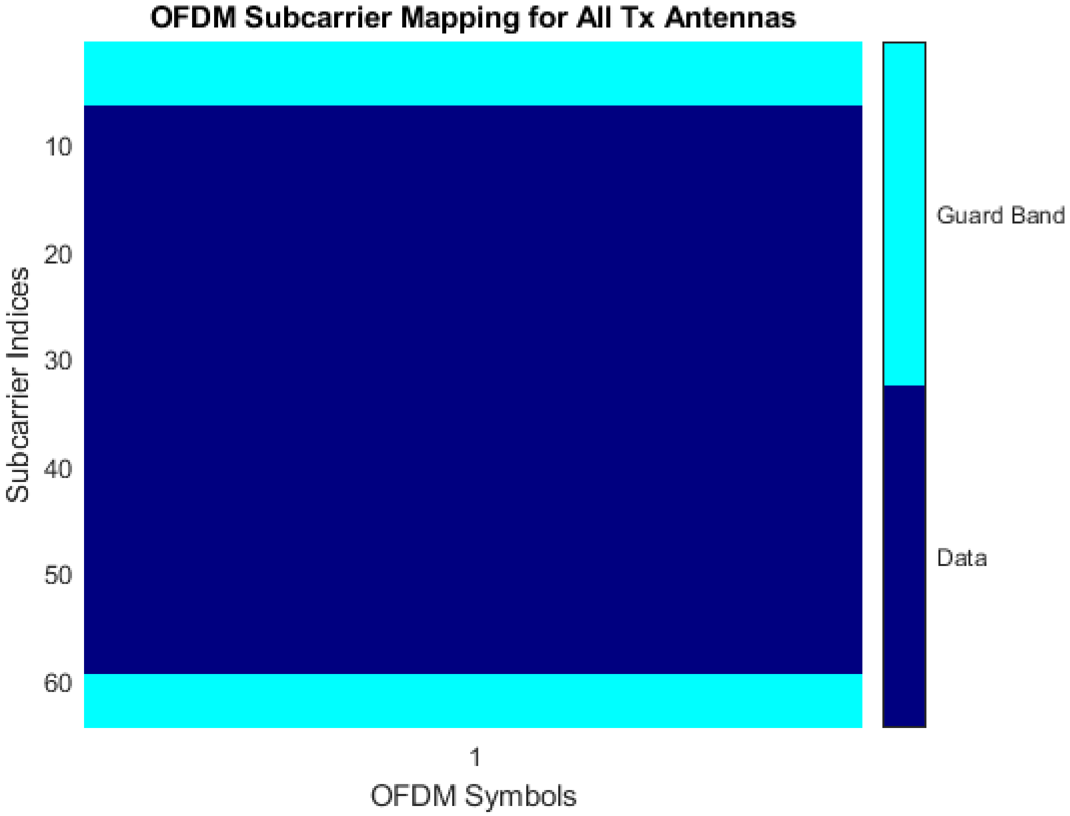
The `showResourceMapping` method displays the time-frequency resource mapping for each transmit antenna.

Construct an OFDM modulator.

```
mod = comm.OFDMModulator;
```

Apply the `showResourceMapping` method.

```
showResourceMapping(mod)
```

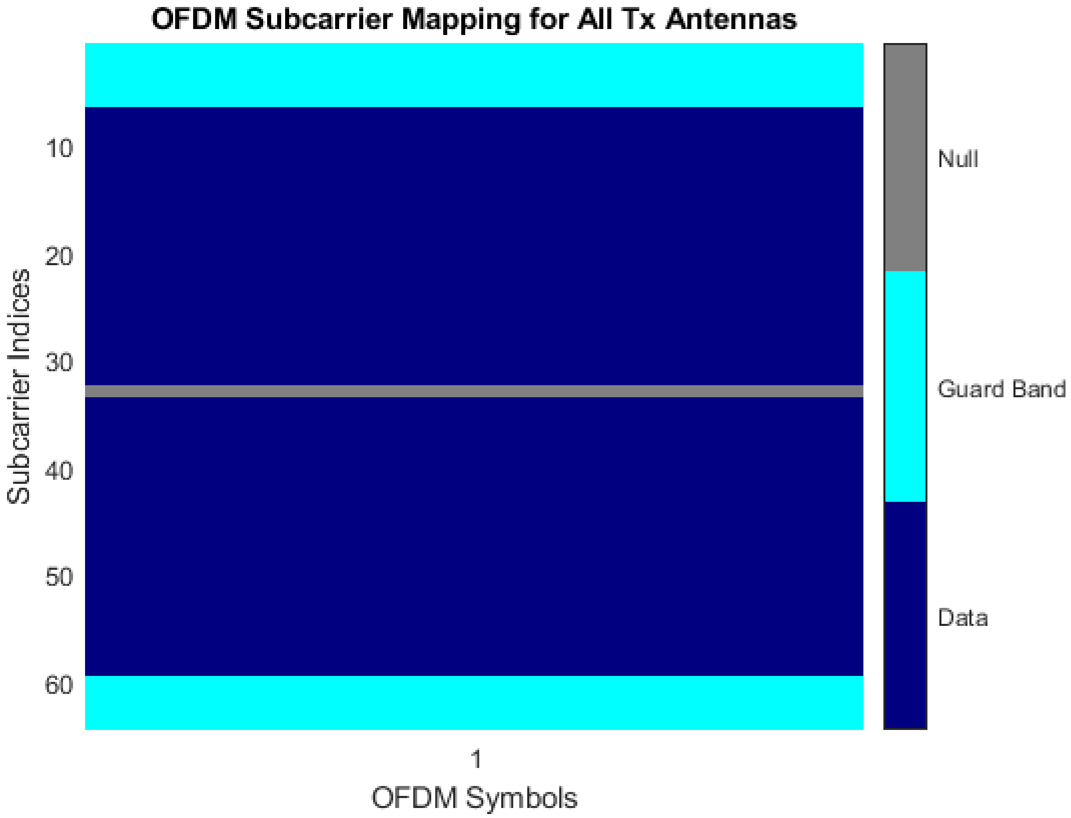


Insert a DC null.

```
mod.InsertDCNull = true;
```

Show the resource mapping after adding the DC null.

```
showResourceMapping(mod)
```



Create OFDM Modulator and Specify Pilots

Create an OFDM modulator and specify the subcarrier indices for the pilot signals. Specify the indices for each symbol and transmit antenna. When the number of transmit antennas is greater than one, set different pilot indices for each symbol between antennas.

Create an OFDM modulator System object, specifying two symbols and inserting a DC null.

```
mod = comm.OFDMModulator('FFTLength',128,'NumSymbols',2,...
    'InsertDCNull',true);
```

Enable the pilot input port so you can specify the pilot indices.

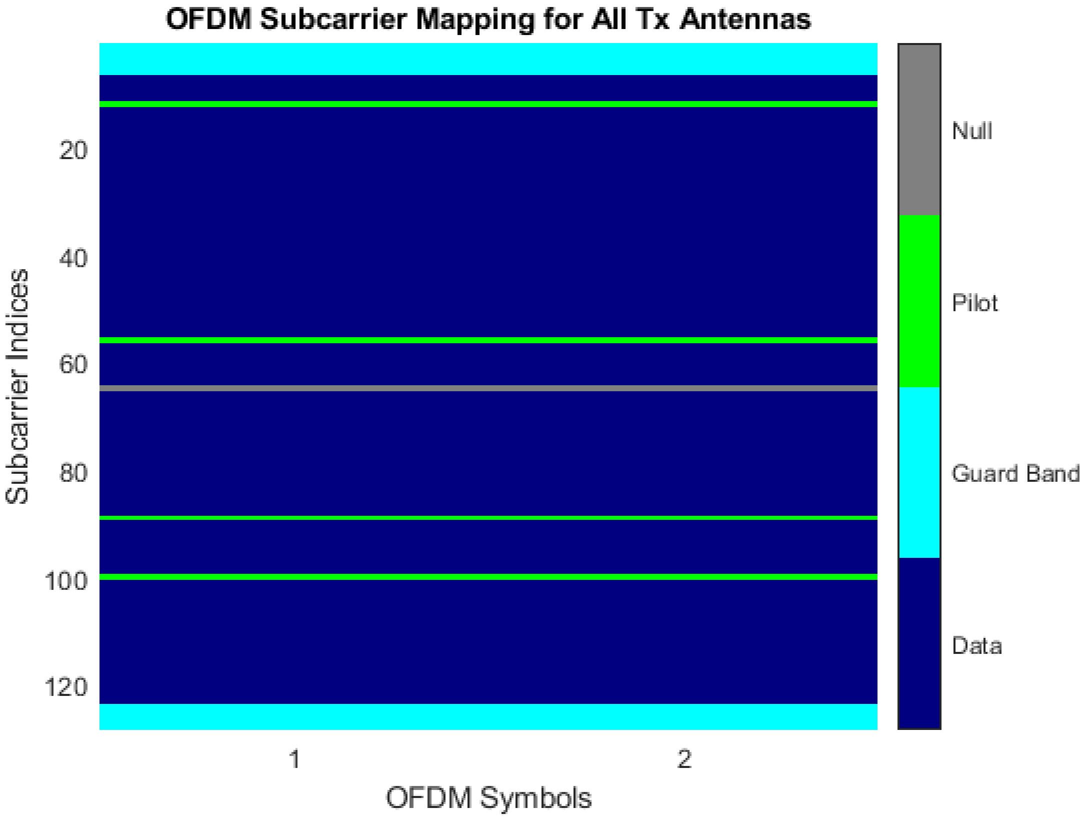
```
mod.PilotInputPort = true;
```

Specify the same pilot indices for both symbols.

```
mod.PilotCarrierIndices = [12; 56; 89; 100];
```

Visualize the placement of the pilot signals and nulls in the OFDM time-frequency grid by using the `showResourceMapping` object function.

```
showResourceMapping(mod)
```

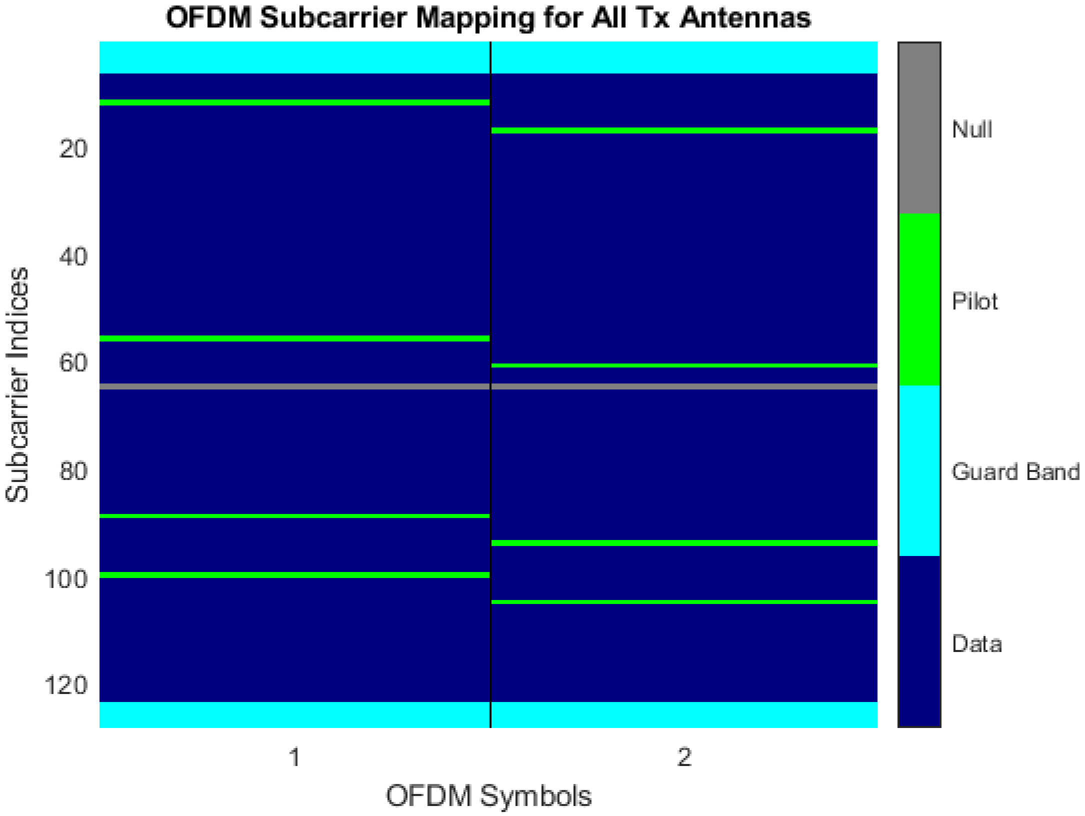


Specify different indices for the second symbol by concatenating a second column of pilot indices to the `PilotCarrierIndices` property.

```
mod.PilotCarrierIndices = cat(2,mod.PilotCarrierIndices, ...
    [17; 61; 94; 105]);
```

Verify that the pilot subcarrier indices differ between the two symbols.

```
showResourceMapping(mod)
```



Increase the number of transmit antennas to two.

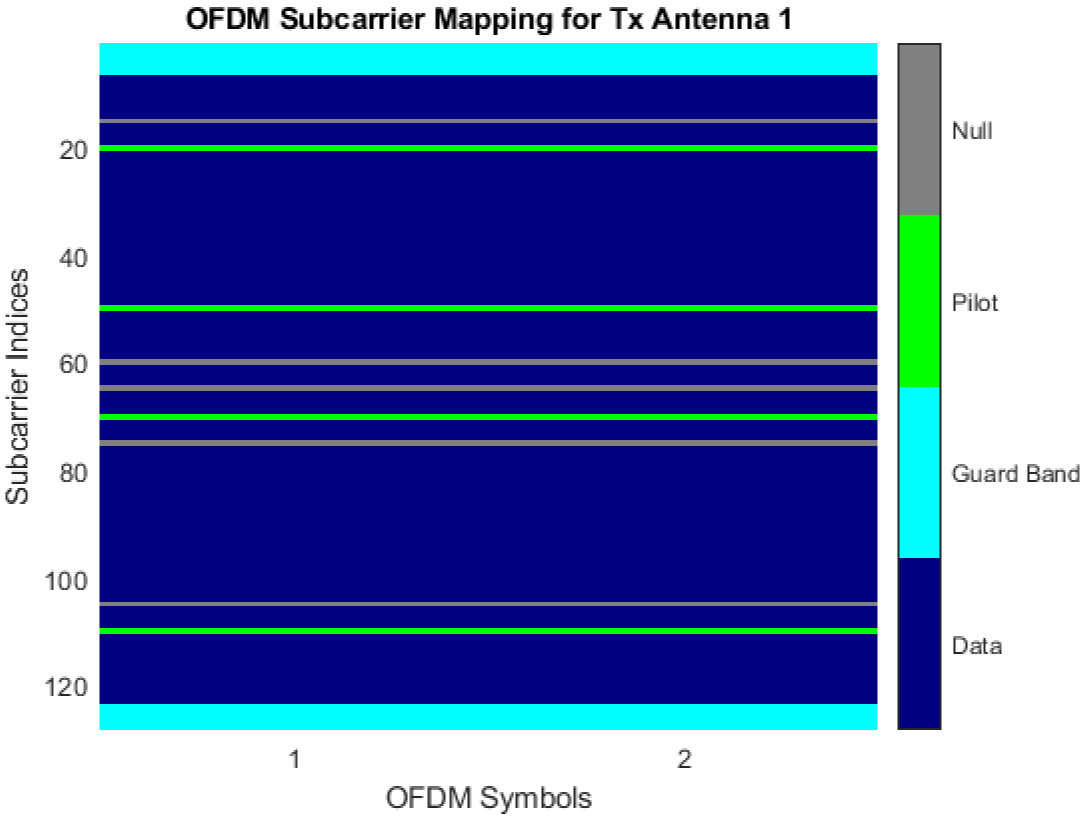
```
mod.NumTransmitAntennas = 2;
```

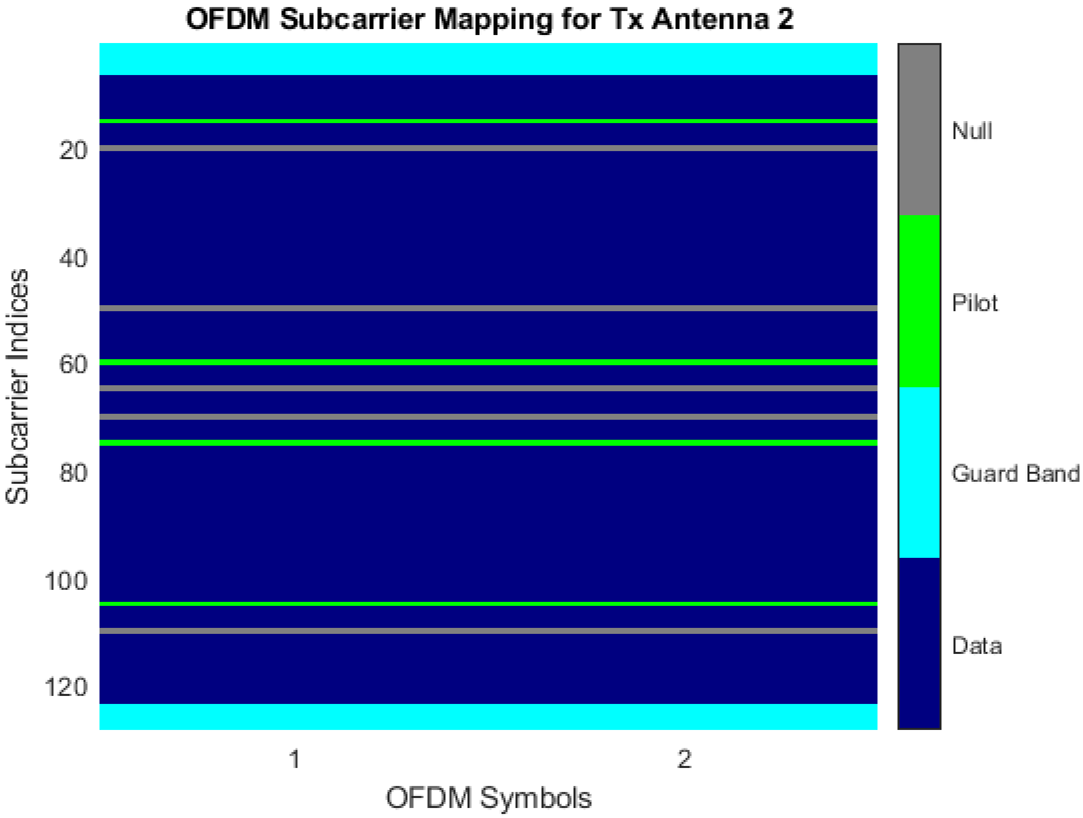
Specify the pilot indices for each of the two transmit antennas. To provide indices for multiple antennas while minimizing interference among the antennas, set the `PilotCarrierIndices` property as a 3-D array such that the indices for each symbol differ among antennas.

```
mod.PilotCarrierIndices = cat(3,[20; 50; 70; 110], [15; 60; 75; 105]);
```

Display the resource mapping for the two transmit antennas. The gray lines denote the insertion of custom nulls. The nulls are created by the object to minimize interference among the pilot symbols from different antennas.

```
showResourceMapping(mod)
```


OFDM Subcarrier Mapping for Tx Antenna 1
Subcarrier Indices
20
40
60
80
100
120
1
2
OFDM Symbols
Null
Pilot
Guard Band
Data



Create OFDM Modulator with Varying Cyclic Prefix Lengths

Specify the length of the cyclic prefix for each OFDM symbol.

Create an OFDM modulator, specifying five symbols, four left and three right guard-band subcarriers, and the cyclic prefix length for each OFDM symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
    'NumSymbols',5,...
    'CyclicPrefixLength',[12 10 14 11 13]);
```

Display the properties of the OFDM modulator, verifyING that the cyclic prefix length changes across symbols.

```
disp(mod)
```

```
  comm.OFDMModulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
           PilotInputPort: false
      CyclicPrefixLength: [12 10 14 11 13]
               Windowing: false
```

```
             NumSymbols: 5
    NumTransmitAntennas: 1
```

Determine OFDM Modulator Data Dimensions

Get the OFDM modulator data dimensions by using the `info` object function.

Construct an OFDM modulator System object™ with user-specified pilot indices, an inserted DC null, and specify two transmit antennas.

```
hMod = comm.OFDMModulator('NumGuardBandCarriers',[4;3], ...
    'PilotInputPort',true, ...
    'PilotCarrierIndices',cat(3,[12; 26; 40; 54], ...
    [11; 25; 39; 53]), ...
    'InsertDCNull',true, ...
    'NumTransmitAntennas',2);
```

Use the `info` object function to get the modulator input data, pilot input data, and output data sizes.

```
info(hMod)
```

```
ans = struct with fields:
     DataInputSize: [48 1 2]
    PilotInputSize: [4 1 2]
        OutputSize: [80 2]
```

Create OFDM Modulated Data

Generate OFDM modulated symbols for use in link-level simulations.

Construct an OFDM modulator with an inserted DC null, seven guard-band subcarriers, and two symbols having different pilot indices for each symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
'PilotInputPort',true, ...
'PilotCarrierIndices',[12 11; 26 27; 40 39; 54 55], ...
'NumSymbols',2, ...
'InsertDCNull',true);
```

Determine input data, pilot, and output data dimensions.

```
modDim = info(mod);
```

Generate random data symbols for the OFDM modulator. The structure variable, `modDim`, determines the number of data symbols.

```
dataIn = complex(randn(modDim.DataInputSize),randn(modDim.DataInputSize));
```

Create a pilot signal that has the correct dimensions.

```
pilotIn = complex(rand(modDim.PilotInputSize),rand(modDim.PilotInputSize));
```

Apply OFDM modulation to the data and pilot signals.

```
modData = step(mod,dataIn,pilotIn);
```

Use the OFDM modulator object to create the corresponding OFDM demodulator.

```
demod = comm.OFDMDemodulator(mod);
```

Demodulate the OFDM signal and output the data and pilot signals.

```
[dataOut, pilotOut] = step(demod,modData);
```

Verify that, within a tight tolerance, the input data and pilot symbols match the output data and pilot symbols.

```
isSame = (max(abs([dataIn(:) - dataOut(:); ...
    pilotIn(:) - pilotOut(:)])) < 1e-10)
```

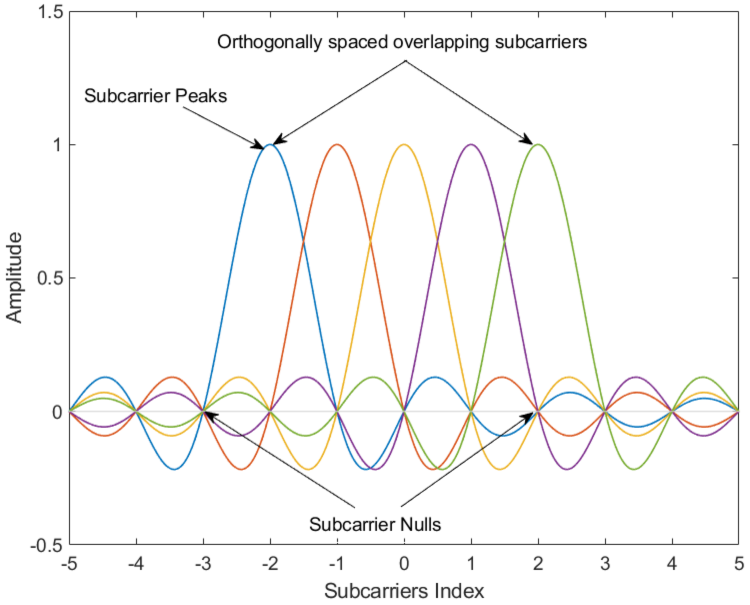
```
isSame = logical
   1
```

More About

Orthogonal Frequency Division Modulation

OFDM operation divides a high-rate data stream into lower data rate substreams by decomposing the transmission frequency band into *N* contiguous individually modulated subcarriers. Multiple parallel and orthogonal subcarriers carry the samples with almost the same bandwidth as a wideband channel. By using narrow orthogonal subcarriers, the OFDM signal gains robustness over a frequency-selective fading channel and eliminates adjacent subcarrier interference. Intersymbol interference (ISI) is reduced because the lower data rate substreams have symbol durations larger than the channel delay spread.

The Frequency domain representation of orthogonal subcarriers in an OFDM waveform looks as follows:



The transmitter applies inverse fast Fourier transform (IFFT) to N symbols at a time. The output of the IFFT is the sum of the N orthogonal sinusoids:

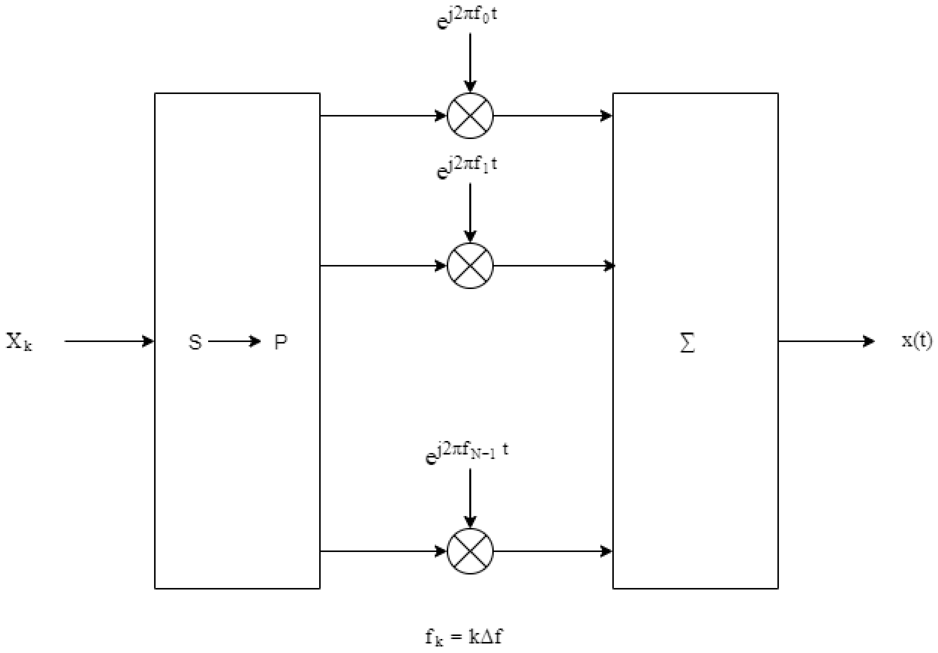
$$x(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi k\Delta f t}, \quad 0 \le t \le T,$$

where $\{X_\mathrm{k}\}$ are data symbols, and T is the OFDM symbol time. The data symbols X_k are typically complex and can be from any digital modulation alphabet (for example, QPSK, 16-QAM, 64-QAM).

The subcarrier spacing is $\Delta f = 1/T$; ensuring that the subcarriers are orthogonal over each symbol period, as shown below:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi m\Delta f t}\right)^* \left(e^{j2\pi n\Delta f t}\right) dt = \frac{1}{T}\int_0^T e^{j2\pi(m-n)\Delta f t}\, dt = 0 \quad \text{for } m \ne n\,.$$

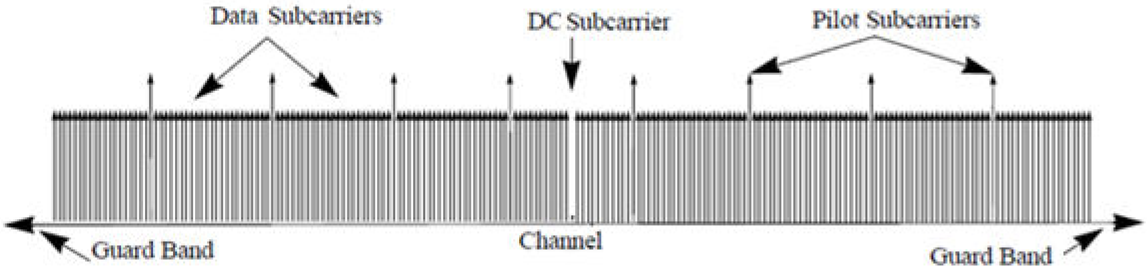
An OFDM modulator consists of a serial-to-parallel conversion followed by a bank of N complex modulators, individually corresponding to each OFDM subcarrier.



Subcarrier Allocation, Guard Bands and Guard Intervals

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, or guard band subcarriers.

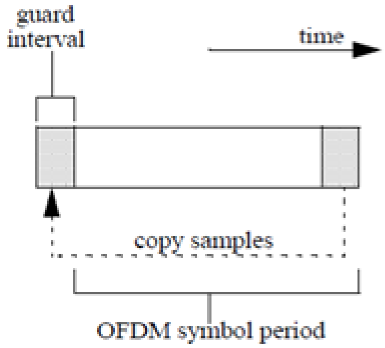


- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.
  - The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
  - The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

Similar to guard bands, guard intervals are used in OFDM to protect the integrity of transmitted signals by reducing intersymbol interference.

Assignment of guard intervals is analogous to the assignment of guard bands. You can model guard intervals to provide temporal separation between OFDM symbols. The guard intervals help preserve intersymbol orthogonality after the signal passes through time-dispersive channels. Guard intervals are created by using cyclic prefixes. Cyclic prefix insertion copies the last part of an OFDM symbol as the first part of the OFDM symbol.



As long as the span of the time dispersion does not exceed the duration of the cyclic prefix, the benefit of cyclic prefix insertion is maintained.

Inserting a cyclic prefix results in a fractional reduction of user data throughput because the cyclic prefix occupies bandwidth that could be used for data transmission.

Raised Cosine Windowing

While the cyclic prefix creates a guard period in time domain to preserve orthogonality, an OFDM symbol rarely begins with the same amplitude and phase exhibited at the end of the prior OFDM symbol causing spectral regrowth and therefore, spreading of signal bandwidth due to intermodulation distortion. To limit this spectral regrowth, it is desired to create a smooth transition between the last sample of a symbol and the first sample of the next symbol. This can be done by using a cyclic suffix and raised cosine windowing.

To create the cyclic suffix, the first N_{WIN} samples of a given symbol are appended to the end of that symbol. However, in order to comply with the 802.11g standard, for example, the length of a symbol cannot be arbitrarily lengthened. Instead, the cyclic suffix must overlap in time and is effectively summed with the cyclic prefix of the following symbol. This overlapped segment is where windowing is applied. Two windows are applied, one of which is the mathematical inverse of the other. The first raised cosine window is applied to the cyclic suffix of symbol k and decreases from 1 to 0 over its duration. The second raised cosine window is applied to the cyclic prefix of symbol k+1 and increases from 0 to 1 over its duration. This process provides a smooth transition from one symbol to the next.

The raised cosine window, $w(t)$, in the time domain can be expressed as:

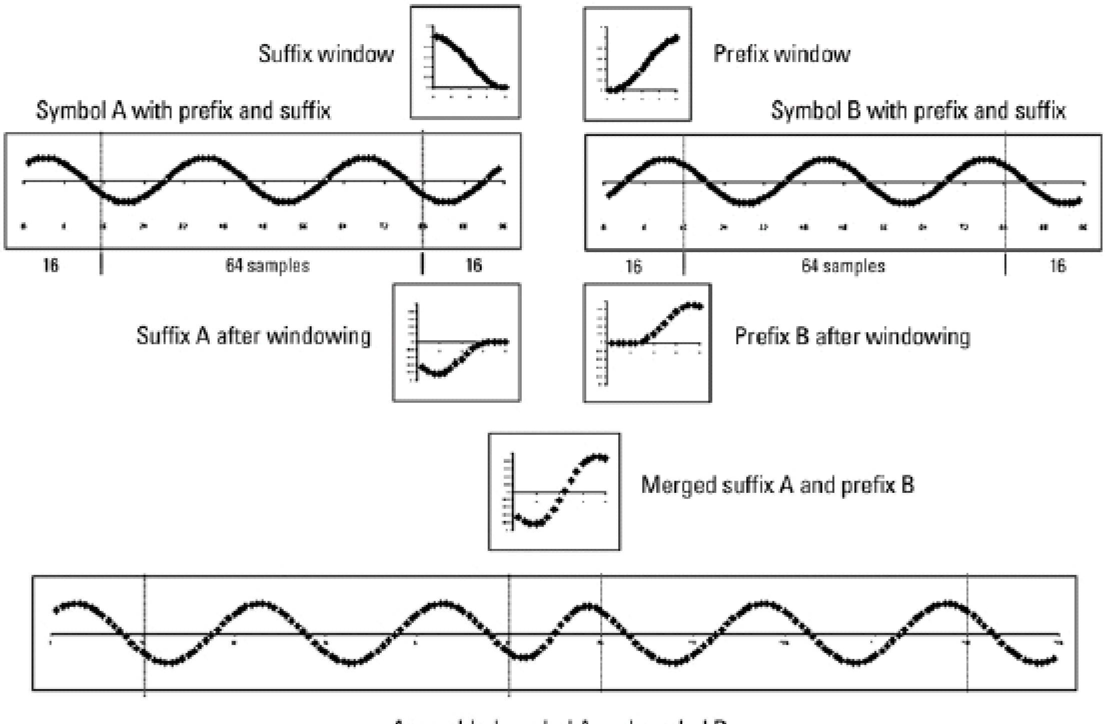
$$w(t) = \begin{cases} 1,\ 0 \le |t| < \dfrac{T - T_W}{2} \\ \dfrac{1}{2}\left\{1 + \cos\left[\dfrac{\pi}{T_W}\left(|t| - \dfrac{T - T_W}{2}\right)\right]\right\}, \dfrac{T - T_W}{2} \le |t| \le \dfrac{T + T_W}{2} \\ 0, \text{otherwise} \end{cases}$$

where:

- T is the OFDM symbol duration including the guard interval.
- T_W is the duration of the window.

Adjust the length of the cyclic suffix via the window length setting property, with suffix lengths set between 1 and the minimum cyclic prefix length. While windowing improves spectral regrowth, it does so at the expense of multipath fading immunity. This occurs because redundancy in the guard band is reduced because the guard band sample values are compromised by the smoothing.

The following figures display the application of raised cosine windowing.



References

[1] Dahlman, Erik, Stefan Parkvall, and Johan Sköld. 4G LTE/LTE-Advanced for Mobile Broadband. Amsterdam: Elsevier, Acad. Press, 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX*. Upper Saddle River, NJ: Prentice Hall, 2007.

[3] Agilent Technologies, Inc., “OFDM Raised Cosine Windowing”, http://wireless.agilent.com/rfcomms/n4010a/n4010aWLAN/onlineguide/ofdm_raised_cosine_windowing.htm.

[4] Montreuil, L., R. Prodan, and T. Kolze. “OFDM TX Symbol Shaping 802.3bn”, http://www.ieee802.org/3/bn/public/jan13/montreuil_01a_0113.pdf. Broadcom, 2013.

[5] “IEEE Standard 802.16TM-2009,” New York: IEEE, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`ofdmmod` | `qammod`

Objects

`comm.OFDMDemodulator` | `comm.QPSKModulator`

Blocks

OFDM Modulator Baseband

Introduced in R2014a

comm.OFDMDemodulator

Package: `comm`

Demodulate using OFDM method

Description

The `OFDMDemodulator` object demodulates using the orthogonal frequency division demodulation method. The output is a baseband representation of the modulated signal, which was input into the `OFDMModulator` companion object.

To demodulate an OFDM signal:

1. Define and set up the OFDM demodulator object. See "Construction" on page 3-253.
2. Call `step` to demodulate a signal according to the properties of `comm.OFDMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.OFDMDemodulator` creates a demodulator System object, `H`, that demodulates an input signal by using the orthogonal frequency division demodulation method.

`H = comm.OFDMDemodulator(Name,Value)` creates an OFDM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.OFDMDemodulator(hMod)` creates an OFDM demodulator object, `H`, whose properties are determined by the corresponding OFDM modulator object, `hMod`.

Properties

FFTLength

The length of the FFT, N_{FFT}, is equivalent to the number of subcarriers used in the modulation process. `FFTLength` must be ≥ 8.

Specify the number of subcarriers. The default is `64`.

NumGuardBandCarriers

The number of guard band subcarriers allocated to the left and right guard bands.

Specify the number of left and right subcarriers as nonnegative integers in $[0, N_{FFT}/2 - 1]$ where you specify the left, N_{leftG}, and right, N_{rightG}, guard bands independently in a 2-by-1 column vector. The default values are `[6; 5]`.

RemoveDCCarrier

A `logical` variable that when `true`, mandates removal of a DC subcarrier. The default value is `false`.

PilotOutputPort

A `logical` property that controls whether to separate the pilot signals and make them available at an additional output port. The location of each pilot output symbol is determined by the pilot subcarrier indices specified in the `PilotCarrierIndices` property. When false, pilot symbols may be present but embedded in the data. The default value is `false`.

PilotCarrierIndices

If the `PilotOutputPort` property is `true`, output separate pilot signals located at the indices specified by the `PilotCarrierIndices` property. If the indices are a 2-D array, the pilot carriers across all the transmit antennas per symbol are the same. If there is more than one transmit antenna (this information is not known by the demodulator), the pilots from different transmit antennas may interfere with each other. To avoid this, specify the pilot carrier indices as a 3-D array with different pilot indices for each symbol across the antennas. This avoids interference between pilots from different transmit antennas, since, on a per-symbol basis, each transmit antenna has different pilot carriers and the OFDM modulator creates custom nulls at the appropriate locations. The size of the third dimension of the `PilotCarrierIndices` property gives the number of transmit antennas.

CyclicPrefixLength

The cyclic prefix length property specifies the length of the OFDM cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same length through all antennas. The default value is `16`.

NumSymbols

This property specifies the number of symbols, N_{sym}. Specify N_{sym} as a positive integer. The default value is `1`.

NumReceiveAntennnas

This property determines the number of antennas, N_{R}, used to receive the OFDM modulated signal. Specify N_{R} as a positive integer. The default value is `1`.

Methods

info	Provide dimensioning information for the OFDM method
reset	Reset states of the `OFDMDemodulator` System object
showResourceMapping	Show the subcarrier mapping of the OFDM symbols created by the OFDM demodulator System object
step	Demodulate using OFDM method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Create and Modify OFDM Demodulator

Construct an OFDM demodulator System object™ with default properties. Modify some of the properties.

Construct the OFDM demodulator.

```
demod = comm.OFDMDemodulator
```

```
demod = 
  comm.OFDMDemodulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: false
      CyclicPrefixLength: 16
              NumSymbols: 1
      NumReceiveAntennas: 1
```

Modify the number of subcarriers and symbols.

```
demod.FFTLength = 128;
demod.NumSymbols = 2;
```

Verify that the number of subcarriers and the number of symbols changed.

```
demod
```

```
demod = 
  comm.OFDMDemodulator with properties:

               FFTLength: 128
    NumGuardBandCarriers: [2x1 double]
         RemoveDCCarrier: false
         PilotOutputPort: false
      CyclicPrefixLength: 16
              NumSymbols: 2
      NumReceiveAntennas: 1
```

Create OFDM Demodulator from OFDM Modulator

Create an OFDM demodulator System object™ from an existing OFDM modulator System object.

Construct an OFDM modulator using default parameters.

```
mod = comm.OFDMModulator('NumTransmitAntennas',4);
```

Construct the corresponding OFDM demodulator from the modulator, `mod`.

```
demod = comm.OFDMDemodulator(mod);
```

Display the properties of the modulator and verify that they match those of the demodulator.

```
mod
```

```
mod = 
  comm.OFDMModulator with properties:

                FFTLength: 64
     NumGuardBandCarriers: [2x1 double]
             InsertDCNull: false
           PilotInputPort: false
       CyclicPrefixLength: 16
                Windowing: false
               NumSymbols: 1
      NumTransmitAntennas: 4
```

```
demod
```

```
demod = 
  comm.OFDMDemodulator with properties:

                FFTLength: 64
     NumGuardBandCarriers: [2x1 double]
          RemoveDCCarrier: false
          PilotOutputPort: false
       CyclicPrefixLength: 16
               NumSymbols: 1
       NumReceiveAntennas: 1
```

Note that the number of transmit antennas is independent of the number of receive antennas.

Visualize Time-Frequency Resource Assignments

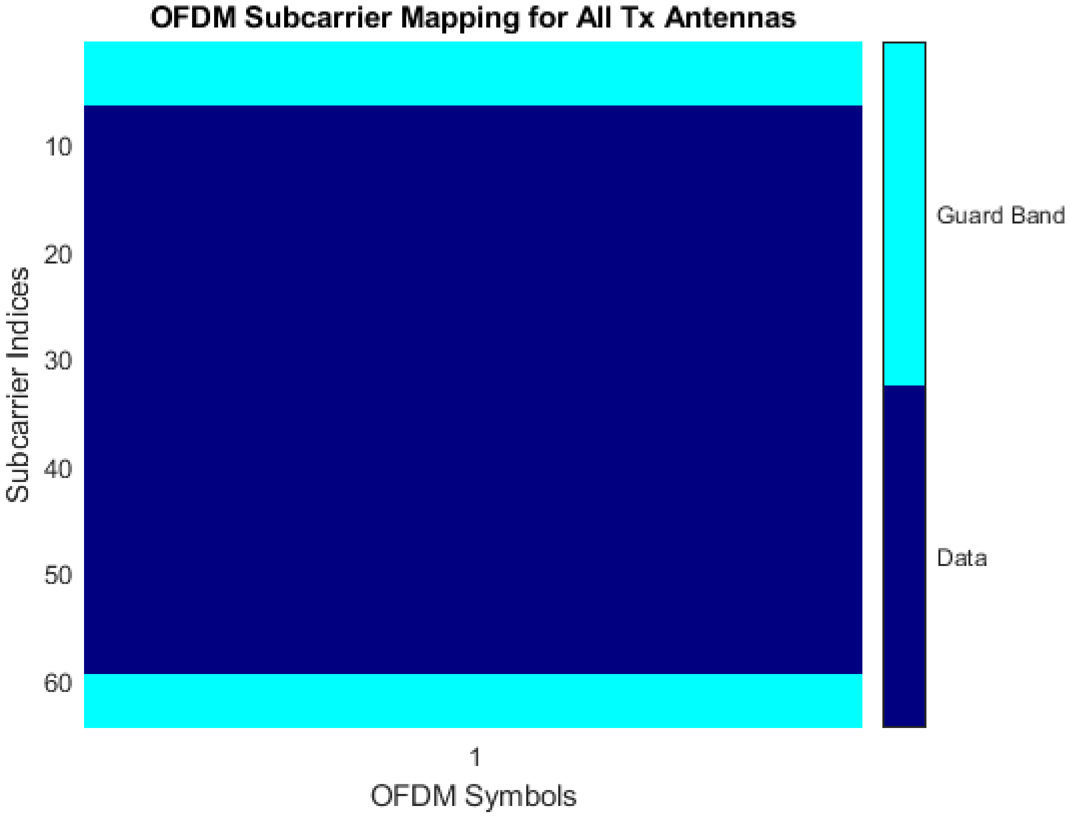
The `showResourceMapping` method shows the time-frequency resource mapping for each transmit antenna.

Construct an OFDM demodulator.

```
demod = comm.OFDMDemodulator;
```

Apply the `showResourceMapping` method.

```
showResourceMapping(demod)
```

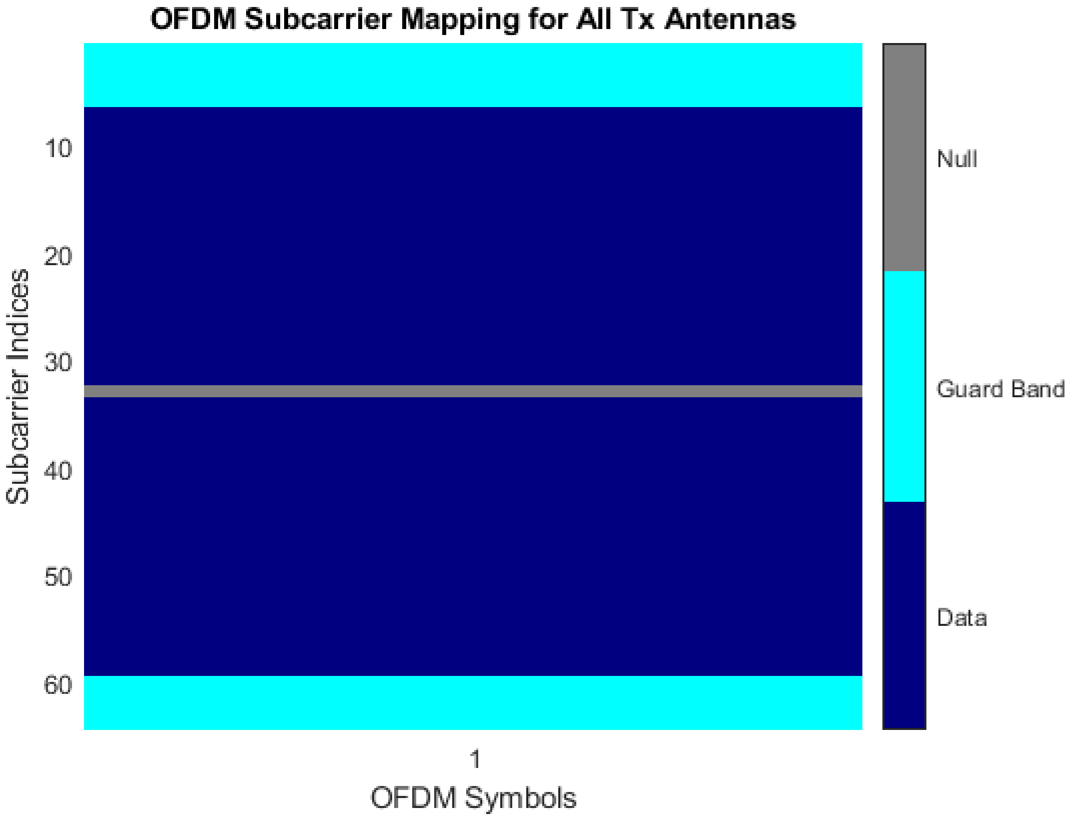


Remove the DC subcarrier.

```
demod.RemoveDCCarrier = true;
```

Show the resource mapping after removing the DC subcarrier.

```
showResourceMapping(demod)
```



Demodulate OFDM Data

Construct an OFDM modulator with an inserted DC null, seven guard-band subcarriers, and two symbols that have different pilot indices for each symbol.

```
mod = comm.OFDMModulator('NumGuardBandCarriers',[4;3],...
'PilotInputPort',true,'PilotCarrierIndices',cat(2,[12; 26; 40; 54],...
[11; 27; 39; 55]),'NumSymbols',2,'InsertDCNull',true);
```

Determine input data, pilot, and output data dimensions.

```
modDim = info(mod)
```

```
modDim = struct with fields:
     DataInputSize: [52 2]
    PilotInputSize: [4 2]
        OutputSize: [160 1]
```

Generate random data symbols for the OFDM modulator. Determine the number of data symbols by using the structure variable, `modDim`.

```
dataIn = complex(randn(modDim.DataInputSize),randn(modDim.DataInputSize));
```

Create a pilot signal that has the correct dimensions.

```
pilotIn = complex(rand(modDim.PilotInputSize),rand(modDim.PilotInputSize));
```

Apply OFDM modulation to the data and pilot signals.

```
modSig = step(mod,dataIn,pilotIn);
```

Use the OFDM modulator object to create the corresponding OFDM demodulator.

```
demod = comm.OFDMDemodulator(mod);
```

Demodulate the OFDM signal and output the data and pilot signals.

```
[dataOut,pilotOut] = step(demod,modSig);
```

Verify that the input data and pilot symbols match the output data and pilot symbols.

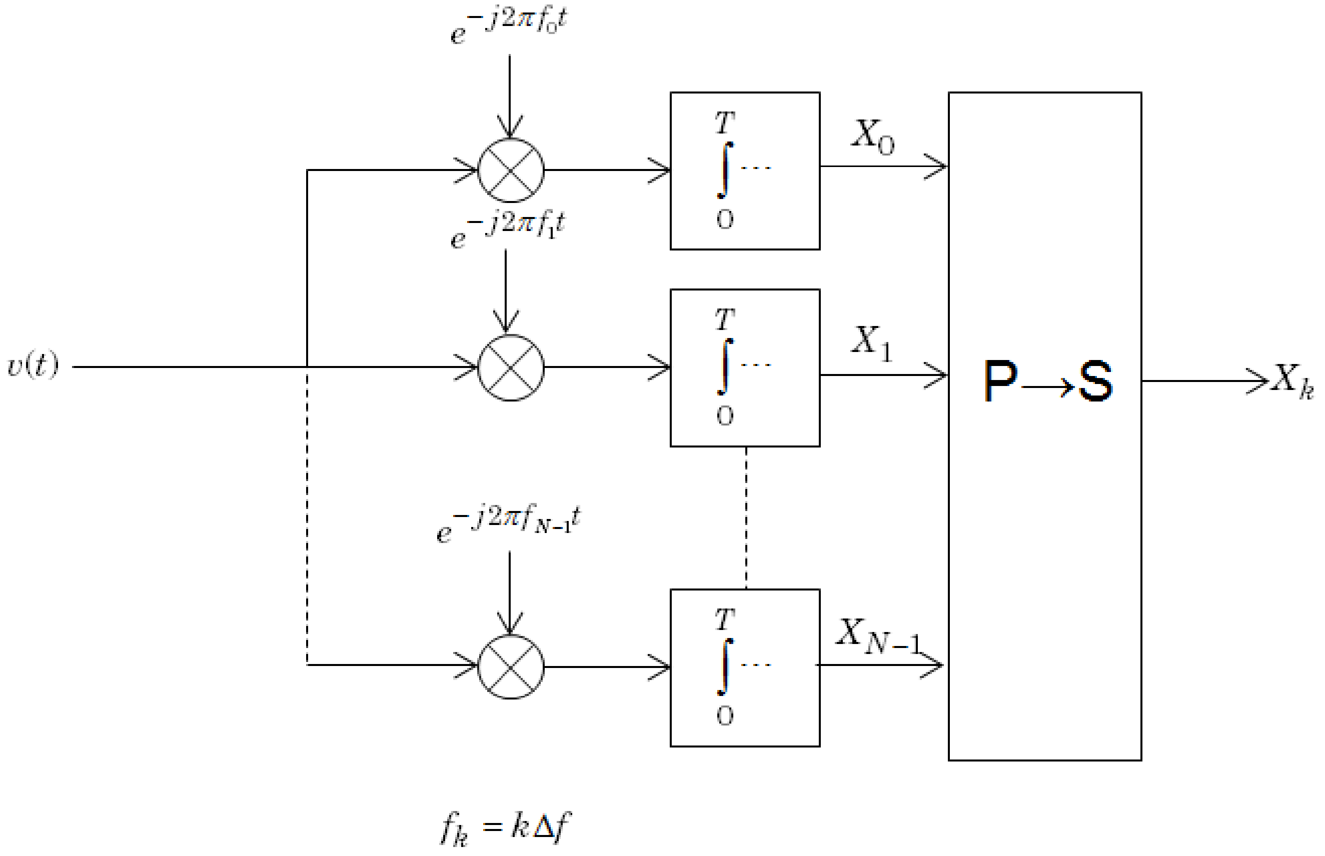
```
isSame = (max(abs([dataIn(:) - dataOut(:); ...
    pilotIn(:) - pilotOut(:)])) < 1e-10)
```

```
isSame = logical
   1
```

Algorithms

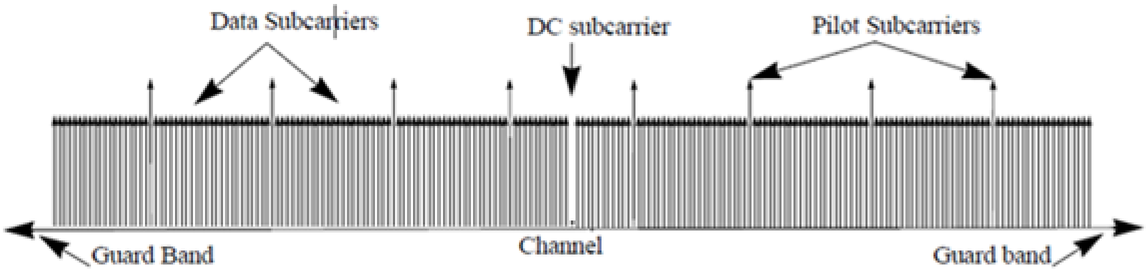
The Orthogonal Frequency Division Modulation (OFDM) Demodulator System object demodulates an OFDM input signal by using an FFT operation that results in *N* parallel data streams.

The figure shows an OFDM demodulator. It consists of a bank of *N* correlators with one assigned to each OFDM subcarrier followed by a parallel-to-serial conversion.

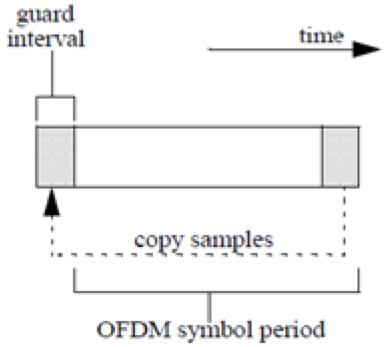


Guard Bands and Intervals

There are three types of OFDM subcarriers: data, pilot, and null. Data subcarriers are used for transmitting data while pilot subcarriers are used for channel estimation. There is no transmission on null subcarriers, which are used to provide a DC null as well as to provide buffers between OFDM resource blocks. These buffers are referred to as guard bands whose purpose is to prevent inter-symbol interference. The allocation of nulls and guard bands varies depending upon the standard, e.g., 802.11n differs from LTE. Consequently, the OFDM modulator object allows the user to assign subcarrier indices as required.



Analogous to the concept of guard bands, the OFDM modulator object supports guard intervals that provide temporal separation between OFDM symbols so that the signal does not lose orthogonality due to time-dispersive channels. As long as the guard interval is longer than the delay spread, each symbol does not interfere with other symbols. Guard intervals are created by using cyclic prefixes in which the last part of an OFDM symbol is copied and inserted as the first part of the OFDM symbol. The benefit of cyclic prefix insertion is maintained as long as the span of the time dispersion does not exceed the duration of the cyclic prefix. The OFDM modulator object enables the cyclic prefix length to be set. The drawback in using a cyclic prefix is increased overhead.



Selected Bibliography

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed, *Fundamentals of WiMAX*, Upper Saddle River, NJ: Prentice Hall, 2007.

[3] I. E. E. E., “IEEE Standard 802.16TM-2009.”

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.OFDMModulator` | `comm.QPSKDemodulator` | `comm.RectangularQAMDemodulator`

Blocks

OFDM Demodulator Baseband

Functions

`ofdmdemod`

Introduced in R2014a

info

System object: `comm.OFDMDemodulator`
Package: `comm`

Provide dimensioning information for the OFDM method

Syntax

```
Y = info(H)
```

Description

`Y = info(H)` provides data dimensioning information for the OFDM demodulator System object, `H`. It returns the expected dimensions for data input into the OFDM demodulator, for the pilot output, and for the data output from the demodulator. The output, `Y`, is a structure containing three fields: `InputSize`, `DataOutputSize`, and `PilotOutputSize`.

Y.InputSize

Gives the dimensions of the demodulator input data, [(N_{FFT} + N_{CP}) × N_{sym}]-by-N_r, where N_{FFT} is the number of subcarriers, N_{CP} is the length of the cyclic prefix, N_{sym} is the number of symbols, and N_r is the number of receive antennas.

Y.DataOutputSize

Shows the dimensions of the demodulator output data, N_{data}-by-N_{sym}-by-N_r, where N_{data} is the number of data subcarriers such that $N_{data} = N_{FFT} - N_{leftG} - N_{rightG} - N_{DCNull} - N_{pilot} - N_{custNull}$. The variables are defined as follows:

N_{FFT}	Number of subcarriers
N_{leftG}	Number of subcarriers in the left guard band
N_{rightG}	Number of subcarriers in the right guard band
N_{DCNull}	Number of subcarriers in the DC null (either 0 or 1)
N_{pilot}	Number of pilot subcarriers
$N_{custNull}$	Number of subcarriers used for custom nulls

Y.PilotOutputSize

Provides the dimensions of the pilot signal output array, N_{pilot}-by-N_{sym}-by-N_r or N_{pilot}-by-N_{sym}-by-N_t-by-N_r, depending on the number of transmit antennas.

reset

System object: `comm.OFDMDemodulator`
Package: `comm`

Reset states of the `OFDMDemodulator` System object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `OFDMDemodulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

showResourceMapping

System object: `comm.OFDMDemodulator`
Package: `comm`

Show the subcarrier mapping of the OFDM symbols created by the OFDM demodulator System object

Syntax

```
showResourceMapping(H)
showResourceMapping(H,CI)
```

Description

`showResourceMapping(H)` shows a visualization of the subcarrier mapping for the OFDM symbols used by the OFDM demodulator System object, `H`. The subcarrier indices are numbered from 1 to N_{FFT}.

`showResourceMapping(H,CI)` shows the resource mapping where the optional argument, `CI`, is used to number the subcarrier indices that will be displayed. `CI` is a 1x2 integer row vector such that `diff(CI)`$= N_{FFT} - 1$.

step

System object: `comm.OFDMDemodulator`
Package: `comm`

Demodulate using OFDM method

Syntax

```
Y = step(H,X)
[Y,PILOT] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the OFDM demodulator System object, `H`, and returns the baseband demodulated output, `Y`. The input is a double-precision, real or complex, 2-D matrix of symbols whose dimensions are a function of the number of subcarriers, the cyclic prefix length, and the number of receive antennas. You can determine the dimensions by using the `info` method. The output, `Y`, is a double-precision, complex, 3-D array.

`[Y,PILOT] = step(H,X)` separates the `PILOT` signal on the subcarriers specified by the PilotCarrierIndices property value of `H`. This syntax applies when the PilotOutputPort property of `H` is true. `PILOT` is a double-precision, complex, 3-D array.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CarrierSynchronizer

Package: `comm`

Compensate for carrier frequency offset

Description

The `comm.CarrierSynchronizer` System object compensates for carrier frequency and phase offsets in signals that use single-carrier modulation schemes. The carrier synchronizer algorithm is compatible with BPSK, QPSK, OQPSK, 8-PSK, PAM, and rectangular QAM modulation schemes.

Note

- This System object does not resolve phase ambiguities created by the synchronization algorithm. As indicated in this table, the potential phase ambiguity introduced by the synchronizer depends on the modulation type:

Modulation	Phase Ambiguity (degrees)
`'BPSK'` or `'PAM'`	0, 180
`'OQPSK'`, `'QPSK'`, or `'QAM'`	0, 90, 180, 270
`'8PSK'`	0, 45, 90, 135, 180, 225, 270, 315

  The “Examples” on page 3-0 demonstrate carrier synchronization and resolution of phase ambiguity.
- For best results, apply carrier synchronization to non-oversampled signals, as demonstrated in “Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization” on page 3-275.

To compensate for frequency and phase offsets in signals that use single-carrier modulation schemes:

1. Create the `comm.CarrierSynchronizer` object and set its properties.
2. Call the object, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
carrSynch = comm.CarrierSynchronizer
carrSynch = comm.CarrierSynchronizer(Name,Value)
```

Description

`carrSynch = comm.CarrierSynchronizer` creates a System object that compensates for carrier frequency offset and phase offset in signals that use single-carrier modulation schemes.

`carrSynch = comm.CarrierSynchronizer(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Modulation` — Modulation type

`'QAM'` (default) | `'8PSK'` | `'BPSK'` | `'OQPSK'` | `'PAM'` | `'QPSK'`

Modulation type, specified as `'QAM'`, `'8PSK'`, `'BPSK'`, `'OQPSK'`, `'PAM'`, or `'QPSK'`.

Example: `comm.CarrierSynchronizer('Modulation','QPSK')` creates a carrier synchronizer System object to use with a QPSK modulated signal.

Tunable: No

`ModulationPhaseOffset` — Modulation phase offset method

`'Auto'` (default) | `'Custom'`

Modulation phase offset method, specified as `'Auto'` or `'Custom'`.

- `'Auto'` — Apply the traditional offset for the specified modulation type.

Modulation	Phase Offset (radians)
`'BPSK'`, `'QAM'`, or `'PAM'`	0
`'OQPSK'` or `'QPSK'`	$\pi/4$
`'8PSK'`	$\pi/8$

- `'Custom'` — Specify a user-defined phase offset with the CustomPhaseOffset property.

Tunable: Yes

`CustomPhaseOffset` — Custom phase offset

`0` (default) | scalar

Custom phase offset in radians, specified as a scalar.

Dependencies

This property applies when the ModulationPhaseOffset property is set to `'Custom'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

2 (default) | positive integer

Number of samples per symbol, specified as a positive integer.

Tunable: Yes

Data Types: `double`

`DampingFactor` — Damping factor of loop

`0.707` (default) | positive scalar

Damping factor of the loop, specified as a positive scalar.

Tunable: Yes

Data Types: `double`

`NormalizedLoopBandwidth` — Normalized bandwidth of loop

`0.01` (default) | scalar

Normalized bandwidth of the loop, specified as a scalar in the range (0,1]. The loop bandwidth is normalized by the sample rate of the synchronizer.

Decreasing the loop bandwidth reduces the synchronizer convergence time but also reduces the pull-in range of the synchronizer.

Tunable: Yes

Data Types: `double`

Usage

Syntax

```
[outSig,phErr] = carrSynch(inSig)
```

Description

`[outSig,phErr] = carrSynch(inSig)` compensates for frequency offset and phase offset in the input signal. This System object returns a compensated output signal and an estimate of the phase error.

Input Arguments

`inSig` — Input signal

scalar | column vector

Input signal, specified as a complex scalar or a column vector of complex values.

Data Types: `double` | `single`
Complex Number Support: Yes

Output Arguments

`outSig` — Output signal

scalar (default) | column vector

Output signal, returned as a scalar or column vector with the same data type and length as `inSig`. The output signal adjusts the input signal compensating for carrier frequency and phase offsets in signals that use single-carrier modulation schemes.

`phErr` — Phase error estimate

scalar (default) | column vector

Phase error estimate in radians, returned as a scalar or column vector with the same length as `inSig`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.CarrierSynchronizer`

info	Characteristic information about carrier synchronizer
clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Correct Phase and Frequency Offset in QPSK Link

Correct phase and frequency offsets of a QPSK signal passed through an AWGN channel. Using preambles, resolve phase ambiguity.

Define the simulation parameters.

```
M = 4; % Modulation order
rng(1993) % For repeatable results
barker = comm.BarkerCode(...
    'Length',13,'SamplesPerFrame',13);  % For preamble
msgLen = 1e4;
numFrames = 10;
frameLen = msgLen/numFrames;
```

Add preambles to each frame, which are used later when performing phase ambiguity resolution. Generate random data symbols, and apply QPSK modulation.

```
preamble = (1+barker())/2;  % Length 13, unipolar
data = zeros(msgLen,1);
for idx = 1 : numFrames
    payload = randi([0 M-1],frameLen-barker.Length,1);
    data((idx-1)*frameLen + (1:frameLen)) = [preamble; payload];
end

modSig = pskmod(data,4,pi/4);
```

Create a `comm.PhaseFrequencyOffset` System object™ to introduce phase and frequency offsets to the modulated input signal. Set the phase offset to 45 degrees, frequency offset to 1 kHz, and sample rate to 10 kHz. The frequency offset is set to 1% of the sample rate.

```
pfo = comm.PhaseFrequencyOffset('PhaseOffset',45, ...
    'FrequencyOffset',1e4,'SampleRate',1e6);
```

Create a carrier synchronizer System object to use for correcting the phase and frequency offsets with samples per symbol set to 1.

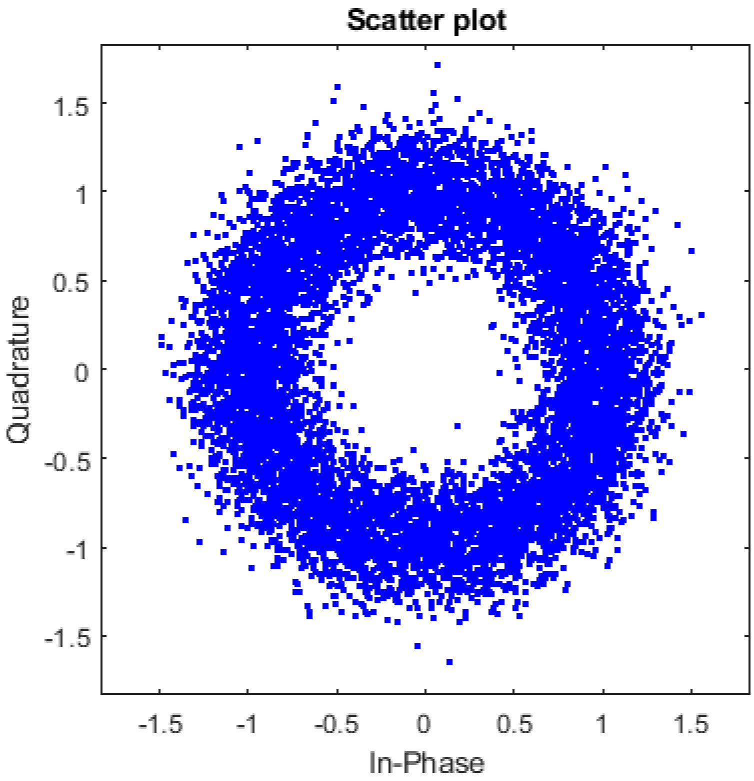
```
carrierSync = comm.CarrierSynchronizer( ...
    'SamplesPerSymbol',1,'Modulation','QPSK');
```

Apply phase and frequency offsets using the `pfo` System object, and then pass the signal through an AWGN channel to add white Gaussian noise.

```
modSigOffset = pfo(modSig);
rxSig = awgn(modSigOffset,12);
```

Display the scatter plot of the received signal. The data appear in a circle instead of being grouped around the reference constellation points due to the frequency offset.

```
scatterplot(rxSig)
```

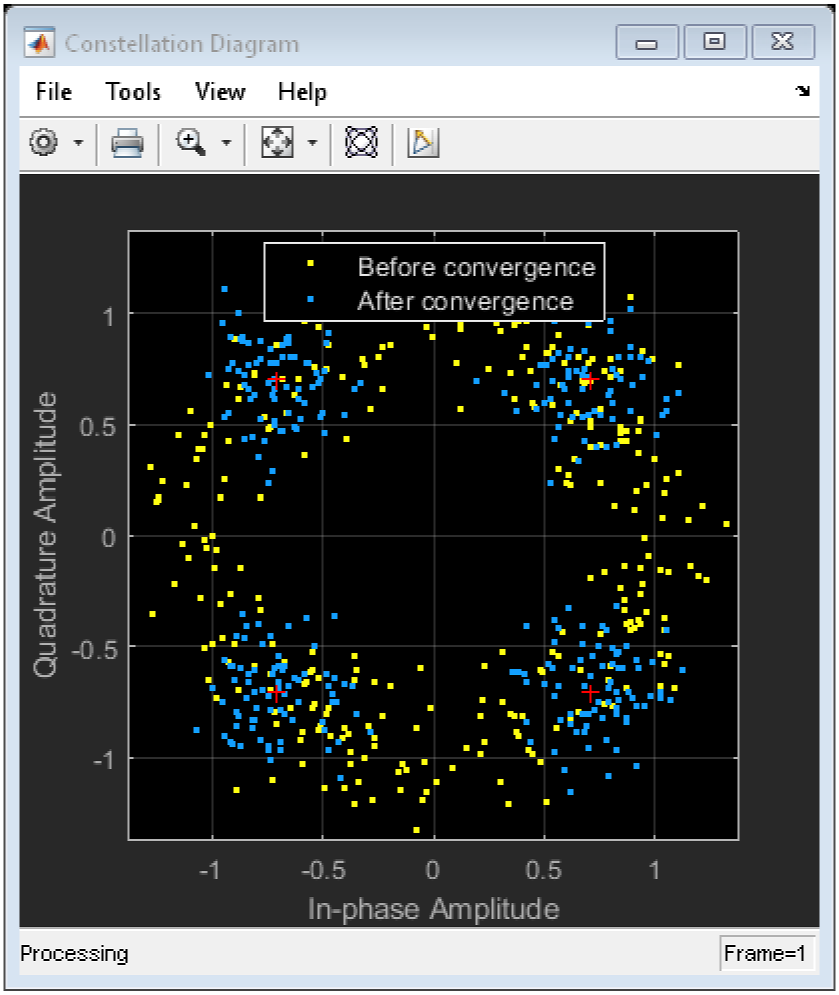


Use the `carrierSync` System object to correct the phase and frequency offset in the received signal.

```
syncSignal = carrierSync(rxSig);
```

Use a constellation diagram to display the first and last 1000 symbols of the synchronized signal. Before convergence of the synchronizer loop, the plotted symbols are not grouped around the reference constellation points. After convergence, the plotted symbols are grouped around the reference constellation points.

```
constDiag = comm.ConstellationDiagram( ...
'SymbolsToDisplaySource','Property','SymbolsToDisplay',300, ...
'ChannelNames',{'Before convergence','After convergence'},'ShowLegend',true, ...
'Position',[400 400 400 400]);

constDiag([syncSignal(1:1000) syncSignal(9001:10000)]);
```



Demodulate the synchronized signal. Compute and display the total bit errors and BER.

```
syncData = pskdemod(syncSignal,4,pi/4);
[syncDataTtlErr,syncDataBER] = biterr(data(6000:end),syncData(6000:end))
```

```
syncDataTtlErr = 3990

syncDataBER = 0.4986
```

Phase ambiguity in the received signal might cause bit errors. Using the preamble, determine phase ambiguity. Remove this phase ambiguity from the synchronized signal to reduce bit errors.

```
idx = 9000 + (1:barker.Length);
phOffset = angle(modSig(idx) .* conj(syncSignal(idx)));
phOffset = round((2/pi) * phOffset); % -1, 0, 1, +/-2
phOffset(phOffset==-2) = 2; % Prep for mean operation
phOffset = mean((pi/2) * phOffset); % -pi/2, 0, pi/2, or pi
disp(['Estimated mean phase offset = ',num2str(phOffset*180/pi),' degrees'])
```

```
Estimated mean phase offset = 180 degrees
```

```
resPhzSig = exp(1i*phOffset) * syncSignal;
```

Demodulate the signal after resolving the phase ambiguity. Recompute and display the updated total bit errors and BER. Removing the phase ambiguity reduces the BER dramatically.

```
resPhzData = pskdemod(resPhzSig,4,pi/4);
[resPhzTtlErr, resPhzBER] = biterr(data(6000:end),resPhzData(6000:end))
```

```
resPhzTtlErr = 403

resPhzBER = 0.0504
```

Estimate Frequency Offset in an 8-PSK Link

Estimate the frequency offset introduced into a noisy 8-PSK signal using a carrier synchronizer System object™.

Define the simulation parameters.

```
M = 8;                  % Modulation order
fs = 1e6;               % Sample rate (Hz)
foffset = 1000;         % Frequency offset (Hz)
phaseoffset = 15;       % Phase offset (deg)
snrdb = 20;             % Signal-to-noise ratio (dB)
```

Create a `comm.PhaseFrequencyOffset` System object to introduce phase and frequency offsets to a modulated signal.

```
pfo = comm.PhaseFrequencyOffset('PhaseOffset',phaseoffset, ...
    'FrequencyOffset',foffset,'SampleRate',fs);
```

Create a carrier synchronizer System object to use for correcting the phase and frequency offsets. Set the `Modulation` property to `8PSK`.

```
carrierSync = comm.CarrierSynchronizer('Modulation','8PSK');
```

Generate random data and apply 8-PSK modulation.

```
data = randi([0 M-1],5000,1);
modSig = pskmod(data,M,pi/M);
```

Apply phase and frequency offsets using the pfo System object, and pass the signal through an AWGN channel to add Gaussian white noise.

```
modSigOffset = pfo(modSig);
rxSig = awgn(modSigOffset,snrdb);
```

Use the carrier synchronizer to estimate the phase offset of the received signal.

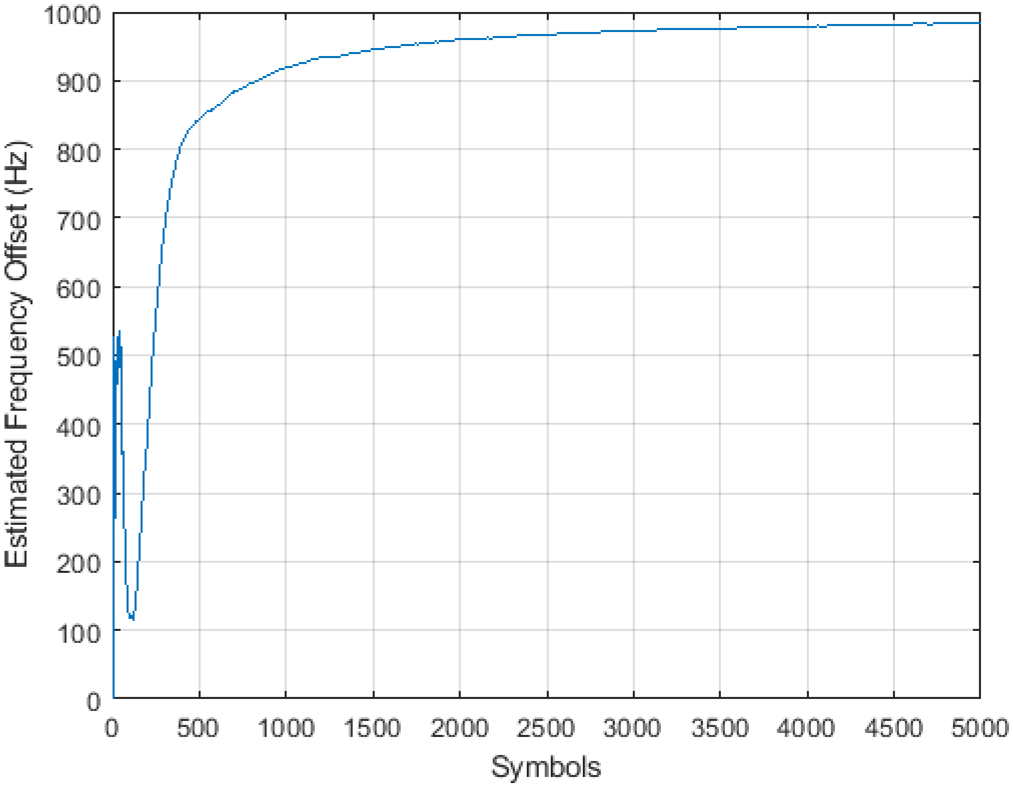
```
[~,phError] = carrierSync(rxSig);
```

Determine the frequency offset by using the `diff` function to compute an approximate derivative of the phase error. The derivative must be scaled by 2π because the phase error is measured in radians.

```
estFreqOffset = diff(phError)*fs/(2*pi);
```

Plot the running mean of the estimated frequency offset. After the synchronizer converges to a solution, the mean value of the estimate is approximately equal to the input frequency offset value of 1000 Hz.

```
rmean = cumsum(estFreqOffset)./(1:length(estFreqOffset))';
plot(rmean)
xlabel('Symbols')
ylabel('Estimated Frequency Offset (Hz)')
grid
```



Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization

Compensation of significant phase and frequency offsets for a 16-QAM signal in an AWGN channel is accomplished in two steps. First, correct the coarse frequency offset using the estimate provided by the coarse frequency compensator, and then fine-tune the correction using carrier synchronization. Because of the coarse frequency correction, the carrier synchronizer converges quickly even though the normalized bandwidth is set to a low value. Lower normalized bandwidth values enable better correction for small residual carrier offsets. After applying phase and frequency offset corrections to the received signal, resolve phase ambiguity using the preambles.

Define the simulation parameters.

```
fs = 10000;       % Sample rate (Hz)
sps = 4;          % Samples per symbol
```

```
M = 16;          % Modulation order
k = log2(M);     % Bits per symbol
rng(1996)        % Set seed for repeatable results
barker = comm.BarkerCode(...
    'Length',13,'SamplesPerFrame',13);  % For preamble
msgLen = 1e4;
numFrames = 10;
frameLen = msgLen/numFrames;
```

Generate data payloads and add the preamble to each frame. The preamble is later used for phase ambiguity resolution.

```
preamble = (1+barker())/2;  % Length 13, unipolar
data = zeros(msgLen, 1);
for idx = 1 : numFrames
    payload = randi([0 M-1],frameLen-barker.Length,1);
    data((idx-1)*frameLen + (1:frameLen)) = [preamble; payload];
end
```

Create a System object for the transmit pulse shape filtering, the receive pulse shape filtering, the QAM coarse frequency compensation, the carrier synchronization, and a constellation diagram.

```
txFilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',sps);
rxFilter = comm.RaisedCosineReceiveFilter(...
    'InputSamplesPerSymbol',sps,'DecimationFactor',sps);
coarse = comm.CoarseFrequencyCompensator('SampleRate',fs, ...
    'FrequencyResolution',10);
fine = comm.CarrierSynchronizer( ...
    'DampingFactor',0.4,'NormalizedLoopBandwidth',0.001, ...
    'SamplesPerSymbol',1,'Modulation','QAM');
axislimits = [-6 6];
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',qammod(0:M-1,M), ...
    'ChannelNames',{'Before convergence','After convergence'}, ...
    'ShowLegend',true,'XLimits',axislimits,'YLimits',axislimits);
```

Also create a System object for the AWGN channel, and the phase and frequency offset to add impairments to the signal. A phase offset greater than 90 degrees is added to induce a phase ambiguity that results in a constellation quadrant shift.

```
ebn0 = 8;
freqoffset = 110;
phaseoffset = 110;
awgnChannel = comm.AWGNChannel('EbNo',ebn0, ...
    'BitsPerSymbol',k,'SamplesPerSymbol',sps);
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',freqoffset, ...
    'PhaseOffset',phaseoffset,'SampleRate',fs);
```

Generate random data symbols, apply 16-QAM modulation, and pass the modulated signal through the transmit pulse shaping filter.

```
txMod = qammod(data,M);
txSig = txFilter(txMod);
```

Apply phase and frequency offsets using the `pfo` System object, and then pass the signal through an AWGN channel to add white Gaussian noise.

```
txSigOffset = pfo(txSig);
rxSig = awgnChannel(txSigOffset);
```

The coarse frequency compensator System object provides a rough correction for the frequency offset. For the conditions in this example, correcting the frequency offset of the received signal correction to within 10 Hz of the transmitted signal is sufficient.

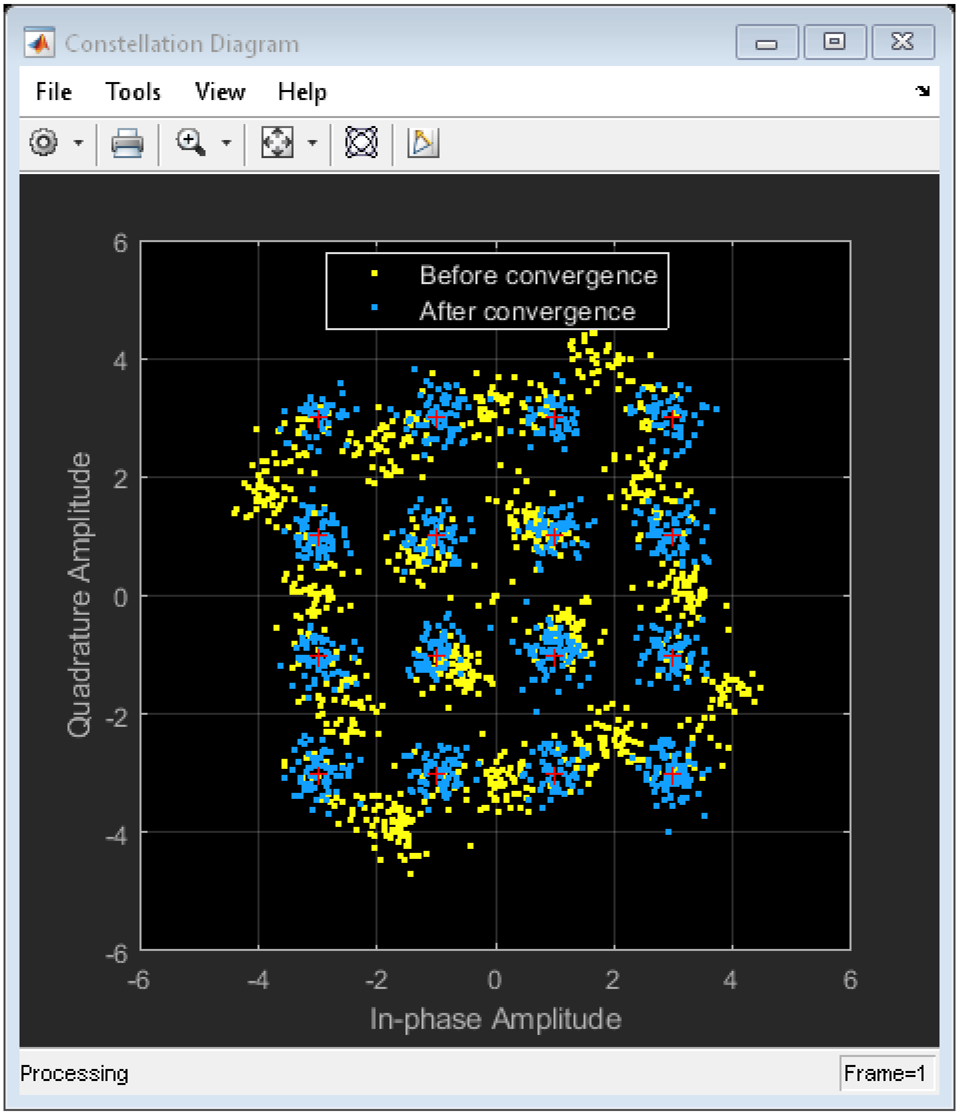
```
syncCoarse = coarse(rxSig);
```

Pass the signal through the receive pulse shaping filter, and apply fine frequency correction.

```
rxFiltSig = fine(rxFilter(syncCoarse));
```

Display the constellation diagram of the first and last 1000 symbols in the signal. Before convergence of the synchronization loop, the spiral nature of the diagram indicates that the frequency offset is not corrected. After the carrier synchronizer has converged to a solution, the symbols are aligned with the reference constellation.

```
constDiagram([rxFiltSig(1:1000) rxFiltSig(9001:end)])
```



Demodulate the signal. Account for the signal delay caused by the transmit and receive filters to align the received data with the transmitted data. Compute and display the total bit errors and BER. When checking the bit errors, use the later portion of the received signal to be sure the synchronization loop has converged.

```
rxData = qamdemod(rxFiltSig,M);
delay = (txFilter.FilterSpanInSymbols + rxFilter.FilterSpanInSymbols) / 2;
idxSync = 2000; % Check BER for the received signal after the synchronization loop has converged
[syncDataTtlErr,syncDataBER] = biterr(data(idxSync:end-delay),rxData(idxSync+delay:end))
```

```
syncDataTtlErr = 16116

syncDataBER = 0.5042
```

Depending on the random data used, there may be bit errors resulting from phase ambiguity in the received signal after the synchronization loop converges and locks. In this case, you can use the preamble to determine and then remove the phase ambiguity from the synchronized signal to reduce bit errors. If phase ambiguity is minimal, the number of bit errors may be unchanged.

```
idx = 9000 + (1:barker.Length);
phOffset = angle(txMod(idx) .* conj(rxFiltSig(idx+delay)));

phOffsetEst = mean(phOffset);
disp(['Phase offset = ',num2str(rad2deg(phOffsetEst)),' degrees'])
```

```
Phase offset = -90.1401 degrees
```

```
resPhzSig = exp(1i*phOffsetEst) * rxFiltSig;
```

Demodulate the signal after resolving the phase ambiguity. Recompute the total bit errors and BER.

```
resPhzData = qamdemod(resPhzSig,M);
[resPhzTtlErr,resPhzBER] = biterr(data(idxSync:end-delay),resPhzData(idxSync+delay:end))
```

```
resPhzTtlErr = 5

resPhzBER = 1.5643e-04
```

MSK Signal Recovery

Model channel impairments such as timing phase offset, carrier frequency offset, and carrier phase offset for a minimum shift keying (MSK) signal. Use `comm.MSKTimingSynchronizer` and `comm.CarrierSynchronizer` System objects to synchronize such signals at the receiver. The MSK timing synchronizer recovers the timing offset, while a carrier synchronizer recovers the carrier frequency and phase offsets.

Initialize system variables by running the MATLAB script `configureMSKSignalRecoveryEx`. Define the logical control variable `recoverTimingPhase` to enable timing phase recovery, and `recoverCarrier` to enable carrier frequency and phase recovery.

```
configureMSKSignalRecoveryEx;
recoverTimingPhase = true;
recoverCarrier = true;
```

Modeling Channel Impairments

Specify the sample delay, `timingOffset`, that the channel model applies. Create a variable fractional delay object to introduce the timing delay to the transmitted signal.

```
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
```

Create a `comm.PhaseFrequencyOffset` System object to introduce carrier phase and frequency offsets to a modulated signal. Because the MSK modulator upsamples the transmitted symbols, set the `SampleRate` property to the ratio of the `samplesPerSymbol` and the sample time, `Ts`.

```
freqOffset = 50;
phaseOffset = 30;
pfo = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',freqOffset, ...
    'PhaseOffset',phaseOffset, ...
    'SampleRate',samplesPerSymbol/Ts);
```

Create a `comm.AWGNChannel` System object to add white Gaussian noise to the modulated signal. The noise power is determined by the `EbNo` property, that is the bit energy to noise power spectral density ratio. Because the MSK modulator generates symbols with 1 Watt of power, set the signal power property of the AWGN channel System object to 1.

```
EbNo = 20 + 10*log10(samplesPerSymbol);
chAWGN = comm.AWGNChannel(...
    'NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'EbNo',EbNo,...
    'SignalPower',1, ...
    'SamplesPerSymbol',samplesPerSymbol);
```

Timing Phase, Carrier Frequency, and Carrier Phase Synchronization

Create an MSK timing synchronizer to recover symbol timing phase using a fourth-order nonlinearity method.

```
timeSync = comm.MSKTimingSynchronizer(...
    'SamplesPerSymbol',samplesPerSymbol, ...
    'ErrorUpdateGain',0.02);
```

Create a carrier synchronizer to recover both carrier frequency and phase. Because the MSK constellation is QPSK with a 0-degree phase offset, set the `comm.CarrierSynchronizer` accordingly.

```
phaseSync = comm.CarrierSynchronizer(...
    'Modulation','QPSK', ...
    'ModulationPhaseOffset','Custom', ...
    'CustomPhaseOffset',0, ...
    'SamplesPerSymbol',1);
```

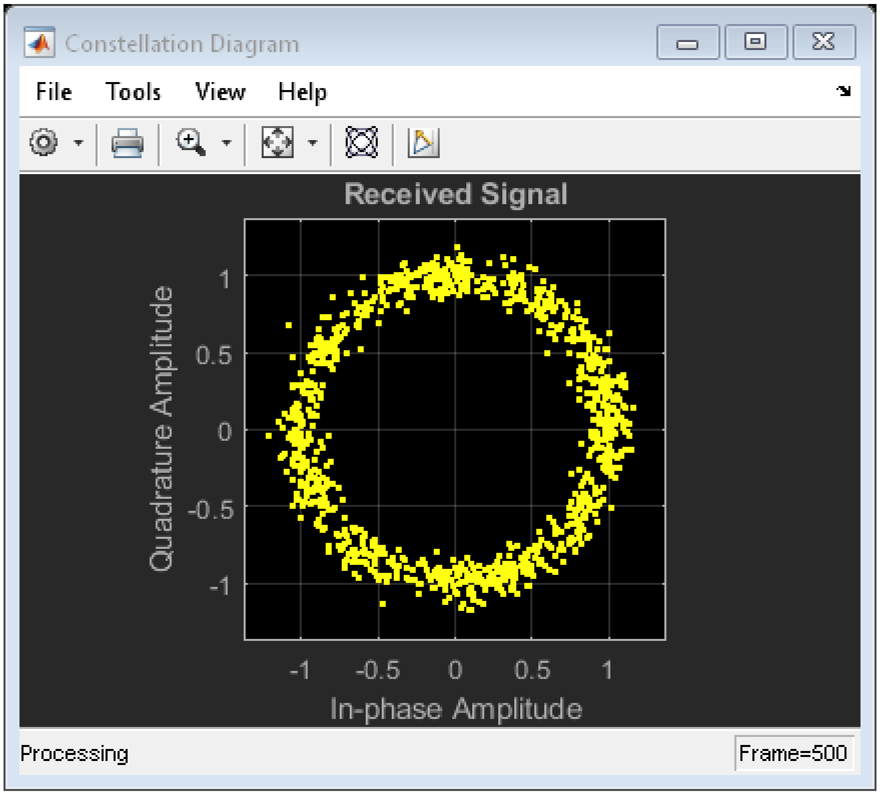
Stream Processing Loop

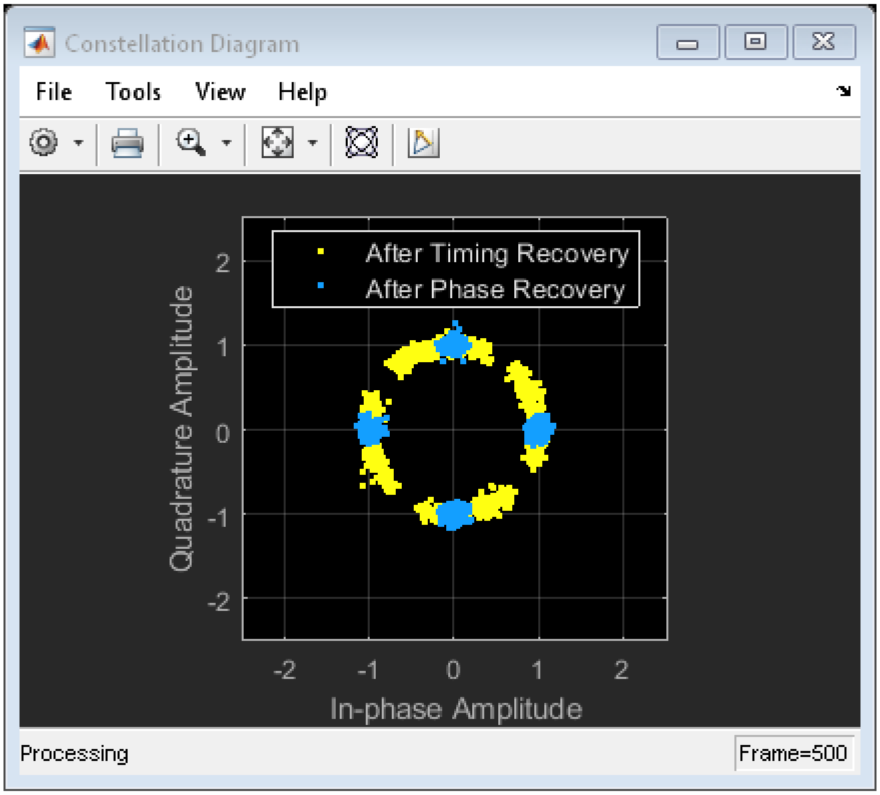
The simulation modulates data using MSK modulation. The modulated symbols pass through the channel model, which applies timing delay, carrier frequency and phase shift, and additive white Gaussian noise. The receiver performs timing phase and carrier frequency and phase recovery. Finally, the signal symbols are demodulated and the bit error rate is calculated. The `plotResultsMSKSignalRecoveryEx` script generates scatter plots in this order to show these effects:

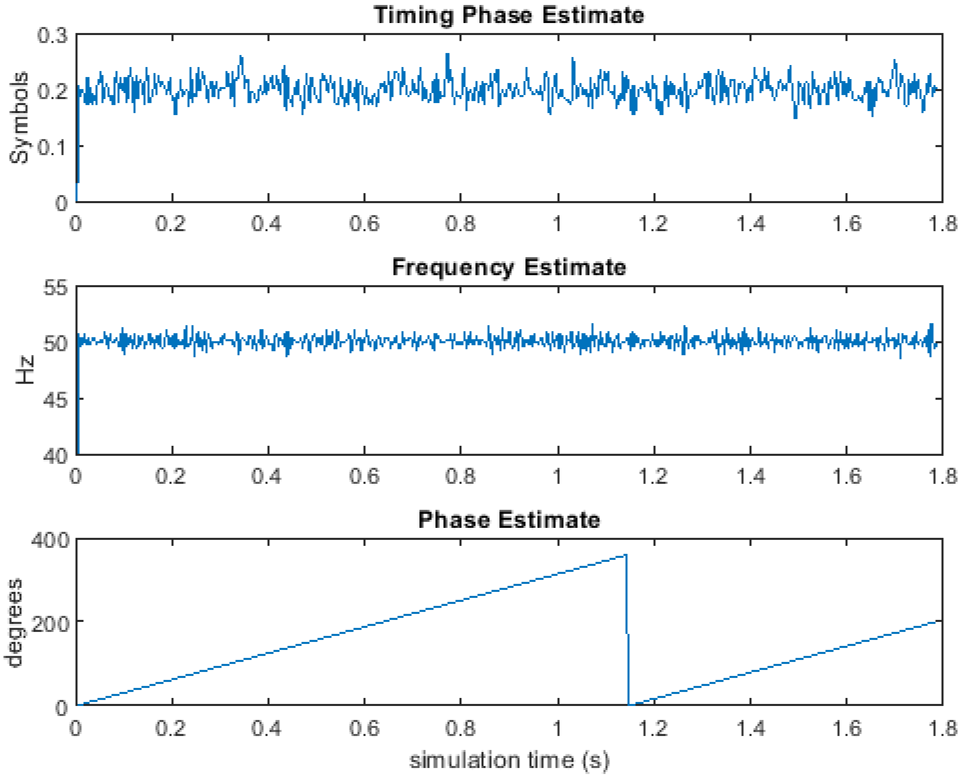
1. Channel impairments
2. Timing synchronization
3. Carrier synchronization

At the end of the simulation, the example displays the timing phase, frequency, and phase estimates as a function of simulation time.

```
for p = 1:numFrames
    %------------------------------------------------------------------------
    % Generate and modulate data
    %------------------------------------------------------------------------
    txBits = randi([0 1],samplesPerFrame,1);
    txSym = modem(txBits);
    %------------------------------------------------------------------------
    % Transmit through channel
    %------------------------------------------------------------------------
    %
    % Add timing offset
    rxSigTimingOff = varDelay(txSym,timingOffset*samplesPerSymbol);
    %
    % Add carrier frequency and phase offset
    rxSigCFO = pfo(rxSigTimingOff);
    %
    % Pass the signal through an AWGN channel
    rxSig = chAWGN(rxSigCFO);
    %
    % Save the transmitted signal for plotting
    plot_rx = rxSig;
    %
    %------------------------------------------------------------------------
    % Timing recovery
    %------------------------------------------------------------------------
    if recoverTimingPhase
        % Recover symbol timing phase using fourth-order nonlinearity
        % method
        [rxSym,timEst] = timeSync(rxSig);
        % Calculate the timing delay estimate for each sample
        timEst = timEst(1)/samplesPerSymbol;
    else
        % Do not apply timing recovery and simply downsample the received
        % signal
        rxSym = downsample(rxSig,samplesPerSymbol);
        timEst = 0;
    end

    % Save the timing synchronized received signal for plotting
    plot_rxTimeSync = rxSym;

    %------------------------------------------------------------------------
    % Carrier frequency and phase recovery
    %------------------------------------------------------------------------
    if recoverCarrier
        % The following script applies carrier frequency and phase recovery
        % using a second order phase-locked loop (PLL), and removes phase ambiguity
        [rxSym,phEst] = phaseSync(rxSym);
        removePhaseAmbiguityMSKSignalRecoveryEx;
        freqShiftEst = mean(diff(phEst)/(Ts*2*pi));
        phEst = mod(mean(phEst),360); % in degrees
    else
        freqShiftEst = 0;
        phEst = 0;
    end

    % Save the phase synchronized received signal for plotting
    plot_rxPhSync = rxSym;
    %-----------------------------------------------------------------------
    % Demodulate the received symbols
    %-----------------------------------------------------------------------
    rxBits = demod(rxSym);
    %-----------------------------------------------------------------------
    % Calculate the bit error rate
    %-----------------------------------------------------------------------
    errorStats = BERCalc(txBits,rxBits);
    %-----------------------------------------------------------------------
    % Plot results
    %-----------------------------------------------------------------------
    plotResultsMSKSignalRecoveryEx;
end
```




Constellation Diagram
File
Tools
View
Help
After Timing Recovery
After Phase Recovery
Quadrature Amplitude
2
1
0
-1
-2
-2
-1
0
1
2
In-phase Amplitude
Processing
Frame=500



Display the bit error rate and the total number of symbols processed by the error rate calculator.

```
BitErrorRate = errorStats(1)
TotalNumberOfSymbols = errorStats(3)
```

```
BitErrorRate =

   4.0001e-06


TotalNumberOfSymbols =

      499982
```

Conclusion and Further Experimentation

The recovery algorithms are demonstrated by using constellation plots taken after timing, carrier frequency, and carrier phase synchronization.

Open the script to create a writable copy of this example and its supporting files. Then, to show the effects of the recovery algorithms, you can enable and disable the logical control variables `recoverTimingPhase` and `recoverCarrier` and rerun the simulation.

Appendix

This example uses these scripts:

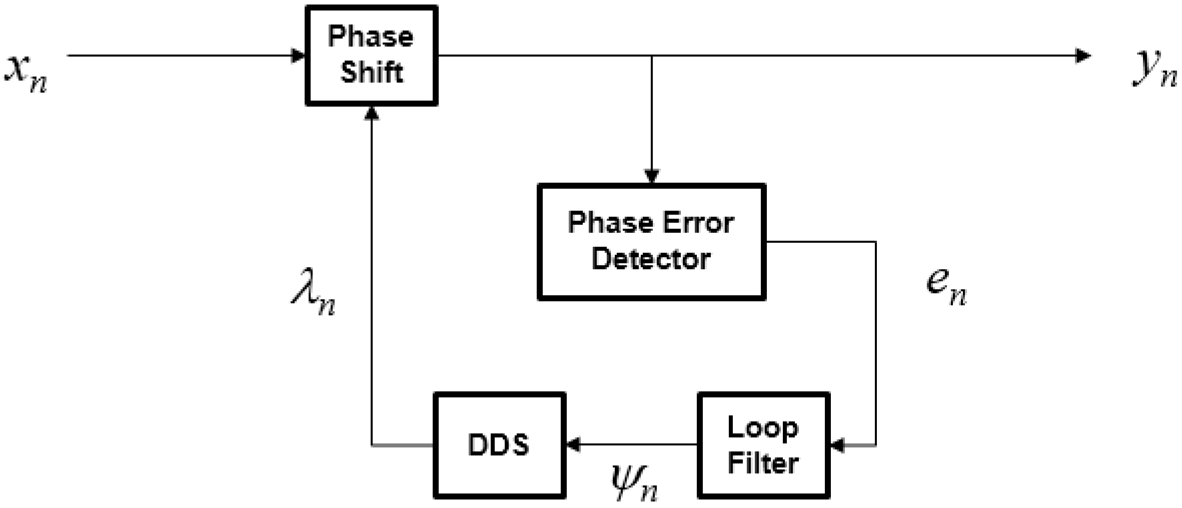
- `configureMSKSignalRecoveryEx`
- `plotResultsMSKSignalRecoveryEx`
- `removePhaseAmbiguityMSKSignalRecoveryEx`

Algorithms

The `comm.CarrierSynchronizer` System object is a closed-loop compensator that uses the PLL-based algorithm described in [1]. The output of the synchronizer, y_n, is a frequency-shifted version of the complex input signal, x_n, for the nth sample. The synchronizer output is

$$y_n = x_n e^{i\lambda_n},$$

where λ_n is the output of the direct digital synthesizer (DDS). The DDS is the discrete-time version of a voltage-controlled oscillator and is a core component of discrete-time phase locked loops. In the context of this System object, the DDS works as an integration filter.



To correct for the frequency offset, first the algorithm determines the phase error, e_n. The value of the phase error depends on the modulation scheme.

Modulation	Phase Error
QAM or QPSK	$e_n = \text{sgn}(\text{Re}\{x_n\}) \times \text{Im}\{x_n\} - \text{sgn}(\text{Im}\{x_n\}) \times \text{Re}\{x_n\}$ For a detailed description of this equation, see [1].
BPSK or PAM	$e_n = \text{sgn}(\text{Re}\{x_n\}) \times \text{Im}\{x_n\}$ For a detailed description of this equation, see [1].

Modulation	Phase Error
8-PSK	$e_{\text{n}} =$ $\begin{cases} \text{sgn}(\text{Re}\{x_{\text{n}}\}) \times \text{Im}\{x_{\text{n}}\} - (\sqrt{2}-1)\text{sgn}(\text{Im}\{x_{\text{n}}\}) \times \text{Re}\{x_{\text{n}}\}, & \text{for} \\ (\sqrt{2}-1)\text{sgn}(\text{Re}\{x_{\text{n}}\}) \times \text{Im}\{x_{\text{n}}\} - \text{sgn}(\text{Im}\{x_{\text{n}}\}) \times \text{Re}\{x_{\text{n}}\}, & \text{for} \end{cases}$ For a detailed description of this equation, see [2].
OQPSK	$e_{\text{n}} = \text{sgn}\left(\text{Re}\left\{x_{\text{n-SamplePerSymbol/2}}\right\}\right)$ $\times \text{Im}\left\{x_{\text{n-SamplePerSymbol/2}}\right\} - \text{sgn}(\text{Im}\{x_{\text{n}}\})$ $\times \text{Re}\{x_{\text{n}}\}$

To ensure system stability, the phase error passes through a biquadratic loop filter governed by

$$\psi_{\text{n}} = g_{\text{I}} e_{\text{n}} + \psi_{\text{n}-1} \,,$$

where ψ_{n} is the output of the loop filter at sample n, and g_{I} is the integrator gain. The integrator gain is determined from the equation

$$g_{\text{I}} = \frac{4\left(\theta^2/d\right)}{K_{\text{p}}K_0} \quad ,$$

where θ, d, K_0, and K_{p} are determined from the System object properties. Specifically,

$$\theta = \frac{B_{\text{n}}}{\left(\zeta + \frac{1}{4\zeta}\right)} \text{ and } d \;=\; 1 + 2\zeta\theta + \theta^2,$$

where B_{n} is the normalized loop bandwidth, and ζ is the damping factor. The phase recovery gain, K_0, is equal to the number of samples per symbol. The modulation type determines the phase error detector gain, K_{p}.

Modulation	K_{p}
BPSK, PAM, QAM, QPSK, or OQPSK	2
8-PSK	1

The output of the loop filter is then passed to the DDS. The DDS is another biquadratic loop filter whose expression is based on the forward Euler integration rule

$$\lambda_{\text{n}} = (g_{\text{P}} e_{\text{n-1}} + \psi_{\text{n-1}}) + \lambda_{\text{n-1}} \,,$$

where g_{P} is the proportional gain that is expressed as

$$g_{\text{P}} = \frac{4\zeta(\theta/d)}{K_{\text{p}}K_0} \;.$$

The `info` object function of this System object returns estimates of the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay. The normalized pull-in range, $(\Delta f)_{\text{pull-in}}$, is expressed in radians and estimated as

$$(\Delta f)_{\text{pull-in}} \approx \min(1, 2\pi\sqrt{2}\zeta B_{\text{n}}) \;.$$

The expression for $(\Delta f)_{\text{pull-in}}$ becomes less accurate as $2\pi\sqrt{2}\zeta B_{\text{n}}$ approaches 1.

The maximum frequency lock delay, T_{FL}, and phase lock delay, T_{PL}, are expressed in samples and estimated as

$$T_{\text{FL}} \approx 4\frac{(\Delta f)^2_{\text{pull-in}}}{B_{\text{n}}^3} \text{ and } T_{\text{PL}} \approx \frac{1.3}{B_{\text{n}}} .$$

References

[1] Rice, M. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 359–393.

[2] Zhijie, H., Y. Zhiqiang, Z. Ming, and W. Kuang. “8PSK Demodulation for New Generation DVB-S2.” *2004 International Conference on Communications, Circuits and Systems.* Vol. 2, 2004, pp. 1447–1450.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.CoarseFrequencyCompensator` | `comm.PhaseFrequencyOffset` | `comm.SymbolSynchronizer`

Blocks

Carrier Synchronizer

Topics

“QPSK Transmitter and Receiver”

Introduced in R2015a

comm.CCDF

Package: `comm`

Measure complementary cumulative distribution function

Description

The `comm.CCDF` object measures the probability that the instantaneous power of the input signal being a specified level above its average power.

To measure complementary cumulative distribution of a signal:

1. Define and set up your CCDF object. See "Construction" on page 3-287 .
2. Call `step` to measure complementary cumulative distribution according to the properties of `comm.CCDF`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`ccdf = comm.CCDF` creates a complementary cumulative distribution function measurement (CCDF) System object, `ccdf`, that measures the probability of a signal's instantaneous power to be a specified level above its average power.

`ccdf = comm.CCDF(Name,Value)` creates a CCDF object, `ccdf`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`NumPoints`

Number of CCDF points

Specify the number of CCDF points that the object calculates. This property requires a numeric, positive, integer scalar. The default is `1000`. Use this property with the `MaximumPowerLimit` on page 3-0 property to control the size of the histogram bins. The object uses these bins to estimate CCDF curves. This controls the resolution of the curves. All input channels must have the same number of CCDF points.

`MaximumPowerLimit`

Maximum expected input signal power

Maximum expected input signal power limit for each input channel, specified as a scalar or row vector with length equal to the number of input channels. The default is `50`. This property is expected to have the same units as the PowerUnits property. When you set this property to a scalar, the object assumes that the signals in all input channels have the same expected maximum power. When you set this property to a row vector, the object assumes that the *i*-th element of the vector is the maximum expected power for the signal at the *i*-th input channel. For each input channel, the object obtains CCDF results by integrating a histogram of instantaneous input signal powers. The object sets the bins of the histogram so that the last bin collects all power occurrences that are equal to, or greater than the power that you specify in this property. The object issues a warning if any input signal exceeds its specified maximum power limit. Use this property with the NumPoints property to control the size of the histogram bins that the object uses to estimate CCDF curves (such as control the resolution of the curves).

PowerUnits

Power units

Specify the power measurement units as one of dBm | dBW | Watts. The default is dBm. The step method outputs power measurements in the units specified in the PowerUnits property. When you set this property to dBm or dBW, the step method outputs relative power values in a dB scale. When you set this property to Watts, the step method outputs relative power values in a linear scale. When you call the step method, the object assumes that the units of MaximumPowerLimit have the same value you specified in the PowerUnits property.

AveragePowerOutputPort

Enable average power measurement output

When you set this property to true, running the object returns the running average power measurements. The default is false.

PeakPowerOutputPort

Enable peak power measurement output

When you set this property to true, running the object returns the peak power measurements. The default is false.

PAPROutputPort

Enable PAPR measurement output

When you set this property to true, running the object returns the peak-to-average-power measurements. The default is false.

Methods

getPercentileRelativePower	Get relative power value for a given probability
getProbability	Get the probability for a given relative power value
plot	Plot CCDF curves
reset	Reset states of CCDF measurement object
step	Measure complementary cumulative distribution function

Common to All System Objects	
release	Allow System object property value changes

Examples

Obtain CCDF curves for 16-QAM and QPSK signals

Create a CCDF System object and specify that it output average power and peak power measurements.

```
ccdf = comm.CCDF('AveragePowerOutputPort',true, ...
    'PeakPowerOutputPort',true);
```

Generate 16-QAM and QPSK modulated signals.

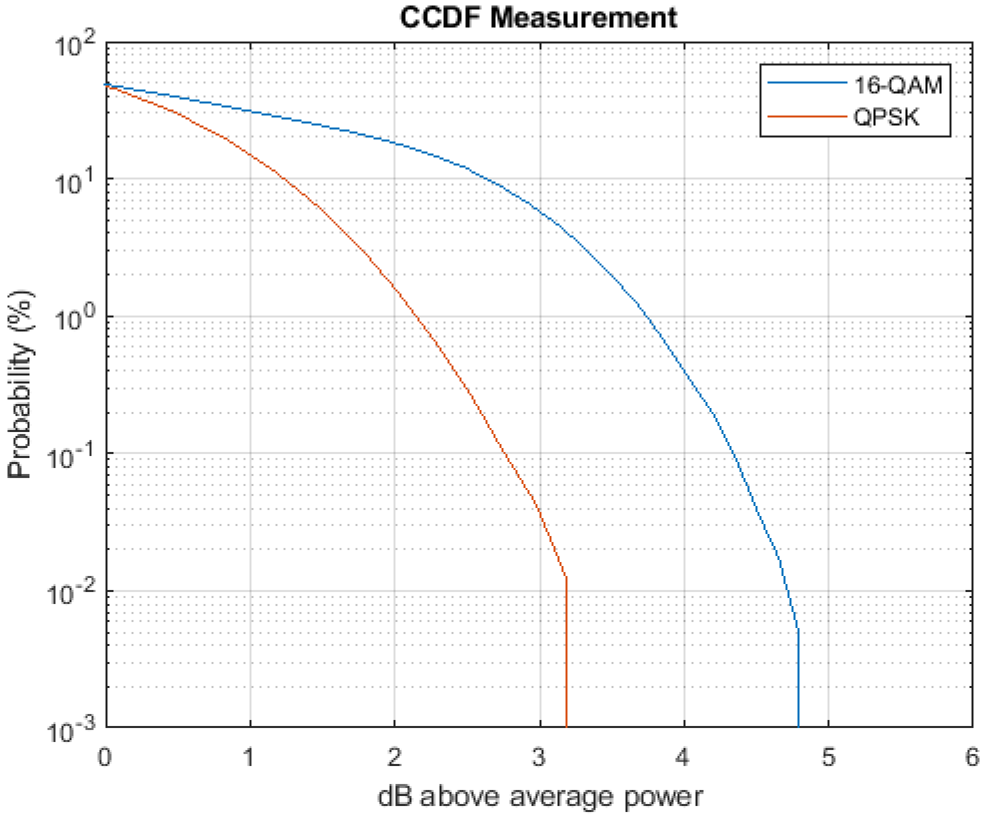
```
qamTxSig = qammod(randi([0 15],20e3,1),16,'UnitAveragePower',true);
qpskTxSig = pskmod(randi([0 3],20e3,1),4,pi/4);
```

Pass the signals through an AWGN channel.

```
qamRxSig = awgn(qamTxSig,15);
qpskRxSig = awgn(qpskTxSig,15);
```

Measure the CCDF of the two waveforms. Plot the CCDF using the `plot` method of `comm.CCDF`.

```
[CCDFy,CCDFx,AvgPwr,PeakPwr] = ccdf([qamRxSig qpskRxSig]);

plot(ccdf)
legend('16-QAM','QPSK')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.EVM` | `comm.MER`

Introduced in R2012a

getPercentileRelativePower

System object: `comm.CCDF`
Package: `comm`

Get relative power value for a given probability

Syntax

```
R = getPercentileRelativePower(H,P)
```

Description

`R = getPercentileRelativePower(H,P)` finds the relative power values, `R`. The power of the signal of interest is above its average power by `R` dB (if PowerUnits equals 'dBW', or 'dBm') or by a factor of `R` (in linear scale if PowerUnits equals 'Watts') with a probability `P`.

The method output `R`, is a column vector with the *i*-th element corresponding to the relative power for the *i*-th input channel. The method input `P` can be a double precision scalar, or a vector with a number of elements equal to the number of input channels. If `P` is a scalar, then all the relative powers in `R` correspond to the same probability value specified in `P`. If `P` is a vector, then the *i*-th element of `R` corresponds to a power value that occurs in the *i*-th input channel, with a probability specified in the *i*-th element of `P`.

For the *i*-th input channel, this method evaluates the inverse CCDF curve at probability value $P(i)$.

Examples

Obtain CCDF curves for a unit variance AWGN signal and a dual- one signal. The AWGN signal is RPW1 dB above its average power one percent of the time, and the dual-tone signal is RPW2 dB above its average power 10 percent of the time. This example finds the values of RPW1 and RPW2.

```
n = [0:5e3-1].';
s1 = randn(5e3,1);                     % AWGN signal
s2 = sin(0.01*pi*n)+sin(0.03*pi*n);   % dual-tone signal
hCCDF = comm.CCDF;                     % create a CCDF object
step(hCCDF,[s1 s2]);                   % step the CCDF measurements
plot(hCCDF)                            % plot CCDF curves
legend('AWGN','Dual-tone')
RPW = getPercentileRelativePower(hCCDF,[1 10]);
RPW1 = RPW(1)
RPW2 = RPW(2)
```

getProbability

System object: comm.CCDF
Package: comm

Get the probability for a given relative power value

Syntax

P = getProbability(H,R)

Description

P = getProbability(H,R) finds the probability, P, of the power level of the signal of interest being R dBs (if PowerUnits equals 'dBW', or 'dBm') or Watts (if PowerUnits equals 'Watts') above its average power. P is a column vector with the *i*-th element corresponding to the probability value for the *i*-th input channel. Input R can be a double precision scalar or a vector with a number of elements equal to the number of input channels. If R is a scalar, then all the probability values in P correspond to the same relative power specified in R. If R is a vector, then the ith element of P contains a probability value for the *i*-th channel and for the relative power specified in the *i*-th element of R.

For the *i*-th input channel, this method evaluates the CCDF curve at relative power value R(*i*)

Examples

Obtain CCDF curves for a unit variance AWGN signal and a dual- tone signal. Find the probability that the AWGN signal power is 5 dB above its average power and that the dual-tone signal power is 3 dB above its average power.

```
n = [0:5e3-1].';
s1 = randn(5e3,1);                   % AWGN signal
s2 = sin(0.01*pi*n)+sin(0.03*pi*n);  % dual-tone signal
hCCDF = comm.CCDF;
step(hCCDF,[s1 s2]);
plot(hCCDF)                          % plot CCDF curves
legend('AWGN','Dual-tone')
P = getProbability(hCCDF,[5 3])      % get probabilities
```

plot

System object: `comm.CCDF`
Package: `comm`

Plot CCDF curves

Syntax

`D = plot(H)`

Description

`D = plot(H)` plots CCDF measurements in the CCDF System object, `H`. The `plot` method returns the plot handles as an output, `D`. This method plots the same number of curves as there are input channels. The `H` input can be followed by parameter-value pairs to specify additional properties of the curves. For example, `plot`(`H`,`LineWidth`,`2`) will create curves with line widths of 2 points.

The `comm.CCDF` System object does not support C code generation for this method.

reset

System object: `comm.CCDF`
Package: `comm`

Reset states of CCDF measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CCDF` object, `H`.

step

System object: `comm.CCDF`
Package: `comm`

Measure complementary cumulative distribution function

Syntax

```
[CCDFY,CCDFX] = step(H,X)
[CCDFY,CCDFX,AVG] = step(H,X)
[CCDFY,CCDFX,PEAK] = step(H,X)
[CCDFY,CCDFX,PAPR] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[CCDFY,CCDFX] = step(H,X)` updates CCDF, average power, and peak power measurements for input `X` using the CCDF System object, `H`. It outputs the y-axis, `CCDFY`, and x-axis, `CCDFX`, CCDF points. `X` must be a double precision, *M*-by-*N* matrix, where *M* is the number of time samples and *N* is the number of input channels. The `step` method outputs `CCDFY` as a matrix whose *i*-th column contains updated probability values measured from the *i*-th column of input matrix `X`. `CCDFY` contains the *y*-axis points of the CCDF curves of each channel. The `step` method outputs `CCDFX` as a matrix containing, in its *i*-th column, the corresponding updated instantaneous-to-average power ratios for the ith column of input matrix `X`. `CCDFX` contains the *x*-axis points of the CCDF curves of each channel. The object sets the number of rows in `CCDFY` and `CCDFX` equal to `NumPoints` property + 1. The probability values are percentages in the `[0 100]` interval. When you set the `PowerUnits` property to `dBW` or `dBm`, the relative powers are in dB scale. When you set the `PowerUnits` property to `Watts`, the relative powers are in linear scale. Measurements are updated each time you call the step method until you reset the object. You call the plot method to plot CCDF curves for each channel.

`[CCDFY,CCDFX,AVG] = step(H,X)` returns updated average power measurements, `AVG`, when you set the `AveragePowerOutputPort` property to true. The step method outputs `AVG` as a column vector with the ith element corresponding to an updated average power measurement for the signal available in the ith column of input matrix `X`. You specify the units for `AVG` in the `PowerUnits` property.

`[CCDFY,CCDFX,PEAK] = step(H,X)` returns updated peak power measurements, `PEAK`, when you set the `PeakPowerOutputPort` property to true. The step method outputs `PEAK` as a column vector with the ith element corresponding to an updated peak power measurement for the signal available in the ith column of input matrix `X`. You specify the units for `PEAK` in the `PowerUnits` property.

`[CCDFY,CCDFX,PAPR] = step(H,X)` returns updated peak-to-average power ratio measurements, `PAPR`, when you set the `PAPROutputPort` property to true. The step methods outputs `PAPR` as a column vector with the ith element corresponding to an updated peak-to-average power ratio measurement for the signal available in the ith column of input matrix `X`. When you set the

PowerUnits property to dBW or dBm, the method outputs PAPR in a dB scale. When you set the PowerUnits property to Watts, the method outputs PAPR in a linear scale. You can combine optional output arguments when you set their enabling properties. Optional outputs must be listed in the same order as the order of the enabling properties. For example,

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.CoarseFrequencyCompensator

Package: `comm`

Compensate for frequency offset for PAM, PSK, or QAM

Description

The `CoarseFrequencyCompensator` System object compensates for the frequency offset of received signals.

To compensate for the frequency offset of a PAM, PSK, or QAM signal:

1. Define and set up your coarse frequency compensator object. See "Construction" on page 3-297.
2. Call `step` to compensate for the frequency offset of a PAM, PSK, or QAM signal according to the properties of `comm.CoarseFrequencyCompensator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`CFC = comm.CoarseFrequencyCompensator` creates a coarse frequency offset compensator object, `CFC`. This object uses an open-loop technique to estimate and compensate for the carrier frequency offset in a received signal.

`CFC = comm.CoarseFrequencyCompensator(Name,Value)` creates a coarse frequency offset compensator object, `CFC`, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Properties

`Modulation`

Modulation type

Specify the signal modulation type as `BPSK`, `QPSK`, `OQPSK`, `8PSK`, `PAM`, or `QAM`. The default is `QAM`. This property is nontunable.

`Algorithm`

Algorithm used to estimate frequency offset

Specify the estimation algorithm as one of `FFT-based` or `Correlation-based`. The default is `FFT-based`. This property is nontunable.

The table shows the allowable combinations of the modulation type and the estimation algorithm.

Modulation	FFT-Based Algorithm	Correlation-Based Algorithm
BPSK, QPSK, 8PSK, PAM	✓	✓
OQPSK, QAM	✓	

Use the correlation-based algorithm for HDL implementations and for other situations in which you want to avoid using an FFT.

This property appears when `Modulation` is `'BPSK'`, `'QPSK'`, `'8PSK'`, or `'PAM'`.

`FrequencyResolution`

Frequency resolution (Hz)

Specify the frequency resolution for the offset frequency estimation as a positive, real scalar of data type `double`. This property establishes the FFT length used to perform spectral analysis and must be less than the sample rate. The default is `0.001`. This property is nontunable.

`MaximumFrequencyOffset`

Maximum measurable frequency offset (Hz)

Specify the maximum measurable frequency offset as a positive, real scalar of data type `double`.

The value of this property must be less than f_{samp} / M, where f_{samp} is the sample rate and M is the modulation order. As a best practice, set `MaximumOffset` to less than $r / (4M)$. This property applies only if `Algorithm` is `Correlation-based`. The default is `0.05`. This property is nontunable.

`SampleRate`

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type `double`. The default is `1`. This property is nontunable.

`SamplesPerSymbol`

Samples per symbol

Specify the number of samples per symbol, s, as a real positive finite integer scalar, such that $s \geq 2$. The default value is `4`. This property is nontunable.

This property appears when `Modulation` is `'OQPSK'`.

Methods

info	Characteristic information about coarse frequency compensator
reset	Reset states of the `CoarseFrequencyCompensator` object
step	Compensate for frequency offset

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Compensate for Frequency Offset in a QPSK Signal

Compensate for a 4 kHz frequency offset imposed on a noisy QPSK signal.

Set the example parameters.

```
nSym = 2048;       % Number of input symbols
sps = 4;           % Samples per symbol
nSamp = nSym*sps;  % Number of samples
fs = 80000;        % Sampling frequency (Hz)
```

Create a square root raised cosine transmit filter.

```
txfilter = comm.RaisedCosineTransmitFilter(...
    'RolloffFactor',0.2, ...
    'FilterSpanInSymbols',8,   ...
    'OutputSamplesPerSymbol',sps);
```

Create a phase frequency offset object to introduce the 4 kHz frequency offset.

```
freqOffset = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',-4000, ...
    'SampleRate',fs);
```

Create a coarse frequency compensator object to compensate for the offset.

```
freqComp = comm.CoarseFrequencyCompensator(...
    'Modulation','QPSK', ...
    'SampleRate',fs, ...
    'FrequencyResolution',1);
```

Generate QPSK symbols, filter the modulated data, pass the signal through an AWGN channel, and apply the frequency offset.

```
data = randi([0 3],nSym,1);
modData = pskmod(data,4,pi/4);
txSig = txfilter(modData);
rxSig = awgn(txSig,20,'measured');
offsetData = freqOffset(rxSig);
```

Compensate for the frequency offset using `freqComp`. When the frequency offset is high, it is beneficial to do coarse frequency compensation prior to receive filtering because filtering suppresses energy in the useful spectrum.

```
[compensatedData,estFreqOffset] = freqComp(offsetData);
```

Display the estimate of the frequency offset.

```
estFreqOffset
```

```
estFreqOffset =

  -3.9999e+03
```

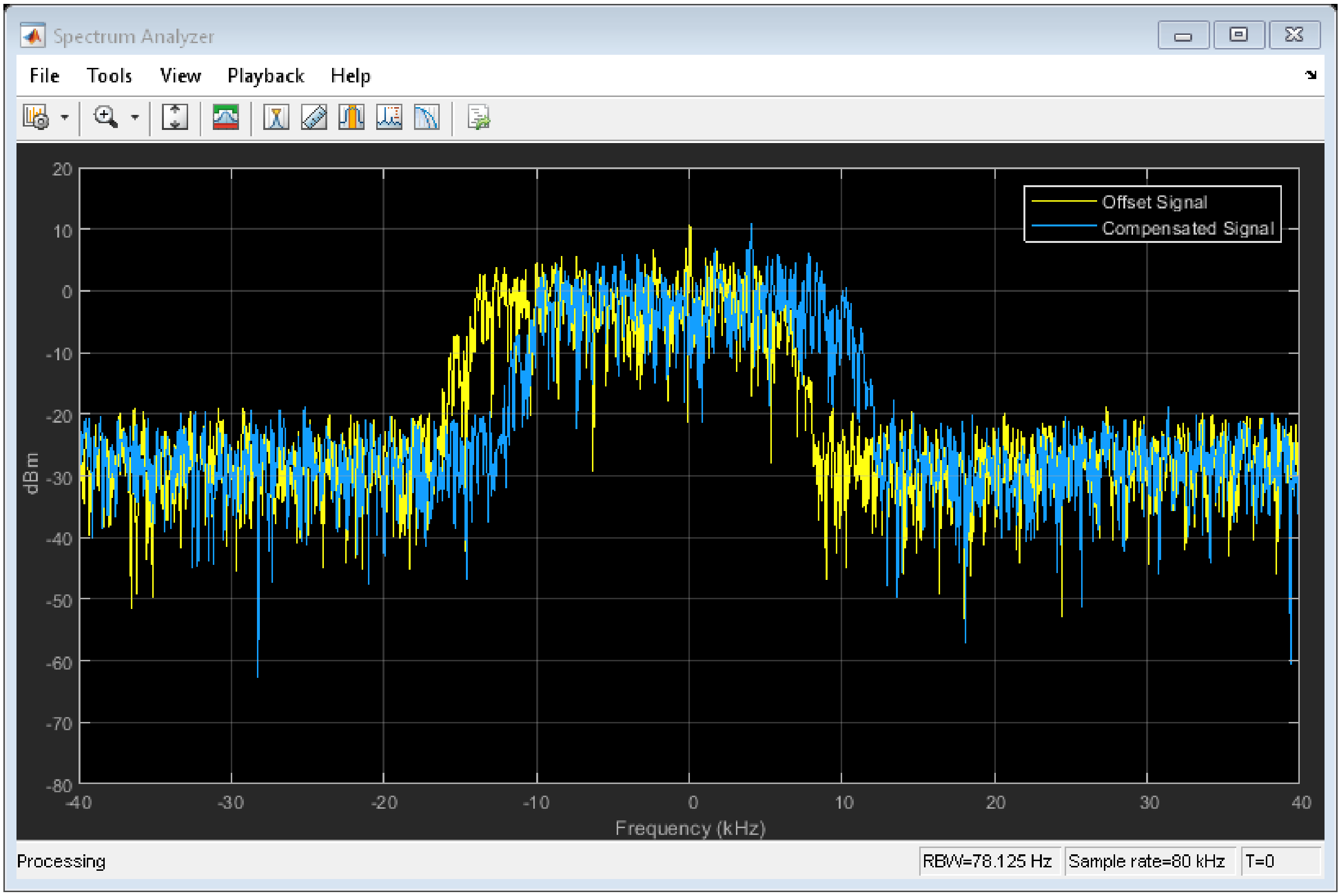
Return information about the `freqComp` object. To obtain the FFT length, you must call `freqComp` prior to calling the `info` method.

```
freqCompInfo = info(freqComp)
```

```
freqCompInfo = 

  struct with fields:

    FFTLength: 131072
    Algorithm: 'FFT-based'
```

Create a spectrum analyzer object and plot the offset and compensated spectra. Verify that the compensated signal has a center frequency at 0 Hz and that the offset signal has a center frequency at -4 kHz.

```
specAnal = dsp.SpectrumAnalyzer('SampleRate',fs,'ShowLegend',true, ...
    'ChannelNames',{'Offset Signal' 'Compensated Signal'});
specAnal([offsetData compensatedData])
```



Compensate for Frequency Offset Using Coarse and Fine Compensation

Correct for a phase and frequency offset in a noisy QAM signal using a carrier synchronizer. Then correct for the offsets using both a carrier synchronizer and a coarse frequency compensator.

Set the example parameters.

```
fs = 10000;           % Symbol rate (Hz)
sps = 4;              % Samples per symbol
M = 16;               % Modulation order
k = log2(M);          % Bits per symbol
```

Create a QAM modulator and an AWGN channel.

```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',k,'SamplesPerSymbol',sps);
```

Create a constellation diagram object to visualize the effects of the offset compensation techniques. Specify the constellation diagram to display only the last 4000 samples.

```
constdiagram = comm.ConstellationDiagram(...
    'ReferenceConstellation',qammod(0:M-1,M), ...
    'SamplesPerSymbol',sps, ...
    'SymbolsToDisplaySource','Property','SymbolsToDisplay',4000, ...
    'XLimits',[-5 5],'YLimits',[-5 5]);
```

Introduce a frequency offset of 400 Hz and a phase offset of 30 degrees.

```
phaseFreqOffset = comm.PhaseFrequencyOffset(...
    'FrequencyOffset',400,...
    'PhaseOffset',30,...
    'SampleRate',fs);
```

Generate random data symbols and apply 16-QAM modulation.

```
data = randi([0 M-1],10000,1);
modSig = qammod(data,M);
```

Create a raised cosine filter object and filter the modulated signal.

```
txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',sps, ...
    'Gain',sqrt(sps));
txSig = txfilter(modSig);
```

Apply the phase and frequency offset, and then pass the signal through the AWGN channel.

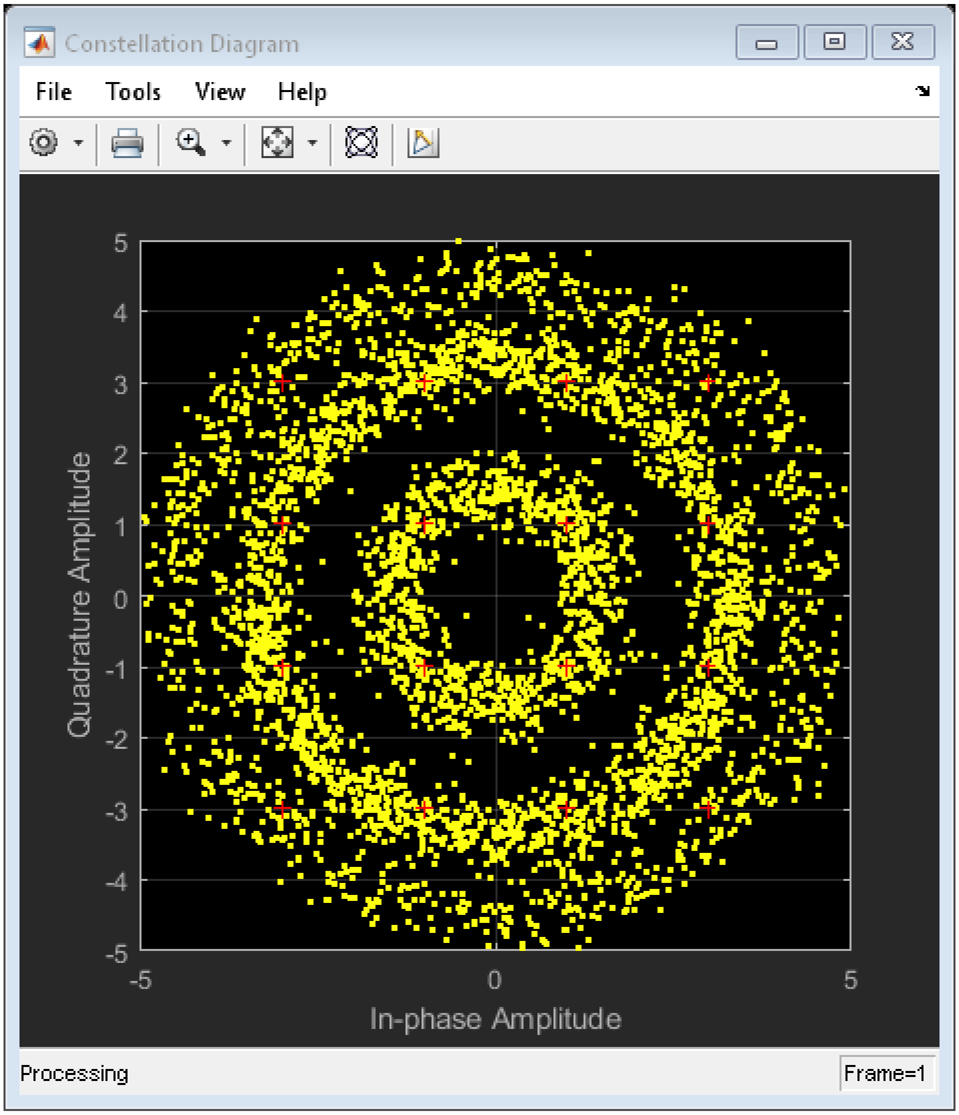
```
freqOffsetSig = phaseFreqOffset(txSig);
rxSig = channel(freqOffsetSig);
```

Apply fine frequency correction to the signal by using the carrier synchronizer.

```
fineSync = comm.CarrierSynchronizer('DampingFactor',0.7, ...
    'NormalizedLoopBandwidth',0.005, ...
    'SamplesPerSymbol',sps, ...
    'Modulation','QAM');
rxData = fineSync(rxSig);
```

Display the constellation diagram of the last 4000 symbols.

```
constdiagram(rxData)
```



Even with time to converge, the spiral nature of the plot shows that the carrier synchronizer has not yet compensated for the large frequency offset. The 400 Hz offset is 1% of the sample rate.

Repeat the process with a coarse frequency compensator inserted before the carrier synchronizer.

Create a coarse frequency compensator to reduce the frequency offset to a manageable level.

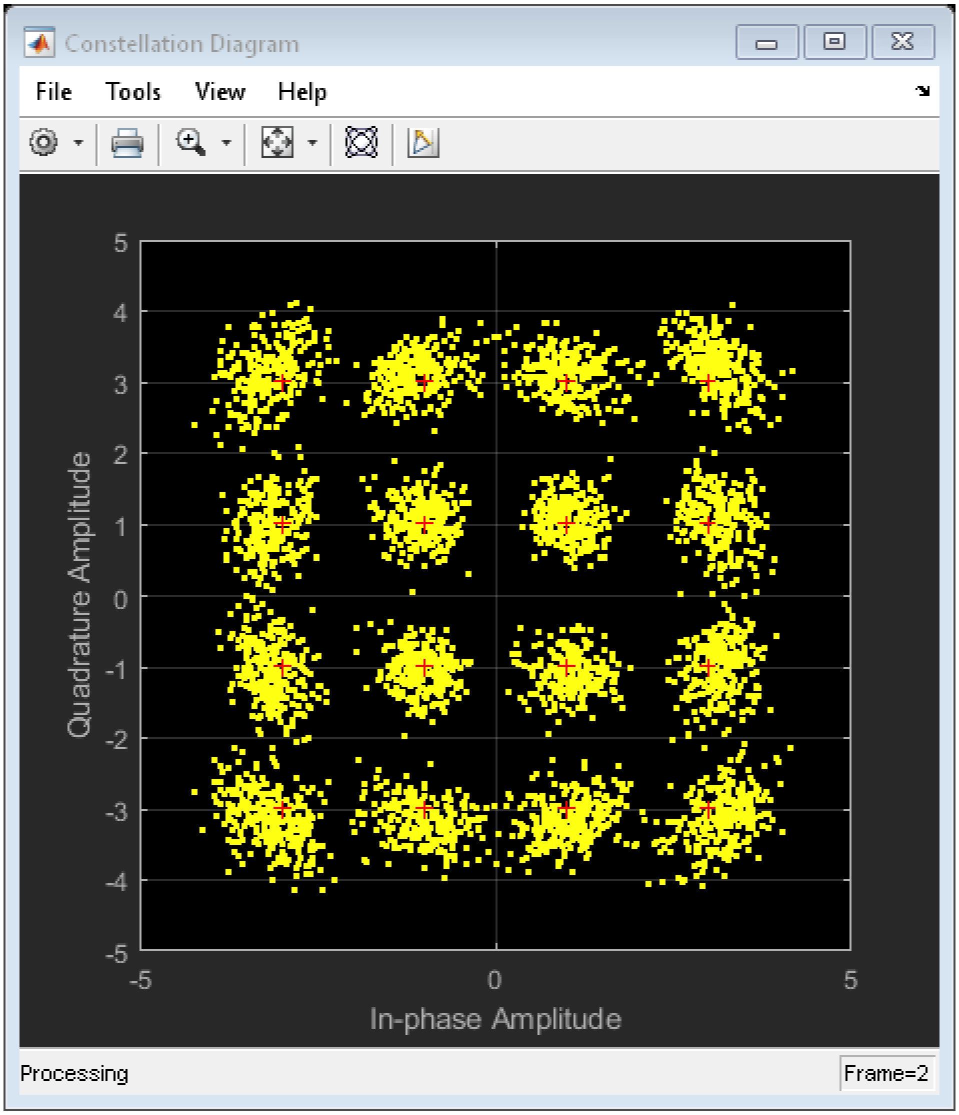
```
coarseSync = comm.CoarseFrequencyCompensator('Modulation','QAM','FrequencyResolution',1,'SampleRa
```

Pass the received signal to the coarse frequency compensator and then to the carrier synchronizer.

```
syncCoarse = coarseSync(rxSig);
rxData = fineSync(syncCoarse);
```

Plot the constellation diagram of the signal after coarse and fine frequency compensation.

```
constdiagram(rxData)
```



The received data now aligns with the reference constellation.

Algorithms

Correlation-Based

The correlation-based estimation algorithm, which can be used to estimate the frequency offset for PSK and PAM signals, is described in [1]. To determine the frequency offset, *Δf*, the algorithm performs a maximum likelihood (ML) estimation of the complex-valued oscillation exp(*j2πΔft*). The observed signal, r_k, is represented as

$$r_k = e^{j(2\pi\Delta f k \mathrm{T}_S + \theta)},\ 1 \le k \le N\,,$$

where T_s is the sampling interval, θ is an unknown random phase, and N is the number of samples. The maximum likelihood estimation of the frequency offset is equivalent to seeking the maximum of the likelihood function, *Λ(Δf)*,

$$\Lambda(\Delta f) \approx \left|\sum_{i=1}^{N} r_i e^{-j2\pi\Delta f i T_s}\right|^2 = \sum_{k=1}^{N}\sum_{m=1}^{N} r_k r_m^* e^{-j2\pi\Delta f T_s(k-m)} .$$

After simplifying, the problem is expressed as a discrete Fourier transform, weighted by a parabolic windowing function. It is expressed as

$$\mathrm{Im}\left\{\sum_{k=1}^{N-1} k(N-k)R(k)e^{j2\pi\widehat{\Delta f}T_s}\right\} = 0 ,$$

where *R(k)* denotes the estimated autocorrelation of the sequence r_k and is represented as

$$R(k) \triangleq \frac{1}{N-k}\sum_{i=k+1}^{N} r_i r_{i-k}^*,\ 0 \le k \le N-1 .$$

The term *k(N–k)* is the parabolic windowing function. In [1], it is shown that *R(k)* is a poor estimate of the autocorrelation of r_k when k = 0 or when k is close to N. Consequently, the windowing function can be expressed as a rectangular sequence of 1s for k = 1, 2, ..., L, where $L \le N$ – 1. The results is a modified ML estimation strategy in which

$$\mathrm{Im}\left\{\sum_{k=1}^{L} R(k)e^{-j2\pi\widehat{\Delta f}kT_s}\right\} = 0 .$$

This results in an estimate of $\widehat{\Delta f}$ in which

$$\widehat{\Delta f} \cong \frac{f_{samp}}{\pi(L+1)}\arg\left\{\sum_{k=1}^{L} R(k)\right\} .$$

The sampling frequency, f_{samp}, is the reciprocal of T_s. The number of elements used to compute the autocorrelation sequence, L, are determined as

$$L = \mathrm{round}\left(\frac{f_{samp}}{f_{max}}\right) - 1,$$

where f_{max} is the maximum expected frequency offset and `round` is the nearest integer function. The frequency offset estimate improves when $L \ge 7$ and leads to the recommendation that $f_{max} \le f_{samp}$ / (4M).

FFT-Based

FFT-based algorithms can be used to estimate the frequency offset for all modulation types. Two variations are used in comm.CoarseFrequencyCompensator.

- For `BPSK`, `QPSK`, `8PSK`, `PAM`, or `QAM` modulations the FFT-based algorithm used is described in [2]. The algorithm estimates $\widehat{\Delta f}$ by using a periodogram of the m^{th} power of the received signal and is given as

  $$\widehat{\Delta f} = \frac{f_{samp}}{N \cdot m}\underset{f}{\mathrm{argmax}}\left|\sum_{k=0}^{N-1} r^m(k)e^{-j2\pi kt/N}\right|, \quad \left(-\frac{R_{sym}}{2} \le f \le \frac{R_{sym}}{2}\right),$$

  where m is the modulation order, *r(k)* is the received sequence, R_{sym} is the symbol rate, and N is the number of samples. The algorithm searches for a frequency that maximizes the time average of the m^{th} power of the received signal multiplied by various frequencies in the range of [–R_{sym}/2, R_{sym}/2]. As the form of the algorithm is the definition of the discrete Fourier transform of $r^m(t)$, searching for a frequency that maximizes the time average is equivalent to searching for a peak line in the spectrum of $r^m(t)$. The number of points required by the FFT is

$$N = 2^{\left\lceil \log_2\left(\frac{f_{samp}}{f_r}\right)\right\rceil},$$

where f_r is the desired frequency resolution.

- For OQPSK modulation the FFT-based algorithm used is described in [4]. The algorithm searches for spectral peaks at +/- 200 kHz around the symbol rate. This technique locates desired peaks in the presence of interference from spectral content around baseband frequencies due to filtering.

References

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions.” *IEEE® Transactions on Communications*. Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

[2] Wang, Y., K. Shi, and E. Serpedi. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach.” *EURASIP Journal on Applied Signal Processing*. 2004:13, pp. 1993–2001.

[3] Nakagawa, T., M. Matsui, T. Kobayashi, K. Ishihara, R. Kudo, M. Mizoguchi, and Y. Miyamoto. “Non-Data-Aided Wide-Range Frequency Offset Estimator for QAM Optical Coherent Receivers.” *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*. March 2011, pp. 1–3.

[4] Olds, Jonathan. "*Designing an OQPSK demodulator*".

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer` | `comm.PhaseFrequencyOffset` | `dsp.FFT`

Introduced in R2015b

info

System object: `comm.CoarseFrequencyCompensator`
Package: `comm`

Characteristic information about coarse frequency compensator

Syntax

```
S = info(CFC)
```

Description

`S = info(CFC)` returns a structure, `S`, containing characteristic information for the `CoarseFrequencyCompensator` System object, `CFC`. `S` has fields `FFTLength`, `Algorithm`, and `MaxLag`. `Algorithm` is the type of algorithm used in estimating the frequency offset. `FFTLength` is the number of samples used in the FFT and is provided when the algorithm is `FFT-based`. `MaxLag` is the number of samples used in estimating the autocorrelation and is provided when the algorithm is `Correlation-based`.

Note The `step` method must be run once to determine the `FFTLength`.

reset

System object: `comm.CoarseFrequencyCompensator`
Package: `comm`

Reset states of the `CoarseFrequencyCompensator` object

Syntax

```
reset(CFC)
```

Description

`reset(CFC)` resets the internal states of the `CoarseFrequencyCompensator` object, `CFC`.

step

System object: `comm.CoarseFrequencyCompensator`
Package: `comm`

Compensate for frequency offset

Syntax

```
Y = step(CFC,X)
[Y,EST] = step(CFC,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(CFC,X)` compensates for the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a column vector. The step method outputs the compensated signal `Y` as a complex column vector having the same dimensions and data type as `X`.

`[Y,EST] = step(CFC,X)` returns a scalar estimate of the frequency offset, `EST`.

Note `CFC` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.ChannelFilter

Package: `comm`

Filter signal using multipath gains at specified path delays

Description

Use the `comm.ChannelFilter` System object to filter a signal using multipath gains at specified path delays.

To filter a signal using multipath gains:

1. Create the `comm.ChannelFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
chanFilt = comm.ChannelFilter
chanFilt = comm.ChannelFilter(Name,Value)
```

Description

`chanFilt = comm.ChannelFilter` creates a multipath channel filter System object to filter an input signal with path gains at the specified path delays

`chanFilt = comm.ChannelFilter(Name,Value)` sets properties using one or more name-value pairs. For example, `'SampleRate',1e6` sets the sampling rate to 1 MHz. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Sample rate
1 Hz (default) | real positive scalar

Sample rate of the input signal, specified as a real, positive scalar.

Data Types: `double`

`PathDelays` — Discrete path delays
`0` (default) | real scalar | real vector

Delays of the discrete paths in seconds, specified as a real scalar or vector.

Data Types: `double`

`FilterDelaySource` — Channel filter delay source
`'Auto'` (default) | `'Custom'`

Channel filter delay source, specified as either `'Auto'` or `'Custom'`.

- Set `FilterDelaySource` to `'Auto'` to specify the channel filter delay as the minimum possible value.
- Set `FilterDelaySource` to `'Custom'` to specify the channel filter delay as a custom value. The custom value cannot be smaller than the minimum possible value.

Data Types: `char`

`FilterDelay` — Channel filter delay
7 (default) | real non-negative integer scalar

Channel filter delay in samples, specified as a real, non-negative, integer scalar.

Dependencies

To enable this property, set the `FilterDelaySource` property to `'Custom'`. The specified value must be no smaller than the automatically determined channel filter delay when you set `FilterDelaySource` to `'Auto'`.

Data Types: `double`

`NormalizeChannelOutputs` — Normalize outputs by number of receive antennas
`'true'` or `1` (default) | `'false'` or `0`

Normalize outputs by number of receive antennas, specified as either `'true'` (`1`) or `'false'` (`0`).

Data Types: `logical`

Usage

Syntax

```
y = chanFilt(x,g)
```

Description

`y = chanFilt(x,g)` filters input signal `x`, through a multipath channel with path gains `g`, at the path delay locations specified by the PathDelays property.

Input Arguments

`x` — Input signal
matrix

Input signal, specified as a matrix. The argument `x` must be a N_s-by-N_t matrix, where N_s is the number of samples and N_t is the number of transmit antennas.

Data Types: `double`

g — Path gain
array

Path gain, specified as an array. The input G must be a N_s-by-N_p-by-N_t-by-N_r or 1-by-N_p-by-N_t-by-N_r array, where N_r is the number of receive antennas and N_p is the number of paths, i.e., the length of the PathDelays property.

Data Types: `double`

Output Arguments

y — Channel output
matrix

Channel output, returned as a N_s-by-N_r matrix.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.ChannelFilter`

info	Return characteristic information about channel filter

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Explore Spatial Diversity of Channel in Distributed MIMO System

In a distributed MIMO system, send the same signal from two geographically separate transmitters (Tx) and combine the received signals at one single receiver (Rx) to explore spatial diversity. The two Txs are not co-located and they experience different multipath (path delays) to the Rx. Specify the path delays respectively.

```
delay1 = [0 1.5 2.3 5.2 6.6];
delay2 = [0 3.7 6.2];
```

Configure one channel filter object per Tx.

```
chanFilt1 = comm.ChannelFilter('PathDelays', delay1);
chanFilt2 = comm.ChannelFilter('PathDelays', delay2);
```

The two channel filters get different filter delays.

```
fd1 = chanFilt1.info.ChannelFilterDelay
```

```
fd1 = 6
```

```
fd2 = chanFilt2.info.ChannelFilterDelay
```

```
fd2 = 4
```

In order to combine the two channel outputs at Rx, the two channel filters must have the same filter delay. Customize the filter delay for each object.

```
set(chanFilt1, ...
    "FilterDelaySource", 'Custom', ...
    "FilterDelay", max(fd1, fd2));
set(chanFilt2, ...
    "FilterDelaySource", 'Custom', ...
    "FilterDelay", max(fd1, fd2));
```

Set up system parameters. It is assumed one antenna at each Tx and Rx.

```
[Nt, Nr] = deal(1);
Ns  = 30720;
Np1 = length(chanFilt1.PathDelays);
Np2 = length(chanFilt2.PathDelays);
M   = 64;
```

Generate random 64-QAM signal and complex path gains for the two channels.

```
x  = qammod(randi([0, M-1], Ns, Nt), M);
g1 = complex(rand(Ns, Np1, Nt, Nr), rand(Ns, Np1, Nt, Nr));
g2 = complex(rand(Ns, Np2, Nt, Nr), rand(Ns, Np2, Nt, Nr));
```

Perform channel filtering and Rx combining. The combined 2x1 MIMO channel has a filter delay of max(`fd1`, `fd2`).

```
y = chanFilt1(x, g1) + chanFilt2(x, g2);
```

Perform Channel Filtering for an LTE 2x2 EVA Profile

Construct a channel filter object with the LTE Extended Vehicular A model (EVA) delay profile.

```
chanFilt = comm.ChannelFilter( ...
    'SampleRate', 30.72e6, ...
    'PathDelays', [0 30 150 310 370 710 1090 1730 2510]*1e-9);
```

Set up system parameters. There are two transmit and receive antennas.

```
[Nt, Nr] = deal(2);
Ns = 30720;
Np = length(chanFilt.PathDelays);
M  = 256;
```

Generate random 256-QAM signal and complex path gains.

```
x = qammod(randi([0, M-1], Ns, Nt), M);
g = complex(rand(Ns, Np, Nt, Nr), rand(Ns, Np, Nt, Nr));
```

Filter the signal with path gains for the EVA delay profile.

```
y = chanFilt(x, g);
```

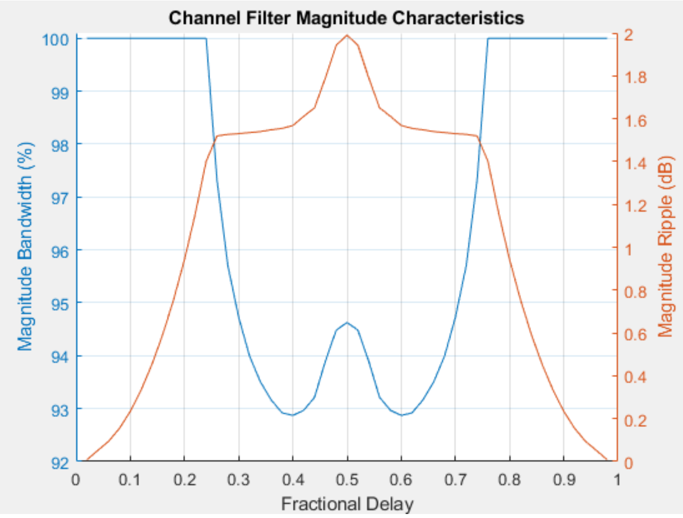
Algorithms

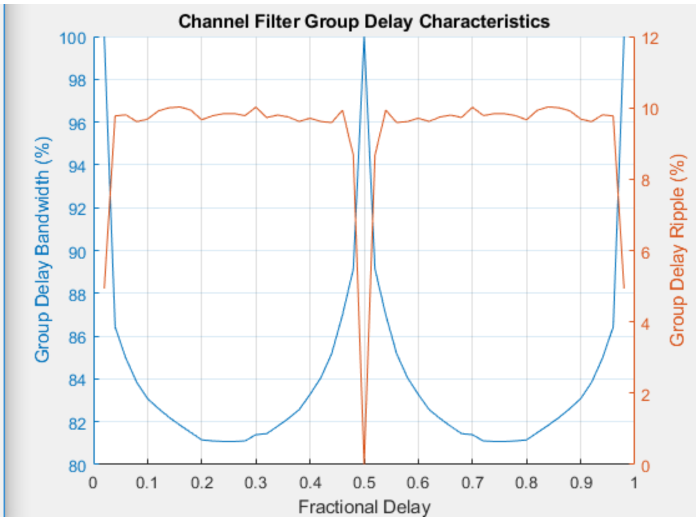
The channel filter implements a fractional delay (FD) finite impulse response (FIR) bandpass filter with a length of 16 coefficients for each candidate fractional delay at 0, 0.02, 0.04, ..., 0.98.

Each discrete path is rounded to its nearest candidate fractional delay, so the delay error limit is 1% of the sample time. To achieve a group delay bandwidth exceeding 80% and a magnitude bandwidth exceeding 90%, the algorithm selects the optimal FIR coefficient values for each fractional delay, while satisfying the following criteria:

- Group delay ripple ≤ 10%
- Magnitude ripple ≤ 2 dB
- Magnitude bandedge attenuation = 3 dB

The plots show bandwidths that satisfy the design criteria for group delay ripple, magnitude ripple, and magnitude bandedge attenuation.





For additional information, see the article *A Matlab-based Object-Oriented Approach to Multipath Fading Channel Simulation* at MATLAB Central.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
comm.MIMOChannel | comm.RayTracingChannel | comm.RayleighChannel |
comm.RicianChannel

Introduced in R2020b

comm.ConstellationDiagram

Package: comm

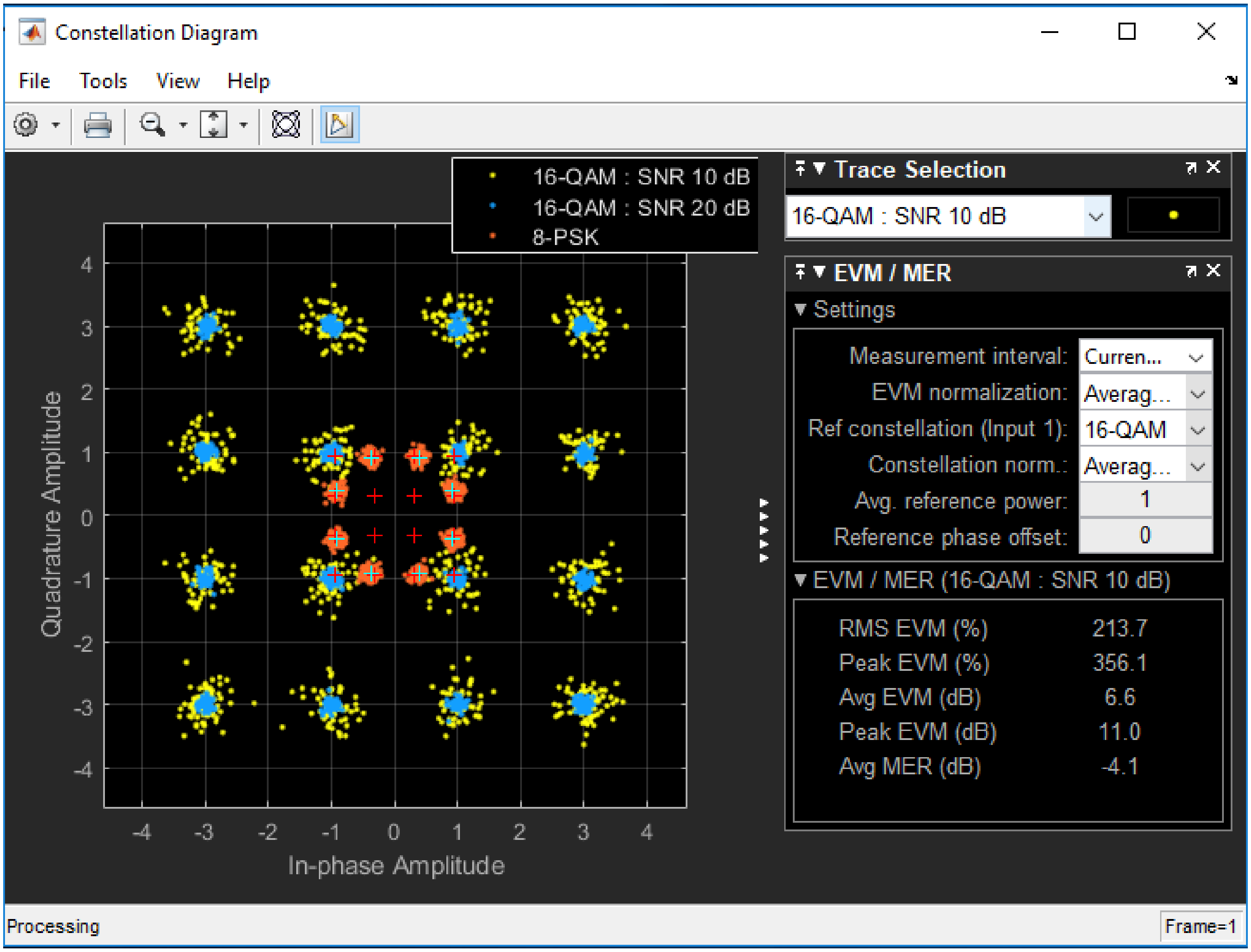
Display constellation diagram for input signals

Description

The comm.ConstellationDiagram System object displays real and complex-valued floating and fixed-point signals in the IQ plane. Use this System object to perform qualitative and quantitative analysis on modulated single-carrier signals.

In the constellation diagram window you can:

- Input and plot multiple signals on a single constellation diagram. You can define one reference constellation for each input signal. For more information, see ReferenceConstellation.
- Choose which channels are displayed by selecting signals in the legend. Use the ShowLegend property to display the legend. For a multichannel signal, specify the input as a matrix with individual signals defined in the columns of the matrix.
- Display the "EVM / MER Measurements" on page 3-328 panel, which displays calculated error vector magnitude (EVM) and modulation error ratio (MER) measurements. When multiple signals are input to a comm.ConstellationDiagram System object, use the **Trace Selection** pane to choose the signal being measured.



Trace Selection: 16-QAM : SNR 10 dB

EVM / MER

Settings

Measurement interval: Curren...
EVM normalization: Averag...
Ref constellation (Input 1): 16-QAM
Constellation norm.: Averag...
Avg. reference power: 1
Reference phase offset: 0

EVM / MER (16-QAM : SNR 10 dB)

RMS EVM (%)	213.7
Peak EVM (%)	356.1
Avg EVM (dB)	6.6
Peak EVM (dB)	11.0
Avg MER (dB)	-4.1

Processing

Frame=1

To display constellation diagrams:

1 Create the `comm.ConstellationDiagram` object and set its properties.
2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
constdiag = comm.ConstellationDiagram
constdiag = comm.ConstellationDiagram(Name,Value)
```

Description

`constdiag = comm.ConstellationDiagram` returns a `comm.ConstellationDiagram` System object that displays real and complex-valued floating and fixed-point signals in the IQ plane.

`constdiag = comm.ConstellationDiagram(Name,Value)` set the properties of the System object using one or more name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Example: `constdiag = comm.ConstellationDiagram('SampleOffset',1e3)` specifies that the first `1000` samples received will not be displayed.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Name — Title of Constellation Diagram window

`'Constellation Diagram'` (default) | character vector

Title of the Constellation Diagram window, specified as a character vector.

`SamplesPerSymbol` — Number of samples used to represent each symbol

`1` (default) | positive integer

Number of samples used to represent each symbol, specified as a positive integer. When the `SamplesPerSymbol` property is greater than 1, the signal is downsampled before it is plotted.

Tunable: Yes

`SampleOffset` — Number of samples to skip before plotting points

`0` (default) | nonnegative integer

Number of samples to skip before plotting points, specified as a nonnegative integer less than SamplesPerSymbol. This property specifies the number of samples to skip when downsampling the input signal.

Tunable: Yes

`SymbolsToDisplaySource` — Source of symbols to display

`'Input frame length'` (default) | `'Property'`

Source of symbols to display, specified as:

- `'Input frame length'` — The number of symbols to display is equal to the input frame length divided by SamplesPerSymbol.
- `'Property'` — SymbolsToDisplay specifies the maximum number of symbols to display.

Tunable: Yes

SymbolsToDisplay — Maximum number of symbols to display

256 (default) | positive integer

Maximum number of symbols to display, specified as a positive integer. Use SymbolsToDisplay to limit the maximum number of symbols displayed when long signals are input. Symbols plotted are the most recent symbols received.

Tunable: Yes

Dependencies

This property applies when SymbolsToDisplaySource is set to 'Property'.

ReferenceConstellation — Reference constellations

[0.7071+0.7071i -0.7071+0.7071i -0.7071-0.7071i 0.7070-0.7071i] (default) | row vector | cell array

Reference constellations for the input signals, specified as a row vector or cell array of vectors defining the ideal constellation points for each input signal. Input signals can be single channel or multichannel. You can define one reference constellation for each input signal. For multichannel input signals, one reference constellation specification applies to all the individual signals in that input signal. To obtain the "EVM / MER Measurements" on page 3-328, you must set the **ReferenceConstellation** property.

Tunable: Yes

Data Types: double
Complex Number Support: Yes

ReferenceMarker — Marker for reference display

'+' (default) | string | cell array

Specify the marker for the reference display as a string or cell array of strings. Select the marker symbol as one of the markers in this table.

Value	Description
'o'	Circle
'+'	Plus sign
'*'	Asterisk
'.'	Point
'x'	Cross
'_'	Horizontal line
'|'	Vertical line
'square' or 's'	Square
'diamond' or 'd'	Diamond
'^'	Upward-pointing triangle
'v'	Downward-pointing triangle
'>'	Right-pointing triangle
'<'	Left-pointing triangle

Value	Description
`'pentagram'` or `'p'`	Five-pointed star (pentagram)
`'hexagram'` or `'h'`	Six-pointed star (hexagram)
`'none'`	No markers

Tunable: Yes

Data Types: `string`

`ReferenceColor` — Color for reference display constellation

`[1 0 0]` (red) (default) | row vector | cell array

Color for reference display constellation, specified as a three-element row vector indicating RGB component colors or as a cell array containing RGB component colors for each input signal.

Data Types: `double`

`ShowReferenceConstellation` — Display the reference constellation

`true` (default) | `false`

Display the reference constellation, specified as `true` or `false`.

Tunable: Yes

Data Types: `logical`

`ShowTrajectory` — Turn on signal trajectory plot

`false` (default) | `true`

Turn on signal trajectory plot, specified as `false` or `true`. The signal trajectory is a plot of the in-phase component versus the quadrature component of a modulated signal. See the **Show Signal Trajectory** button on the toolbar in the “Signal Display” on page 3-327.

Tunable: Yes

Data Types: `logical`

`Position` — Scope window position and size

410-by-300 pixel window at center of screen (default) | four-element vector

Scope window position and size in pixels, specified as a four-element vector of the form [*left bottom width height*]. The first two elements in the vector indicate the location of the lower left corner and the last two specify the size of the window. The default value for the location depends on the screen resolution. By default, the window is positioned in the center of the screen with a width and height of 410 and 300 pixels, respectively.

Tunable: Yes

Data Types: `double`

`NumInputPorts` — Number of input ports

`1` (default) | integer in the range [1, 20]

Specify the number of input ports, as an integer in the range [1, 20]. Each input signal, whether it is a multichannel signal or a single channel signal, becomes a separate input port in the scope.

When multichannel input signals are specified, the maximum number of input ports is limited by the total number of input signals defined. The total number of input signals cannot exceed 20.

ShowGrid — Turn on grid

true (default) | false

Turn on the grid, specified as true or false.

Tunable: Yes

Data Types: logical

ChannelNames — Names for input channels

empty cell (default) | cell array of strings or character vectors

Names for input channels, specified as a cell array of strings or character vectors. If you do not specify names, the channels are labeled as Channel 1, Channel 2, etc.

The names assigned to input channels appear in the legend and the **Measurements** > **Trace Selection** pane.

To show the legend, set ShowLegend to true. The legend displays after you provide an input signal to the comm.ConstellationDiagram System object.

To show the **Trace Selection** pane, select **Measurements** > **Trace Selection**. To enable the **Trace Selection** pane, you must first provide an input signal to the comm.ConstellationDiagram System object.

Example: constDiag = comm.ConstellationDiagram('ChannelNames',{'8-QAM','8-PSK'}) assigns names for two input channels to 8-QAM and 8-PSK.

Tunable: Yes

ShowLegend — Display legend

false (default) | true

Display the legend, specified as false or true. The names listed in the legend are the signal names specified by the ChannelNames property.

From the legend, you can control which signals to plot. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal and hide all other signals, right-click the signal name. To show all signals, press **Esc**.

Tunable: Yes

Data Types: logical

ColorFading — Add color fading effect

false (default) | true

Add color fading effect, specified as false or true. When you set this property to true, the points in the display fade as the interval of time after they are first plotted increases. This animation resembles an oscilloscope display.

Data Types: logical

`Title` — Plot title

blank (default) | character vector | string

Plot title, specified as a character vector or string.

Tunable: Yes

`XLimits` — *x*-axis limits

`[-1.375 1.375]` (default) | two-element numeric vector

x-axis limits, specified as a two-element numeric vector of the form [*xmin xmax*].

Tunable: Yes

`YLimits` — *y*-axis limits

`[-1.375 1.375]` (default) | two-element numeric vector

y-axis limits, specified as a two-element numeric vector of the form [*ymin ymax*].

Tunable: Yes

`XLabel` — *x*-axis label

`'In-phase Amplitude'` (default) | character vector | string

x-axis label, specified as a character vector or string.

Tunable: Yes

`YLabel` — *y*-axis label

`'Quadrature Amplitude'` (default) | character vector | string

y-axis label, specified as a character vector or string.

Tunable: Yes

`EnableMeasurements` — Display measurements pane

`false` (default) | `true`

Display measurements pane, specified as `false` or `true`. To compute and display EVM or MER measurements, activate this pane.

Tunable: Yes

Data Types: `logical`

`MeasurementInterval` — Measurement interval

`'Current Display'` (default) | `'All displays'` | positive integer

Measurement interval, specified as `'Current Display'`, `'All displays'`, or a positive integer in the range [2 SymbolsToDisplay]. This property specifies the window length for the EVM and MER measurements.

When the input signal contains one sample per symbol and the reference constellation is provided, the constellation diagram display can measure the signal quality in terms of EVM and MER. The “EVM / MER Measurements” on page 3-328 pane can be displayed by clicking the **Signal Quality** button. See the toolbar in the “Signal Display” on page 3-327. After the number of input data samples is greater than MeasurementInterval, the EVM and MER measurements are computed.

Tunable: Yes

`EVMNormalization` — EVM normalization method

`'Average constellation power'` (default) | `'Peak constellation power'`

EVM normalization method, specified as `'Average constellation power'` or `'Peak constellation power'`. For more information, see "EVM / MER Measurements" on page 3-328.

Tunable: Yes

Usage

Syntax

`constdiag(signal1,signal2,...,signalN)`

Description

`constdiag(signal1,signal2,...,signalN)` displays up to `NumInputPorts` signals in one constellation diagram.

Input Arguments

`signal` — Input signal or signals to plot

column vector | matrix

Specify one or more signals to be plotted in the `comm.ConstellationDiagram`. Signals can have different data types and dimensions. To create a multichannel signal, specify a matrix with individual signals defined in the columns of the matrix.

Example: `constDiag([siganl1_1,signal1_2],signal2)` displays the multichannel and multi-input constellation diagram. First input is two concatenated column vectors of length N to make an N-by-2 matrix input signal and the second input is a single channel signal.

Data Types: `double`
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to Scopes

show	Show scope window
hide	Hide scope window
isVisible	Determine visibility of scope window

Common to All System Objects

step	Run System object algorithm

release Release resources and allow changes to System object property values and input characteristics

reset Reset internal states of System object

Examples

Plot Amplitude-Imbalanced QPSK Constellation

QPSK modulate random data symbols and apply an amplitude imbalance to the signal. Pass the signal through a noisy channel. Plot the resultant constellation.

Create a constellation diagram object. Because the default reference constellation for the `comm.ConstellationDiagram` System object is QPSK, it is not necessary to set additional properties.

```
constDiagram = comm.ConstellationDiagram;
```

Generate random data symbols and apply QPSK modulation.

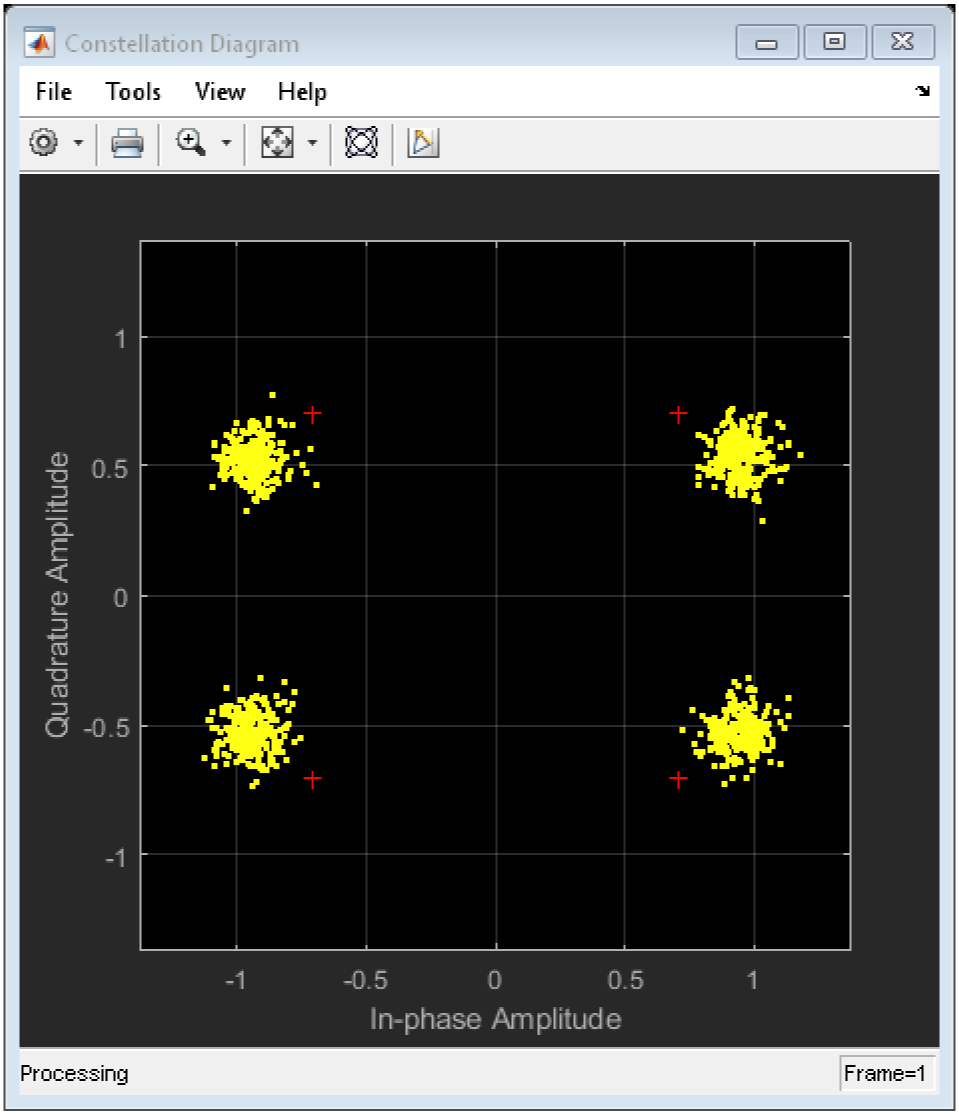
```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Apply an amplitude imbalance to the modulated signal.

```
txSig = iqimbal(modData,5);
```

Pass the transmitted signal through an AWGN channel and display the constellation diagram. Observe that the data points have shifted from their ideal locations.

```
rxSig = awgn(txSig,20);
constDiagram(rxSig)
```



Plot 16-QAM Constellation

Apply 16-QAM modulation, transmit data using an AWGN channel, and plot the signal constellation.

Create a 16-QAM reference constellation.

```
M = 16;
refC = qammod(0:M-1,M);
```

Create a `comm.ConstellationDiagram` System object. Specify the constellation reference points and axes limits using name-value pairs.

```
constDiagram = comm.ConstellationDiagram('ReferenceConstellation',refC, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Generate random, 16-ary data symbols.

```
data = randi([0 M-1],1000,1);
```

Apply 16-QAM modulation.

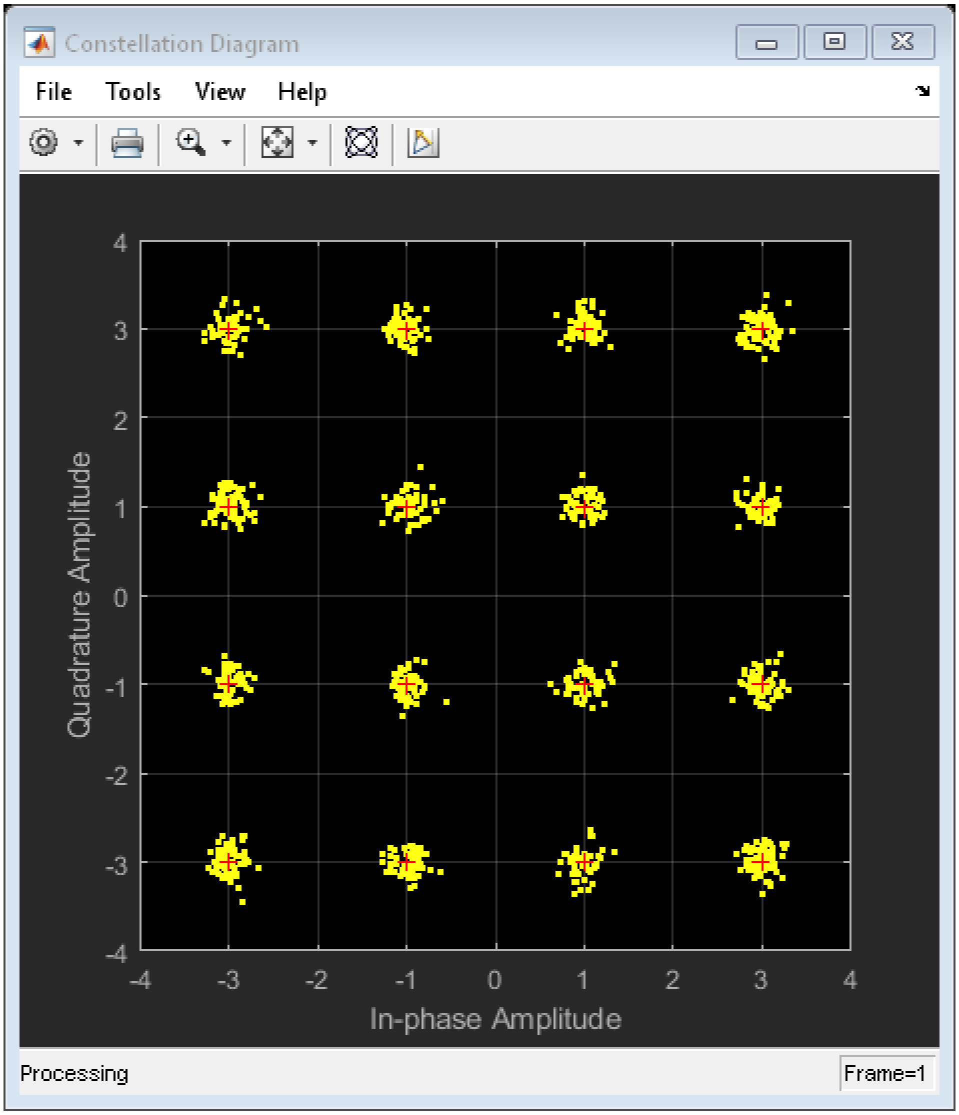
```
sym = qammod(data,M);
```

Pass the modulated signal through an AWGN channel.

```
rcv = awgn(sym,15);
```

Display the constellation diagram.

```
constDiagram(rcv)
```



View Multi-Input Signals Constellation

Use the `comm.ConstellationDiagram` System object to visualize the constellation of multi-input and multichannel modulated signals. Plot a multichannel signal with two 16-QAM signals with SNR 10 and 20 for the first input, and one 8-PSK signal for the second input.

Create a 16-QAM and a 8-PSK reference constellation.

```
M = 16;
refQAM = qammod(0:M-1,M);
S = 8;
refPSK = pskmod(0:S-1,S,pi/8);
```

Create a comm.ConstellationDiagram object™.

```
constDiag = comm.ConstellationDiagram(2,...
      'ReferenceConstellation',{refQAM,refPSK},'ShowLegend',true,...
      'XLimits',[-6 6],'YLimits',[-6 6], ...
      'ChannelNames',{'16-QAM , SNR 10 dB','16-QAM , SNR 20 dB','8-PSK'});
```

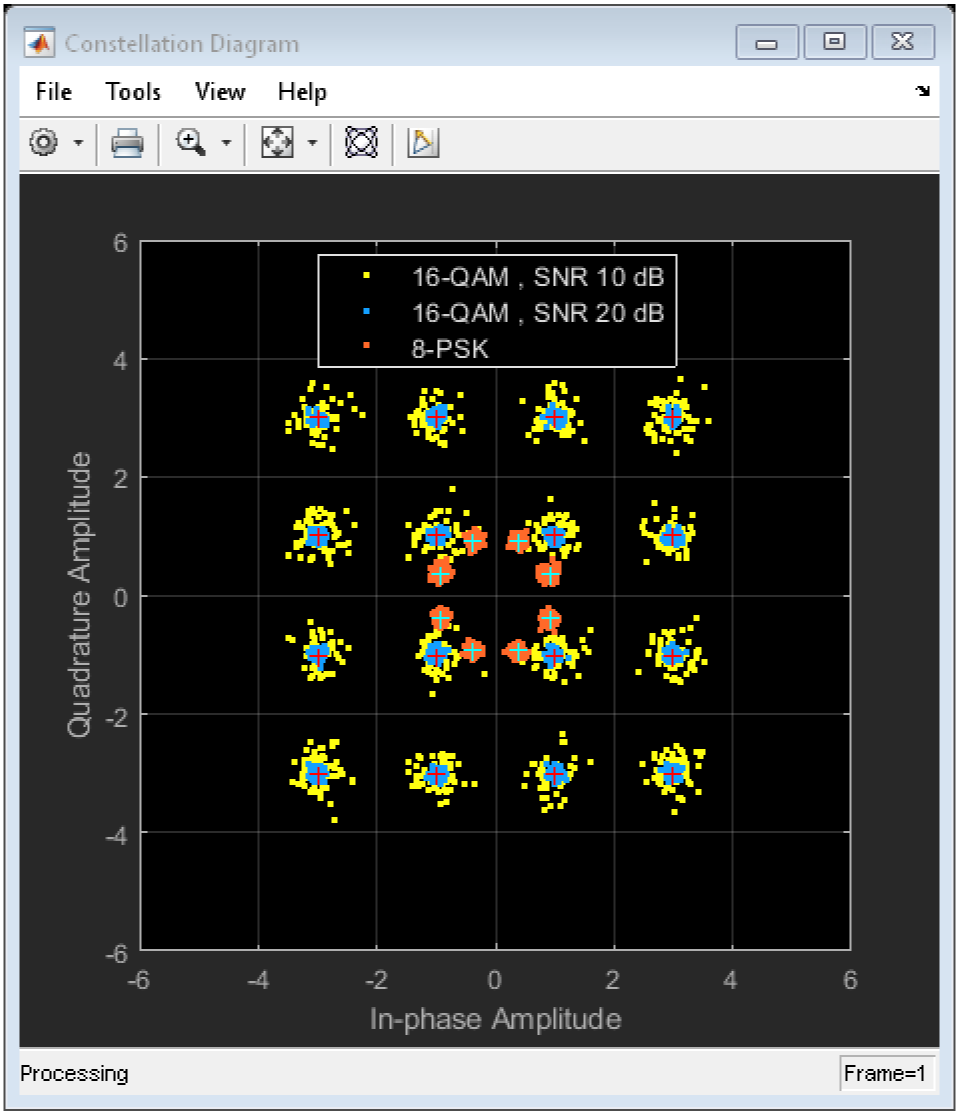
Generate random data symbols, modulate the symbols, and add AWGN with two different SNRs to yield two received signals.

```
d = randi([0 M-1],1000,1);
dQAM = qammod(d,M);
rcv1_1 = awgn(dQAM,10);
rcv1_2 = awgn(dQAM,20);
d = randi([0 S-1],1000,1);
dPSK = pskmod(d,S,pi/8);
rcv2 = awgn(dPSK,20);
```

For the first input, create a multichannel signal by concatenating the two received 16-QAM signals. A single reference constellation is applied for all the multichannel signals of one input. Second input uses a single channel 8-PSK signal. This input has a separate reference constellation.

View the multi-input and multichannel signals .

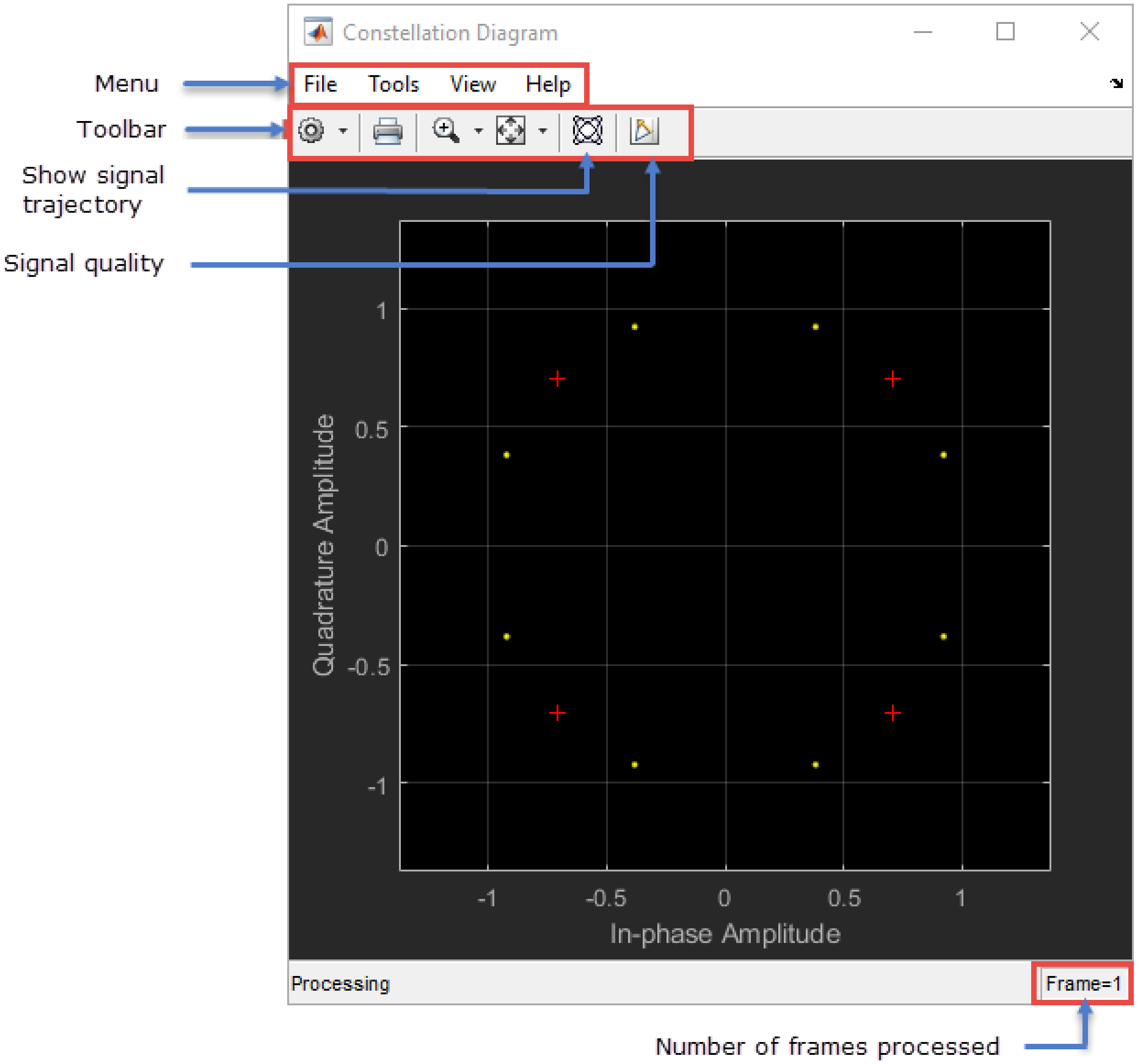
```
constDiag([rcv1_1,rcv1_2],rcv2);
```



More About

Signal Display

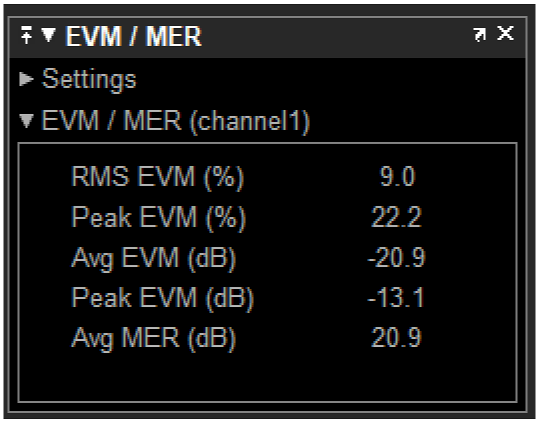
To communicate simulation data that corresponds to the current display, the scope uses the **Frames** indicator on the scope window. This figure highlights important aspects of the Constellation Diagram window.



To change the Constellation Diagram window settings, select menu options under **File**, **Tools**, and **View**. Changes made via menu options adjust the property settings of the System object accordingly.

EVM / MER Measurements

The **EVM / MER** signal quality pane displays the measurement settings, and error vector magnitude (EVM) and modulation error ratio (MER) measurement calculation results for the specified trace selection.



```
RMS EVM (%)      9.0
Peak EVM (%)     22.2
Avg EVM (dB)     -20.9
Peak EVM (dB)    -13.1
Avg MER (dB)     20.9
```

- **EVM** — An error vector is a vector in the IQ plane from the ideal constellation point to the actual point at the receiver. The root mean square error vector magnitude, EVM_{RMS}, is measured for the average and peak constellation power.

  On the constellation diagram, you can display the EVM_{RMS} measurements normalized by either the `Average constellation power` or `Peak constellation power` method as computed using these algorithms.

EVM Normalization Method	Algorithm
`Average constellation power`	Average constellation power normalization: $$\mathrm{EVM}_k = 100\sqrt{\frac{e_k}{P_{\mathrm{avg}}}}$$ $\mathrm{EVM}_{\mathrm{RMS}}$, in percent, for average constellation power normalization: $$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{avg}}}}$$
`Peak constellation power`	Peak constellation power normalization $$\mathrm{EVM}_k = 100\sqrt{\frac{e_k}{P_{\mathrm{max}}}}$$ EVM_{RMS}, in percent, for peak constellation power normalization $$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{max}}}}$$

  The **EVM / MER** pane shows the average and peak $\mathrm{EVM}_{\mathrm{RMS}}$ in both percent and decibels for the selected trace. The EVM reported in decibels is computed as EVM (dB) = 10-log10($\mathrm{EVM}_{\mathrm{MS}}$) = 20-log10($\mathrm{EVM}_{\mathrm{RMS}}$), where:

  - $e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
  - I_k is the in-phase value of the kth symbol in the input vector.
  - Q_k is the quadrature phase value of the kth symbol in the input vector.
  - I_k and Q_k represent ideal (reference) symbol values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbol values.
  - N is the input vector length.
  - P_{avg} is the value for `Average constellation power`.
  - P_{max} is the value for `Peak constellation power`.
  - $\text{EVM}_{\text{RMS}} = \sqrt{\text{EVM}_{\text{MS}}}$

  The maximum EVM value in a vector is $\text{EVM}_{\text{max}} = \max_{k \in [1,\ldots,N]} \{\text{EVM}_k\}$, where k is the kth symbol in a vector of length N.

  For more information, see `comm.EVM`.

- **MER** — MER is the ratio of the average power of the transmitted signal to the average power of the error vector. The **EVM / MER** pane indicates average MER measurement result in decibels for the selected trace.

  MER is a measure of the SNR in a modulated signal, calculated in dB. The MER over N symbols is

$$\text{MER} = 10 \cdot \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}(e_k)}\right)\text{dB},$$

  where:

  - $e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
  - I_k is the in-phase value of the kth symbol in the input vector.
  - Q_k is the quadrature phase value of the kth symbol in the input vector.
  - I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

  For more information, see `comm.MER`.

Tips

- If you want any of these features, use a `comm.ConstellationDiagram` System object.
  - Measurements
  - Basic reference constellations
  - Signal trajectory plots
  - Maintaining state between calls
- If you want a simple signal constellation snapshot, use the `scatterplot` function.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Blocks
Constellation Diagram

Functions
`eyediagram` | `scatterplot`

Introduced in R2013a

comm.ConvolutionalDeinterleaver

Package: comm

Restore ordering of symbols using shift registers

Description

The ConvolutionalDeinterleaver object recovers a signal that was interleaved using the convolutional Interleaver object. The parameters in the two blocks should have the same values.

To recover convolutionally interleaved binary data:

1. Define and set up your convolutional deinterleaver object. See "Construction" on page 3-332.
2. Call step to convolutionally deinterleave according to the properties of comm.ConvolutionalDeinterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.ConvolutionalDeinterleaver creates a convolutional deinterleaver System object, H. This object restores the original ordering of a sequence that was interleaved using the convolutional interleaver System object.

H = comm.ConvolutionalDeinterleaver(Name,Value) creates a convolutional deinterleaver System object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is 6.

RegisterLengthStep

Symbol capacity difference of each successive shift register

Specify the difference in symbol capacity of each successive shift register, where the last register holds zero symbols as a positive, scalar integer. The default is 2.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector, except the last shift register, which has zero delay. If you set this property to a scalar, then all shift registers, except the last one, store the same specified value. You can also set this property to a column vector with length equal to the value of the `NumRegisters` property. With this setting, the *i*-th shift register stores the *(N–i+1)*-th element of the specified vector. The value of the first element of this property is unimportant because the last shift register has zero delay. The default is `0`.

Methods

reset	Reset states of the convolutional deinterleaver object
step	Restore ordering of symbols using shift registers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans = 21×3

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
      ⋮
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
```

```
intrlvDelay = 6
```

```
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
numSymErrors = 0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Deinterleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalInterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: `comm.ConvolutionalDeinterleaver`
Package: `comm`

Reset states of the convolutional deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalDeinterleaver` object, `H`.

step

System object: `comm.ConvolutionalDeinterleaver`
Package: `comm`

Restore ordering of symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a convolutional interleaver and returns `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional deinterleaver object uses a set of *N* shift registers, where *N* is the value specified by the NumRegisters property. The object sets the delay value of the *k*-th shift register to the product of (*k*-1) and RegisterLengthStep property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the *N*-th register and the next new input occurs, it returns to the first register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.ConvolutionalEncoder

Package: comm

Convolutionally encode binary data

Description

The ConvolutionalEncoder object encodes a sequence of binary input vectors to produce a sequence of binary output vectors.

To convolutionally encode a binary signal:

1. Define and set up your convolutional encoder object. See "Construction" on page 3-337.
2. Call step to encode a sequence of binary input vectors to produce a sequence of binary output vectors according to the properties of comm.ConvolutionalEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.ConvolutionalEncoder creates a System object, H, that convolutionally encodes binary data.

H = comm.ConvolutionalEncoder(Name,Value) creates a convolutional encoder object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.ConvolutionalEncoder(TRELLIS,Name,Value) creates a convolutional encoder object, H This object has the TrellisStructure on page 3-0 property set to TRELLIS, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check if a structure is a valid trellis structure. The default is the result of poly2trellis(7, [171 133]).

TerminationMethod

Termination method of encoded frame

Specify how the encoded frame is terminated as one of Continuous | Truncated | Terminated. The default is Continuous. When you set this property to Continuous, the object retains the encoder states at the end of each input vector for use with the next input vector. When you set this property to `Truncated`, the object treats each input vector independently. The encoder states are reset at the start of each input vector. If you set the `InitialStateInputPort` on page 3-0 property to `false`, the object resets its states to the all-zeros state. If you set the `InitialStateInputPort` property to `true`, the object resets the states to the values you specify in the initial states `step` method input. When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder states to all-zeros states at the end of the vector. For a rate *K*/*N* code, the *step* method outputs a vector with length $N \times {}^{(L + S)}\!/\!{}_{K}$, where S = *constraintLength*–1 (or, in the case of multiple constraint lengths, S = **sum**(*constraintLength*(*i*)–1)). *L* is the length of the input to the `step` method.

`ResetInputPort`

Enable encoder reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

`DelayedResetAction`

Delay output reset

Set this property to `true` to delay resetting the object output. The default is `false`. When you set this property to `true`, the reset of the internal states of the encoder occurs after the object computes the encoded data. When you set this property to `false`, the reset of the internal states of the encoder occurs before the object computes the encoded data. This property applies when you set the `ResetInputPort` on page 3-0 property to `true`.

`InitialStateInputPort`

Enable initial state input

Set this property to `true` to enable a `step` method input that allows the specification of the initial state of the encoder for each input vector. The default is `false`. This property applies when you set the `TerminationMethod` on page 3-0 property to `Truncated`.

`FinalStateOutputPort`

Enable final state output

Set this property to true to obtain the final state of the encoder via a `step` method output. The default is `false`. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous` or `Truncated`.

`PuncturePatternSource`

Source of puncture pattern

Specify the source of the puncture pattern as one of `None | Property`. The default is `None`. When you set this property to `None` the object does not apply puncturing. When you set this property to `Property`, the object punctures the code. This puncturing is based on the puncture pattern vector that you specify in the `PuncturePattern` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous` or `Truncated`.

PuncturePattern

Puncture pattern vector

Specify the puncture pattern used to puncture the encoded data as a column vector. The default is `[1; 1; 0; 1; 0; 1]`. The vector contains `1`s and `0`s, where the `0` indicates the punctured, or excluded, bits. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous` or `Truncated` and the `PuncturePatternSource` on page 3-0 property to `Property`.

Methods

reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Encode and Decode 8-DPSK Modulated Data

Transmit a convolutionally encoded 8-DPSK modulated bit stream through an AWGN channel. Then, demodulate and decode using a Viterbi decoder.

Create the necessary System objects.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)',...
    'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.ViterbiDecoder('InputFormat','Hard');
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay', 34);
```

Process the data using the following steps:

1. Generate random bits
2. Convolutionally encode the data
3. Apply DPSK modulation
4. Pass the modulated signal through AWGN
5. Demodulate the noisy signal
6. Decode the data using a Viterbi algorithm
7. Collect error statistics

```
for counter = 1:20
    data = randi([0 1],30,1);
    encodedData = step(hConEnc, data);
    modSignal = step(hMod, encodedData);
    receivedSignal = step(hChan, modSignal);
    demodSignal = step(hDemod, receivedSignal);
    receivedBits = step(hDec, demodSignal);
    errors = step(hError, data, receivedBits);
end
```

Display the number of errors.

```
errors(2)
```

```
ans = 3
```

Convolutional Encoding and Viterbi Decoding with a Puncture Pattern Matrix

Encode and decode a sequence of bits using a convolutional encoder and a Viterbi decoder with a defined puncture pattern. Verify that the input and output bits are identical

Define a puncture pattern matrix and reshape it into vector form for use with the Encoder and Decoder objects.

```
pPatternMat = [1 0 1;1 1 0];
pPatternVec = reshape(pPatternMat,6,1);
```

Create convolutional encoder and a Viterbi decoder in which the puncture pattern is defined by `pPatternVec`.

```
ENC = comm.ConvolutionalEncoder(...
    'PuncturePatternSource','Property', ...
    'PuncturePattern',pPatternVec);

DEC = comm.ViterbiDecoder('InputFormat','Hard', ...
    'PuncturePatternSource','Property',...
    'PuncturePattern',pPatternVec);
```

Create an error rate counter with the appropriate receive delay.

```
ERR = comm.ErrorRate('ReceiveDelay',DEC.TracebackDepth);
```

Encode and decode a sequence of random bits.

```
dataIn = randi([0 1],600,1);

dataEncoded = step(ENC,dataIn);

dataOut = step(DEC,dataEncoded);
```

Verify that there are no errors in the output data.

```
errStats = step(ERR,dataIn,dataOut);
errStats(2)
```

```
ans = 0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Encoder block reference page. The object properties correspond to the block parameters, except:
The operation mode **Reset on nonzero input via port** block parameter corresponds to the `ResetInputPort` on page 3-0 property.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.APPDecoder` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: `comm.ConvolutionalEncoder`
Package: `comm`

Reset states of the convolutional encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalEncoder` object, `H`.

step

System object: `comm.ConvolutionalEncoder`
Package: `comm`

Convolutionally encode binary data

Syntax

```
Y = step(H,X)
Y = step(H,X,INITSTATE)
Y = step(H,X,R)
[Y,FSTATE] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` encodes the binary data, `X`, using the convolutional encoding that you specify in the TrellisStructure property. It returns the encoded data, `Y`. Both `X` and `Y` are column vectors of data type numeric, logical, or unsigned fixed point of word length `1` (fi object). When the convolutional encoder represents a rate *K*/*N* code, the length of the input vector equals *K*×*L*, for some positive integer, *L*. The step method sets the length of the output vector, `Y`, to *L*×*N*.

`Y = step(H,X,INITSTATE)` uses the initial state specified in the `INITSTATE` input when you set the `TerminationMethod` property to 'Truncated' and the InitialStateInputPort property to `true`. `INITSTATE` must be an integer scalar.

`Y = step(H,X,R)` resets the internal states of the encoder when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar. This syntax applies when you set the TerminationMethod property to `Continuous` and the ResetInputPort property to true.

`[Y,FSTATE] = step(H,X)` returns the final state of the encoder in the integer scalar output `FSTATE` when you set the `FinalStateOutputPort` property to `true`. This syntax applies when you set the `TerminationMethod` property to `Continuous` or `Truncated`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.ConvolutionalInterleaver

Package: comm

Permute input symbols using shift registers with same property values

Description

The ConvolutionalInterleaver object permutes the symbols in the input signal. Internally, this class uses a set of shift registers.

To convolutionally interleave binary data:

1. Define and set up your convolutional interleaver object. See “Construction” on page 3-344.
2. Call step to convolutionally interleave according to the properties of comm.ConvolutionalInterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.ConvolutionalInterleaver creates a convolutional interleaver System object, H, that permutes the symbols in the input signal using a set of shift registers.

H = comm.ConvolutionalInterleaver(Name,Value) creates a convolutional interleaver System object, H. This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is 6.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is 2. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector. You do not need to specify a value for the first shift register, which has zero delay. The default is `0`. The value of the first element of this property is unimportant because the first shift register has zero delay. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the `NumRegisters` on page 3-0 property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans = 21×3

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
      ⋮
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
```

```
intrlvDelay = 6
```

```
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
numSymErrors = 0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Interleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: `comm.ConvolutionalInterleaver`
Package: `comm`

Reset states of the convolutional interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ConvolutionalInterleaver` object, `H`.

step

System object: `comm.ConvolutionalInterleaver`
Package: `comm`

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional interleaver object uses a set of *N* shift registers, where *N* is the value specified by the NumRegisters property. The object sets the delay value of the *k*-th shift register to the product of (*k*-1) and the RegisterLengthStep property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the *N*-th register and the next new input occurs , it returns to the first register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CPFSKDemodulator

Package: `comm`

Demodulate using CPFSK method and Viterbi algorithm

Description

The `CPFSKDemodulator` object demodulates a signal that was modulated using the continuous phase frequency shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using the continuous phase frequency shift keying method:

1. Define and set up your CPFSK demodulator object. See "Construction" on page 3-349 .
2. Call `step` to demodulate the signal according to the properties of `comm.CPFSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.CPFSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input continuous phase frequency shift keying (CPFSK) modulated data using the Viterbi algorithm.

`H = comm.CPFSKDemodulator(Name,Value)` creates a CPFSK demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.CPFSKDemodulator(M,Name,Value)` creates a CPFSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector of length equal to *N*/`SamplesPerSymbol` on page 3-0 and with elements that are integers between –(`ModulationOrder` on page 3-0 –1) and `ModulationOrder`–1. In this case, *N*, is the length of the input signal, which indicates the number of input baseband modulated symbols.

When you set this property to `true`, the `step` method outputs a binary column vector of length equal to *P*×(*N*/`SamplesPerSymbol`), where *P* = **log2**(`ModulationOrder`). The output contains length-*P* bit words. In this scenario, the object first maps each demodulated symbol to an odd integer value, *K*, between –(`ModulationOrder`-1) and `ModulationOrder`–1. The object then maps *K* to the nonnegative integer (*K*+`ModulationOrder`–1)/2. Finally, the object maps each nonnegative integer to a length-*P* binary word, using the mapping specified in the `SymbolMapping` on page 3-0 property.

SymbolMapping

Symbol encoding

Specify the mapping of the modulated symbols as one of `Binary` | `Gray`. The default is `Binary`. This property determines how the object maps each demodulated integer symbol value (in the range 0 and `ModulationOrder` on page 3-0 –1) to a *P*-length bit word, where *P* = `ModulationOrder` on page 3-0 (`ModulationOrder`).

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitOutput` on page 3-0 property to `true`.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, *h*, or a column vector, [h_0, h_1, ... $h_{\text{H-1}}$]

where H-1 represents the length of the column vector.

When h_{i} varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_{i} must be a rational number.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar. The default is 16. The value of this property is also the value of the output delay. That value is the number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of `int8` | `int16` | `int32` | `double`, when you set the `BitOutput` on page 3-0 property to false. The default is `double`.

When you set the `BitOutput` property to `true`, specify the output data type as one of `logical` | `double`.

Methods

reset	Reset states of CPFSK demodulator object
step	Demodulate using CPFSK method and Viterbi algorithm

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Demodulate a signal using CPFSK modulation with Gray mapping

```
% Create a CPFSK modulator, an AWGN channel, and a CPFSK demodulator
    hMod = comm.CPFSKModulator(8, 'BitInput', true, ...
                         'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                         'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPFSKDemodulator(8, 'BitOutput', true, ...
                         'SymbolMapping', 'Gray');

 % Create an error rate calculator, account for the delay caused by the Viterbi algorithm.
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end

    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.004006
Number of errors = 120
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPFSK Demodulator Baseband block reference page. The object properties correspond to the block parameters. For CPFSK the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.CPMDemodulator` | `comm.CPMModulator`

Introduced in R2012a

reset

System object: `comm.CPFSKDemodulator`
Package: `comm`

Reset states of CPFSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPFSKDemodulator` object, `H`.

step

System object: `comm.CPFSKDemodulator`
Package: `comm`

Demodulate using CPFSK method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the CPFSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision, column vector with a length equal to an integer multiple of the number of samples per symbol specified in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CPFSKModulator

Package: `comm`

Modulate using CPFSK method

Description

The `CPFSKModulator` object modulates using the continuous phase frequency shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using the continuous phase frequency shift keying method:

1. Define and set up your CPFSK modulator object. See "Construction" on page 3-355.
2. Call `step` to modulate the signal according to the properties of `comm.CPFSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.CPFSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the continuous phase frequency shift keying (CPFSK) modulation method.

`H = comm.CPFSKModulator(Name,Value)` creates a CPFSK modulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.CPFSKModulator(M,Name,Value)` creates a CPFSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

`BitInput`

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `false`, the `step` method input must be a double-precision or signed integer data type column vector.

This vector comprises odd integer values between –(ModulationOrder on page 3-0 –1) and ModulationOrder–1.

When you set this property to true, the step method input must be a column vector of *P*-length bit words, where *P* = **log2**(ModulationOrder). The input data must be double precision or logical data type. The object maps each bit word to an integer K between 0 and ModulationOrder–1, using the mapping specified in the SymbolMapping on page 3-0 property. The object then maps the integer *K* to the intermediate value 2*K*–(ModulationOrder–1) and proceeds as in the case when you set the BitInput on page 3-0 property to false.

SymbolMapping

Symbol encoding

Specify the mapping of bit inputs as one of Binary | Gray. The default is Binary. This property determines how the object maps each input *P*-length bit word, where *P* = **log2**(ModulationOrder on page 3-0), to an integer between 0 and ModulationOrder–1.

When you set this property to Binary, the object uses a natural binary-coded ordering.

When you set this property to Gray, the object uses a Gray-coded ordering.

This property applies when you set the BitInput on page 3-0 property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is 0.5. The value of this property can be a scalar, *h*, or a column vector, [h_0, h_1, ... $h_{\text{H-1}}$]

where H-1 represents the length of the column vector. The phase shift over a symbol is π × *h*.

When h_{i} varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_{i} must be a rational number.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar. The default is 0.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar. The default is 8. The upsampling factor is the number of output samples that the step method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of double | single. The default is double.

Methods

reset	Reset states of CPFSK modulator object
step	Modulate using CPFSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate a signal using CPFSK modulation with Gray mapping

```
% Create a CPFSK modulator, an AWGN channel, and a CPFSK demodulator
    hMod = comm.CPFSKModulator(8, 'BitInput', true, ...
                         'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                         'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPFSKDemodulator(8, 'BitOutput', true, ...
                         'SymbolMapping', 'Gray');

 % Create an error rate calculator, account for the delay caused by the Viterbi algorithm.
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end

    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.004006
Number of errors = 120
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPFSK Modulator Baseband block reference page. The object properties correspond to the block parameters. For CPFSK the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPMDemodulator` | `comm.CPMModulator`

Introduced in R2012a

reset

System object: `comm.CPFSKModulator`
Package: `comm`

Reset states of CPFSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPFSKModulator` object, `H`.

step

System object: `comm.CPFSKModulator`
Package: `comm`

Modulate using CPFSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the CPFSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with data types double, signed integer, or logical. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol specified in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CPMCarrierPhaseSynchronizer

Package: `comm`

(Removed) Recover carrier phase of baseband CPM signal

Note `comm.CPMCarrierPhaseSynchronizer` has been removed. Use `comm.CarrierSynchronizer` instead.

Description

The `CPMCarrierPhaseSynchronizer` object recovers the carrier phase of the input signal using the 2*P*-Power method. This feedforward method is clock aided, but not data aided. The method is suitable for systems that use certain types of baseband modulation. These types include: continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), and Gaussian minimum shift keying (GMSK).

To recover the carrier phase of the input signal:

1. Define and set up your CPM carrier phase synchronizer object. See “Construction” on page 3-361.
2. Call `step` to recover the carrier phase of the input signal using the 2*P*-Power method according to the properties of `comm.CPMCarrierPhaseSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.CPMCarrierPhaseSynchronizer` creates a CPM carrier phase synchronizer System object, `H`. This object recovers the carrier phase of a baseband continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), or Gaussian minimum shift keying (GMSK) modulated signal using the 2P-power method.

`H = comm.CPMCarrierPhaseSynchronizer(Name,Value)` creates a CPM carrier phase synchronizer object, `H` This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.CPMCarrierPhaseSynchronizer(HALFPOW,Name,Value)` creates a CPM carrier phase synchronizer object, `H`. This object has the `P` on page 3-0 property set to `HALFPOW`, and the other specified properties set to the specified values.

Properties

P

Denominator of CPM modulation index

Specify the denominator of the CPM modulation index of the input signal as a real positive scalar integer value of data type single or double. The default is `2`. This property is tunable.

ObservationInterval

Number of symbols where carrier phase assumed constant

Specify the observation interval as a real positive scalar integer value of data type single or double. The default is `100`.

Methods

reset	Reset states of the CPM carrier phase synchronizer object
step	(Removed) Recover carrier phase of baseband CPM signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover carrier phase of a GMSK signal.

```
% Initialize random seed for repeatability
rng(123)

% Create a GMSK modulator, an AWGN channel, and a GMSK demodulator.
% Use a phase offset of pi/4.
samplesPerSymbol = 4;
modulator = comm.GMSKModulator('BitInput',true,'InitialPhaseOffset',pi/4, ...
    'SamplesPerSymbol',samplesPerSymbol);
awgnChannel = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)', ...
    'SNR',40);
demodulator = comm.GMSKDemodulator('BitOutput',true, ...
    'InitialPhaseOffset',pi/4,'SamplesPerSymbol',samplesPerSymbol);

% Create a System object for frequency offset
frequencyOffset = 1e4;
sampleRate = 1e6;
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',frequencyOffset, ...
    'SampleRate',sampleRate);

% Create a carrier synchronizer
carrierSync = comm.CarrierSynchronizer('SamplesPerSymbol', samplesPerSymbol, ...
    'Modulation','OQPSK');

% Create a preamble for resolving phase ambiguity
preambleLength = 13;
barker = comm.BarkerCode('Length',preambleLength,'SamplesPerFrame', ...
    preambleLength); % For preamble
preamble = (1+barker())/2; % Length 13, unipolar

% Create an error rate calculator, account for the delay caused by
% the Viterbi algorithm used by the GMSK demodulator.
errorCalculator = comm.ErrorRate('ReceiveDelay',demodulator.TracebackDepth);

numFrames = 4000;
symbolsPerFrame = 300;
estimatedPhaseOffset = zeros(symbolsPerFrame*samplesPerSymbol,numFrames);
```

```
errorHistory = zeros(numFrames,1);

for counter = 1:numFrames
    % Transmit 300 bits
    payload = randi([0 1],symbolsPerFrame - preambleLength,1);
    data = [preamble; payload];
    modulatedSignal = modulator(data);
    noisySignal = awgnChannel(modulatedSignal);
    rxSignal = pfo(noisySignal); % with carrier frequency offset

    [syncSignal, phOffset] = carrierSync(rxSignal);
    estimatedPhaseOffset(:, counter) = phOffset;

    phaseCorrection = round((2/pi)*angle(...
        modulatedSignal(1:preambleLength*samplesPerSymbol).* ...
        conj(syncSignal(1:preambleLength*samplesPerSymbol))));
    phaseCorrection(phaseCorrection==-2)=2;
    phaseCorrection = mode(phaseCorrection)*pi/2;

    receivedData = demodulator(exp(1j*phaseCorrection)*syncSignal);
    errorStats = errorCalculator(data,receivedData);
    errorHistory(counter) = errorStats(2);
end

estimatedFreqOffset = estimatedPhaseOffset(:);
% plot((estimatedFreqOffset(2:end)-estimatedFreqOffset(1:end-1))*sampleRate/(2*pi))
% xlabel('Sample')
% ylabel('Estimated Frequency Offset (Hz)');

% Number of bit errors (ignoring those in the first half of the simulation,
% before frequency and phase offset are corrected)
numErrors = errorHistory(end) - errorHistory(0.5*end)
```

```
numErrors =

     0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Phase Recovery block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.CPMCarrierPhaseSynchronizer` has been removed

Errors starting in R2020a

`comm.CPMCarrierPhaseSynchronizer` has been removed. Use `comm.CarrierSynchronizer` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMModulator` | `comm.CarrierSynchronizer`

Introduced in R2012a

reset

System object: `comm.CPMCarrierPhaseSynchronizer`
Package: `comm`

Reset states of the CPM carrier phase synchronizer object

Note `comm.CPMCarrierPhaseSynchronizer` has been removed. Use `comm.CarrierSynchronizer` instead.

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMCarrierPhaseSynchronizer` object, `H`.

step

System object: comm.CPMCarrierPhaseSynchronizer
Package: comm

(Removed) Recover carrier phase of baseband CPM signal

Note comm.CPMCarrierPhaseSynchronizer has been removed. Use comm.CarrierSynchronizer instead.

Syntax

```
[Y,PH] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

[Y,PH] = step(H,X) recovers the carrier phase of the input signal, X, and returns the phase corrected signal, Y, and the carrier phase estimate (in degrees), PH. X must be a complex scalar or column vector input signal of data type single or double.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.CPMDemodulator

Package: `comm`

Demodulate using CPM method and Viterbi algorithm

Description

The `CPMDemodulator` object demodulates a signal that was modulated using continuous phase modulation. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using continuous phase modulation:

1. Define and set up your CPM demodulator object. See “Construction” on page 3-367.
2. Call `step` to demodulate a signal according to the properties of `comm.CPMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.CPMDemodulator` creates a demodulator System object, `H`. This object demodulates the input continuous phase modulated (CPM) data using the Viterbi algorithm.

`H = comm.CPMDemodulator(Name,Value)` creates a CPM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.CPMDemodulator(M,Name,Value)` creates a CPM demodulator object, `H`, with the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property requires a power of two, real, integer scalar. The default is `4`.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector of length equal to *N*/ `SamplesPerSymbol` on page 3-0 and with elements that are integers between -

(`ModulationOrder` on page 3-0 -1) and `ModulationOrder`–1. Here, *N*, is the length of the input signal which indicates the number of input baseband modulated symbols.

When you set this property to `true`, the `step` method outputs a binary column vector of length equal to *P*×(*N*/`SamplesPerSymbol`), where *P* = **log2**(`ModulationOrder`). The output contains length-*P* bit words. In this scenario, the object first maps each demodulated symbol to an odd integer value, *K*, between –(`ModulationOrder`–1) and `ModulationOrder`–1. The object then maps *K* to the nonnegative integer (*K*+`ModulationOrder`–1)/2. Finally, the object maps each nonnegative integer to a length-*P* binary word, using the mapping specified in the `SymbolMapping` on page 3-0 property.

SymbolMapping

Symbol encoding

Specify the mapping of the demodulated symbols as one of `Binary` | `Gray`. The default is `Binary`. This property determines how the object maps each demodulated integer symbol value (in the range `0` and `ModulationOrder` on page 3-0 –1) to a *P*-length bit word, where *P* = **log2**(`ModulationOrder`).

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitOutput` on page 3-0 property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, *h*, or a column vector, [h_0, h_1, …. $h_{\text{H-1}}$]

where H-1 represents the length of the column vector.

When h_i varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_i must be a rational number.

FrequencyPulse

Frequency pulse shape

Specify the type of pulse shaping that the modulator has used to smooth the phase transitions of the input modulated signal as one of `Rectangular` | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`. The default is `Rectangular`.

MainLobeDuration

Main lobe duration of spectral raised cosine pulse

Specify, in number of symbol intervals, the duration of the largest lobe of the spectral raised cosine pulse. This value is the value that the modulator used to pulse-shape the input modulated signal. The default is `1`. This property requires a real, positive, integer scalar. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Spectral Raised Cosine`.

RolloffFactor

Rolloff factor of spectral raised cosine pulse

Specify the roll off factor of the spectral raised cosine pulse. This value is the value that the modulator used to pulse-shape the input modulated signal. The default is `0.2`. This property requires a real scalar between `0` and `1`. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Spectral Raised Cosine`.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape. This value is the value that the modulator used to pulse-shape the input modulated signal. The default is `0.3`. This property requires a real, positive scalar. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Gaussian`.

PulseLength

Pulse length

Specify the length of the frequency pulse shape in symbol intervals. The value of this property requires a real positive integer. The default is `1`.

SymbolPrehistory

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the `step` method. The default is `1`. This property requires a scalar or vector with odd integer elements between –(`ModulationOrder` on page 3-0 –1) and (`ModulationOrder`–1). If the value is a vector, then its length must be one less than the value in the `PulseLength` on page 3-0 property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar. The default is `0`.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar. The default is `8`.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar. The default is `16`. The value of this property is also the output delay, which is the number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of `int8` | `int16` | `int32` | `double`, when you set the `BitOutput` on page 3-0 property to false. When you set the `BitOutput` property to `true`, specify the output data type as one of `logical` | `double`. The default is `double`.

Methods

reset	Reset states of CPM demodulator object
step	Demodulate using CPM method and Viterbi algorithm

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Demodulate a CPM signal with Gray mapping and bit inputs

```
% Create a CPM modulator, an AWGN channel, and a CPM demodulator.
    hMod = comm.CPMModulator(8, 'BitInput', true, ...
                         'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                         'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPMDemodulator(8, 'BitOutput', true, ...
                         'SymbolMapping', 'Gray');
% Create an error rate calculator, account for the delay caused by the Viterbi algorithm.
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.004006
Number of errors = 120
```

Apply GFSK Modulation and Demodulation

Using the `comm.CPMModulator` and `comm.CPMDemodulator` System objects apply GFSK modulation and demodulation to random bit data.

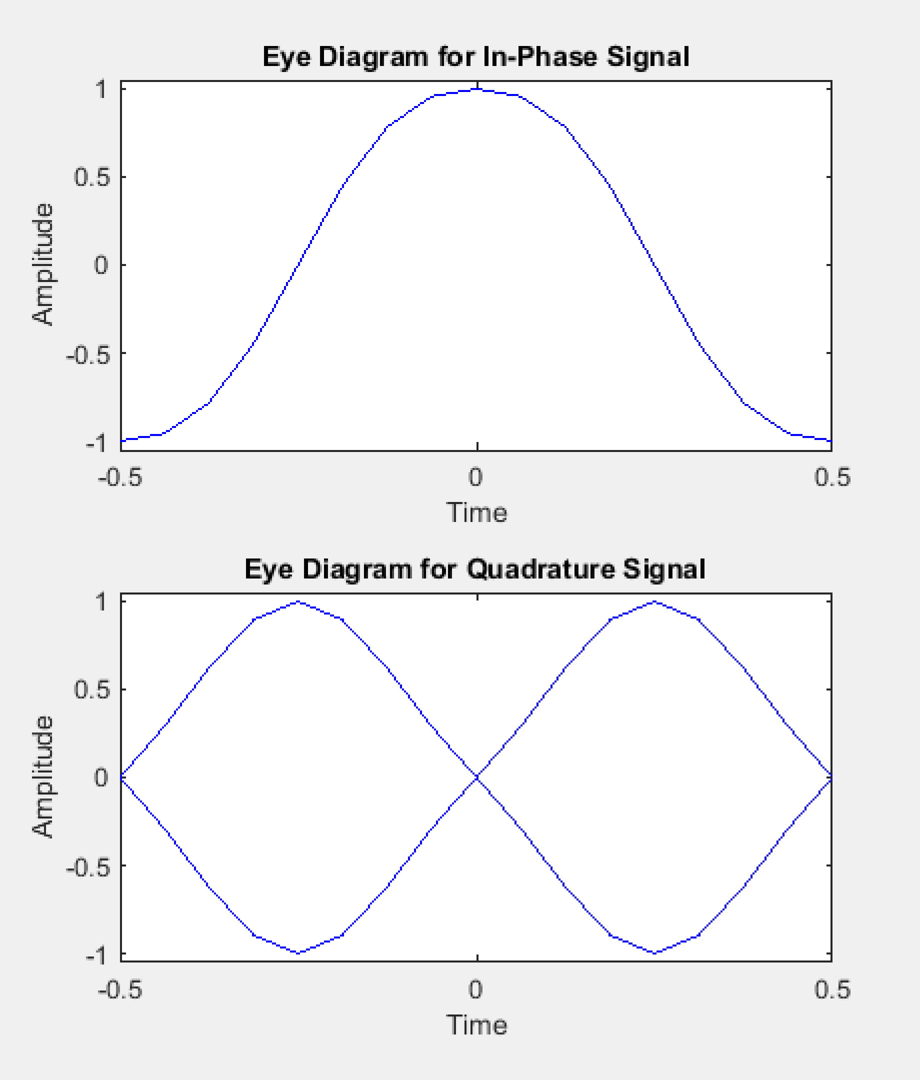
Create a GFSK modulator and demodulator object pair.

```
gfskMod = comm.CPMModulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
```

```
                  'BitInput', true);
gfskDemod = comm.CPMDemodulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
                  'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
                  'BitOutput', true);
```

Generate random bit data and apply GFSK modulation. Use a scatter plot to view the constellation.

```
numSym = 100;
x = randi([0 1],numSym*gfskMod.SamplesPerSymbol,1);
y = gfskMod(x);
eyediagram(y,16)
```



Demodulate the GFSK modulated data. To verify that the demodulated signal data is equal to the original data, you must account for the delay introduced by the Gaussian filtering in the GFSK modulation and demodulation processing.

```
z = gfskDemod(y);
delay = finddelay(x,z);
isequal(x(1:end-delay),z(delay+1:end))
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Demodulator Baseband block reference page. The object properties correspond to the block parameters. For CPM the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPMModulator` | `comm.GMSKDemodulator` | `comm.MSKDemodulator`

Introduced in R2012a

reset

System object: `comm.CPMDemodulator`
Package: `comm`

Reset states of CPM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMDemodulator` object, `H`.

step

System object: `comm.CPMDemodulator`
Package: `comm`

Demodulate using CPM method and Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the CPM demodulator System object, `H`, and returns `Y`. `X` must be a double or single precision, column vector with a length equal to an integer multiple of the number of samples per symbol specified in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CPMModulator

Package: `comm`

Modulate using CPM method

Description

The `CPMModulator` object modulates using continuous phase modulation. The output is a baseband representation of the modulated signal.

To modulate a signal using continuous phase modulation:

1. Define and set up your CPM modulator object. See “Construction” on page 3-375.
2. Call `step` to modulate a signal according to the properties of `comm.CPMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.CPMModulator` creates a modulator System object, `H`. This object modulates the input signal using the continuous phase modulation (CPM) method.

`H = comm.CPMModulator(Name,Value)` creates a CPM modulator object, `H`. This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.CPMModulator(M,Name,Value)` creates a CPM modulator object, `H`, with the `ModulationOrder` property set to `M` and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Size of symbol alphabet

Specify the size of the symbol alphabet. The value of this property must be a power of two, real, integer scalar. The default is `4`.

`BitInput`

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input requires double-precision or signed integer data type column vector. This vector must comprise odd integer values between –(`ModulationOrder` on page 3-0 –1) and `ModulationOrder`–1.

When you set this property to `true`, the `step` method input requires a column vector of *P*-length bit words, where *P* = **log2**(`ModulationOrder`). The input data must have a double-precision or logical data type. The object maps each bit word to an integer *K* between `0` and `ModulationOrder`–1, using the mapping specified in the `SymbolMapping` on page 3-0 property. The object then maps the integer *K* to the intermediate value 2*K*-(`ModulationOrder`–1) and proceeds as in the case when BitInput is false.

SymbolMapping

Symbol encoding

Specify the mapping of bit inputs as one of `Binary` | `Gray`. The default is `Binary`. This property determines how the object maps each input *P*-length bit word, where *P* = **log2**(`ModulationOrder` on page 3-0), to an integer between 0 and `ModulationOrder`–1.

When you set this property to `Binary`, the object uses a natural binary-coded ordering.

When you set this property to `Gray`, the object uses a Gray-coded ordering.

This property applies when you set the `BitInput` on page 3-0 property to true.

ModulationIndex

Modulation index

Specify the modulation index. The default is `0.5`. The value of this property can be a scalar, h, or a column vector, [h_0, h_1, …. $h_{\mathrm{H-1}}$]

where H-1 represents the length of the column vector. The phase shift over a symbol is π × h.

When h_i varies from interval to interval, the object operates in multi-h. When the object operates in multi-h, h_i must be a rational number.

FrequencyPulse

Frequency pulse shape

Specify the type of pulse shaping that the modulator uses to smooth the phase transitions of the modulated signal. Choose from `Rectangular` | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`. The default is `Rectangular`.

MainLobeDuration

Main lobe duration of spectral raised cosine pulse

Specify, in number of symbol intervals, the duration of the largest lobe of the spectral raised cosine pulse. The default is `1`. This property requires a real, positive, integer scalar. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Spectral Raised Cosine`.

RolloffFactor

Rolloff factor of spectral raised cosine pulse

Specify the rolloff factor of the spectral raised cosine pulse. The default is `0.2`. This property requires a real scalar between `0` and `1`. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Spectral Raised Cosine`.

`BandwidthTimeProduct`

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape. The default is `0.3`. This property requires a real, positive scalar. This property applies when you set the `FrequencyPulse` on page 3-0 property to `Gaussian`.

`PulseLength`

Pulse length

Specify the length of the frequency pulse shape in symbol intervals. The value of this property requires a real, positive integer. The default is `1`.

`SymbolPrehistory`

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the `step` method in reverse chronological order. The default is `1`. This property requires a scalar or vector with odd integer elements between –(`ModulationOrder` on page 3-0 –1) and (`ModulationOrder`–1). If the value is a vector, then its length must be one less than the value in the `PulseLength` on page 3-0 property.

`InitialPhaseOffset`

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar. The default is `0`.

`SamplesPerSymbol`

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar. The default is `8`. The upsampling factor is the number of output samples that the `step` method produces for each input sample.

`OutputDataType`

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of CPM modulator object
step	Modulate using CPM method

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate a CPM signal with Gray mapping and bit inputs

```
% Create a CPM modulator, an AWGN channel, and a CPM demodulator.
    hMod = comm.CPMModulator(8, 'BitInput', true, ...
                            'SymbolMapping', 'Gray');
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                            'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.CPMDemodulator(8, 'BitOutput', true, ...
                            'SymbolMapping', 'Gray');
% Create an error rate calculator, account for the delay caused by the Viterbi algorithm.
    delay = log2(hDemod.ModulationOrder)*hDemod.TracebackDepth;
    hError = comm.ErrorRate('ReceiveDelay', delay);
    for counter = 1:100
        % Transmit 100 3-bit words
        data = randi([0 1],300,1);
        modSignal = step(hMod, data);
        noisySignal = step(hAWGN, modSignal);
        receivedData = step(hDemod, noisySignal);
        errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.004006
Number of errors = 120
```

Apply GFSK Modulation and Demodulation

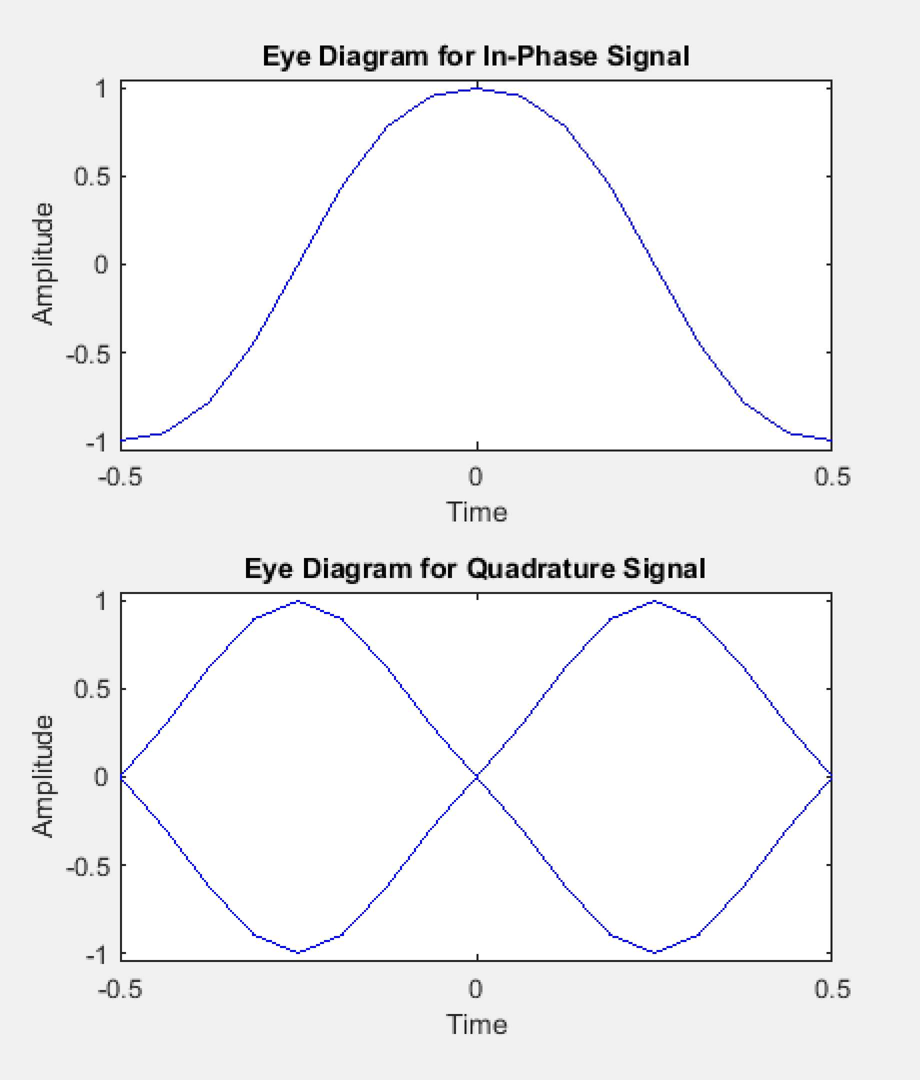
Using the comm.CPMModulator and comm.CPMDemodulator System objects apply GFSK modulation and demodulation to random bit data.

Create a GFSK modulator and demodulator object pair.

```
gfskMod = comm.CPMModulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitInput', true);
gfskDemod = comm.CPMDemodulator('ModulationOrder', 2, 'FrequencyPulse', 'Gaussian', ...
               'BandwidthTimeProduct', 0.5, 'ModulationIndex', 1, ...
               'BitOutput', true);
```

Generate random bit data and apply GFSK modulation. Use a scatter plot to view the constellation.

```
numSym = 100;
x = randi([0 1],numSym*gfskMod.SamplesPerSymbol,1);
y = gfskMod(x);
eyediagram(y,16)
```



Demodulate the GFSK modulated data. To verify that the demodulated signal data is equal to the original data, you must account for the delay introduced by the Gaussian filtering in the GFSK modulation and demodulation processing.

```
z = gfskDemod(y);
delay = finddelay(x,z);
isequal(x(1:end-delay),z(delay+1:end))
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the CPM Modulator Baseband block reference page. The object properties correspond to the block parameters. For CPM the phase shift per symbol is π × *h*, where *h* is the modulation index.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.CPMDemodulator` | `comm.GMSKModulator` | `comm.MSKModulator`

Introduced in R2012a

reset

System object: `comm.CPMModulator`
Package: `comm`

Reset states of CPM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `CPMModulator` object, `H`.

step

System object: `comm.CPMModulator`
Package: `comm`

Modulate using CPM method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the CPM modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with data types double, signed integer, or logical. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol specified in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.CRCDetector

Package: comm

Detect errors in input data using CRC

Description

The comm.CRCDetector System object computes cyclic redundancy check (CRC) checksums for an entire received codeword. For successful CRC detection in a communications system link, you must align the property settings of the comm.CRCDetector System object with the paired comm.CRCGenerator System object. For more information, see “CRC Syndrome Detector Operation” on page 3-388.

To detect errors in the received codeword containing CRC sequence bits:

1. Create the comm.CRCDetector object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
crcdetector = comm.CRCDetector
crcdetector = comm.CRCDetector(Name,Value)
crcdetector = comm.CRCDetector(poly,Name,Value)
```

Description

`crcdetector = comm.CRCDetector` creates a CRC code detector System object. This object detects errors in the received codewords according to a specified generator polynomial.

`crcdetector = comm.CRCDetector(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.CRCDetector('Polynomial','z^16 + z^14 + z + 1')` configures the CRC code detector System object to use the CRC-16 cyclic redundancy check bits when checking for CRC code errors in the received codewords. Enclose each property name in quotes.

`crcdetector = comm.CRCDetector(poly,Name,Value)` creates a CRC code detector System object. This object has the `Polynomial` property set to `poly`, and the other specified properties set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Polynomial` — Generator polynomial
`'z^16 + z^12 + z^5 + 1'` (default) | polynomial character vector | binary row vector | integer row vector

Generator polynomial for the CRC algorithm, specified as one of the following:

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial x^3+ z^2+ 1.
- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, `[3 2 0]` represents the polynomial $z^3 + z^2 + 1$.

For more information, see “Character Representation of Polynomials”.

Some commonly used generator polynomials include:

CRC method	Generator polynomial
CRC-32	`'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'`
CRC-24	`'z^24 + z^23 + z^14 + z^12 + z^8 + 1'`
CRC-16	`'z^16 + z^15 + z^2 + 1'`
Reversed CRC-16	`'z^16 + z^14 + z + 1'`
CRC-8	`'z^8 + z^7 + z^6 + z^4 + z^2 + 1'`
CRC-4	`'z^4 + z^3 + z^2 + z + 1'`

Example: `'z^7 + z^2 + 1'`, `[1 0 0 0 0 1 0 1]`, and `[7 2 0]` represent the same polynomial, $p(z) = z^7 + z^2 + 1$.

Data Types: `double` | `char`

`InitialConditions` — Initial states of internal shift register
`0` (default) | `1` | binary row vector

Initial states of the internal shift register, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial.

Data Types: `logical`

`DirectMethod` — Use direct algorithm for CRC checksum calculations
`false` (default) | `true`

Use direct algorithm for CRC checksum calculations, specified as `false` or `true`.

When you set this property to `true`, the object uses the direct algorithm for CRC checksum calculations. When you set this property to `false`, the object uses the non-direct algorithm for CRC checksum calculations.

For more information on direct and non-direct algorithms, see “Error Detection and Correction”.

Data Types: `logical`

`ReflectInputBytes` — Reflect input bytes

`false` (default) | `true`

Reflect input bytes, specified as `false` or `true`. Set this property to `true` to flip the received codeword on a bytewise basis before entering the data into the shift register.

When you set this property to `true`, the received codeword length divided by the value of the `ChecksumsPerFrame` property must be an integer and a multiple of `8`.

Data Types: `logical`

`ReflectChecksums` — Reflect checksums before final XOR

`false` (default) | `true`

Reflect checksums before final XOR, specified as `false` or `true`. Set this property to `true` to flip the CRC checksums around their centers after the received codeword is completely through the shift register.

When you set this property to `true`, the object flips the CRC checksums around their centers before the final XOR.

Data Types: `logical`

`FinalXOR` — Final XOR

0 (default) | binary scalar | binary vector

Final XOR, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. The XOR operation runs using the value of the `FinalXOR` property and the CRC checksum before comparing with the input checksum. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial. A setting of `0` is equivalent to no XOR operation.

Data Types: `logical`

`ChecksumsPerFrame` — Number of checksums calculated

1 (default) | positive integer

Number of checksums calculated for each received codeword frame, specified as a positive integer. for more information, see “CRC Syndrome Detector Operation” on page 3-388.

Data Types: `double`

Usage

Syntax

```
out = crcdetector(codeword)
[msg,err] = crcdetector(codeword)
```

Description

`out = crcdetector(codeword)` checks CRC code bits for each received codeword frame, removes the checksums, and then concatenates subframes to the output frame.

`[msg,err] = crcdetector(codeword)` also returns the checksum error signal computed when checking CRC code bits for each codeword subframe.

Input Arguments

`codeword` — Received codeword
binary column vector

Received codeword, specified as a binary column vector.

Data Types: `double` | `logical`

Output Arguments

`out` — Output frame
binary column vector

Output frame, returned as a binary column vector that inherits the data type of the input signal. The message word output contains the received codeword with the checksums removed.

The length of the output frame is *n* - *k* * *r* bits, where *n* is the size of the received codeword, *k* is the number of checksums per frame, and *r* is the degree of the generator polynomial.

`err` — Checksum error signal
binary column vector

Checksum error signal, returned as a binary column vector that inherits the data type of the input signal. The length of `Err` equals the value of `ChecksumsPerFrame`. For each checksum computation, an element value of 0 in `err` indicates no checksum error, and an element value of 1 in `err` indicates a checksum error.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

CRC Detection of Errors in a Random Message

Pass binary data through a CRC generator, introduce a bit error, and detect the error using a CRC detector.

Create a random binary vector.

```
x = randi([0 1],12,1);
```

Encode the input message frame using a CRC generator with the `ChecksumsPerFrame` property set to 2. This subdivides the incoming frame into two equal-length subframes.

```
crcgenerator = comm.CRCGenerator([1 0 0 1],'ChecksumsPerFrame',2);
codeword = crcgenerator(x);
```

Decode the codeword and verify that there are no errors in either subframe.

```
crcdetector = comm.CRCDetector([1 0 0 1],'ChecksumsPerFrame',2);
[~, err] = crcdetector(codeword)
```

```
err = 2×1

     0
     0
```

Introduce an error in the second subframe by inverting the last element of subframe 2. Pass the corrupted codeword through the CRC detector and verify that the error is detected in the second subframe.

```
codeword(end) = not(codeword(end));
[~,err] = crcdetector(codeword)
```

```
err = 2×1

     0
     1
```

Cyclic Redundancy Check of Noisy BPSK Data Frames

Use a CRC code to detect frame errors in a noisy BPSK signal.

Create a CRC generator and detector pair using a standard CRC-4 polynomial, $z^4 + z^3 + z^2 + z + 1$.

```
poly = 'z4+z3+z2+z+1';
crcgenerator = comm.CRCGenerator(poly);
crcdetector = comm.CRCDetector(poly);
```

Generate 12-bit frames of binary data and append the CRC bits. Based on the degree of the polynomial, 4 bits are appended to each frame. Apply BPSK modulation and pass the signal through an AWGN channel. Demodulate and use the CRC detector to determine if the frame is in error.

```
numFrames = 20;
frmError = zeros(numFrames,1);
```

```
for k = 1:numFrames
    data = randi([0 1],12,1);                 % Generate binary data
    encData = crcgenerator(data);                   % Append CRC bits
    modData = pskmod(encData,2);              % BPSK modulate
    rxSig = awgn(modData,5);                  % AWGN channel, SNR = 5 dB
    demodData = pskdemod(rxSig,2);            % BPSK demodulate
    [~,frmError(k)] = crcdetector(demodData); % Detect CRC errors
end
```

Identify the frames in which CRC code bit errors are detected.

```
find(frmError)
```

```
ans = 6
```

More About

Cyclic Redundancy Check Coding

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

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

When the number of checksums per frame is greater than 1, the input data frame is divided into subframes, the rule is applied to each data subframe, and individual checksums are appended to each subframe. The subframe codewords are concatenated to output one frame.

For a discussion of the supported CRC algorithms, see “Cyclic Redundancy Check Codes”.

CRC Syndrome Detector Operation

The CRC syndrome detector outputs the received message frame and a checksum error vector according to the specified generator polynomial and number of checksums per frame.

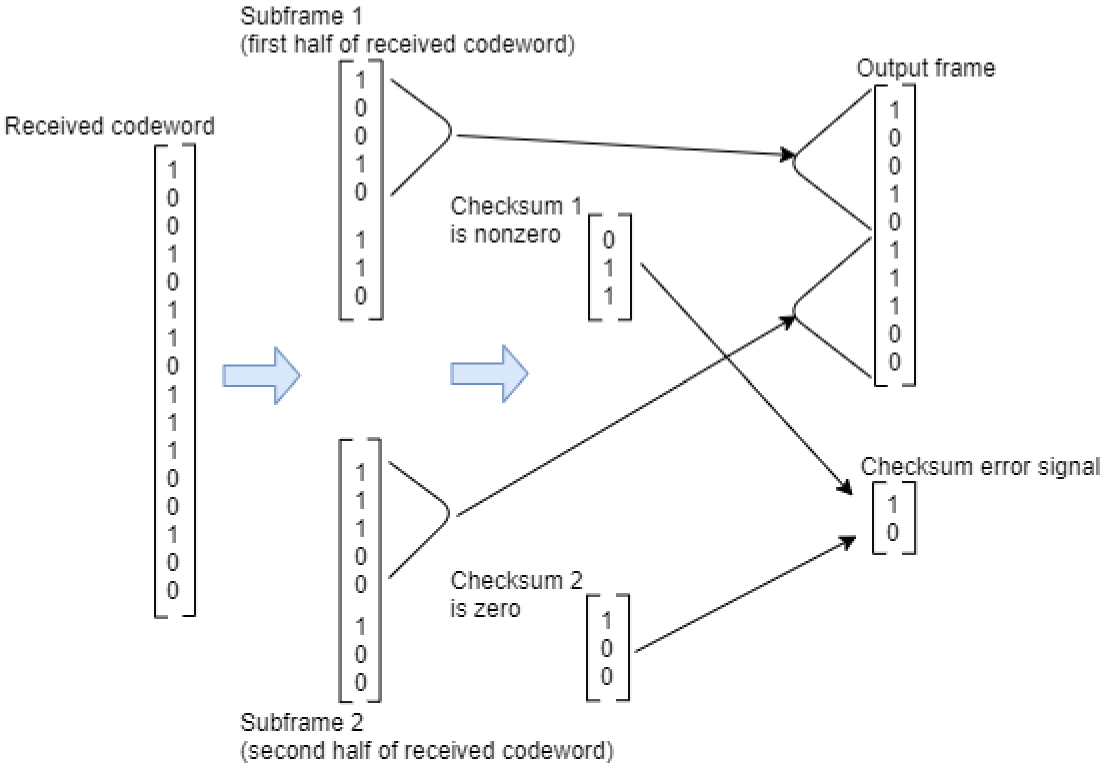
The checksum bits are removed from each subframe, so that the resulting the output frame length is $n - k \times r$, where n is the size of the received codeword, k is the number of checksums per frame, and r is the degree of the generator polynomial. The input frame must be evenly divisible by k.

For a specific initial state of the internal shift register:

1. The received codeword is divided into k equal sized subframes.
2. The CRC is removed from each of the k subframes and compared to the checksum calculated on the received codeword subframes.
3. The output frame is assembled by concatenating the subframe bits of the k subframes and then output as a column vector.

4 The checksum error is output as a binary column vector of length *k*. An element value of 0 indicates an error-free received subframe, and an element value of 1 indicates an error occurred in the received subframe.

For the scenario shown here, a 16-bit codeword is received, a third degree generator polynomial computes the CRC checksum, the initial state is 0, and the number of checksums per frame is 2.



Since the number of checksums per frame is 2 and the generator polynomial degree is 3, the received codeword is split in half and two checksums of size 3 are computed, one for each half of the received codeword. The initial states are not shown, because an initial state of `[0]` does not affect the output of the CRC algorithm. The output frame contains the concatenation of the two halves of the received codeword as a single vector of size 10. The checksum error signal output contains a 2-by-1 binary frame vector whose entries depend on whether the computed checksums are zero. As shown in the figure, the first checksum is nonzero and the second checksum is zero, indicating an error occurred in reception of the first half of the codeword.

References

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.CRCGenerator` | `comm.HDLCRCDetector`

Blocks

General CRC Syndrome Detector

Topics

“Cyclic Redundancy Check Codes”

comm.CRCGenerator

Package: `comm`

Generate CRC code bits and append to input data

Description

The `comm.CRCGenerator` System object generates cyclic redundancy check (CRC) code bits for each input frame and appends them to the frame. For more information, see "CRC Generator Operation" on page 3-399.

To generate CRC code bits for each input frame and append them to the frame:

1. Create the `comm.CRCGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
crcgenerator = comm.CRCGenerator
crcgenerator = comm.CRCGenerator(Name,Value)
crcgenerator = comm.CRCGenerator(poly,Name,Value)
```

Description

`crcgenerator = comm.CRCGenerator` creates a CRC code generator System object. This object generates CRC bits according to a specified generator polynomial and appends them to the input frame.

`crcgenerator = comm.CRCGenerator(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.CRCGenerator('Polynomial','z^16 + z^14 + z + 1')` configures the CRC generator System object to append CRC-16 cyclic redundancy check bits to the input frame. Enclose each property name in quotes.

`crcgenerator = comm.CRCGenerator(poly,Name,Value)` creates a CRC code generator System object. This object has the `Polynomial` property set to `poly`, and the other specified properties set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Polynomial` — Generator polynomial
`'z^16 + z^12 + z^5 + 1'` (default) | polynomial character vector | binary row vector | integer row vector

Generator polynomial for the CRC algorithm, specified as one of the following:

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial x^3+ z^2+ 1.
- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, `[3 2 0]` represents the polynomial $z^3 + z^2 + 1$.

For more information, see "Character Representation of Polynomials".

Some commonly used generator polynomials include:

CRC method	Generator polynomial
CRC-32	`'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'`
CRC-24	`'z^24 + z^23 + z^14 + z^12 + z^8 + 1'`
CRC-16	`'z^16 + z^15 + z^2 + 1'`
Reversed CRC-16	`'z^16 + z^14 + z + 1'`
CRC-8	`'z^8 + z^7 + z^6 + z^4 + z^2 + 1'`
CRC-4	`'z^4 + z^3 + z^2 + z + 1'`

Example: `'z^7 + z^2 + 1'`, `[1 0 0 0 0 1 0 1]`, and `[7 2 0]` represent the same polynomial, $p(z) = z^7 + z^2 + 1$.

Data Types: `double` | `char`

`InitialConditions` — Initial states of internal shift register
`0` (default) | `1` | binary row vector

Initial states of the internal shift register, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial.

Data Types: `logical`

`DirectMethod` — Use direct algorithm for CRC checksum calculations
`false` (default) | `true`

Use direct algorithm for CRC checksum calculations, specified as `false` or `true`.

When you set this property to `true`, the object uses the direct algorithm for CRC checksum calculations. When you set this property to `false`, the object uses the non-direct algorithm for CRC checksum calculations.

For more information on direct and non-direct algorithms, see "Error Detection and Correction".

Data Types: `logical`

`ReflectInputBytes` — Reflect input bytes

`false` (default) | `true`

Reflect input bytes, specified as `false` or `true`. Set this property to `true` to flip the input frame on a bytewise basis before entering the data into the shift register.

When you set this property to `true`, the input frame length divided by the value of the `ChecksumsPerFrame` property must be an integer and a multiple of `8`.

Data Types: `logical`

`ReflectChecksums` — Reflect checksums before final XOR

`false` (default) | `true`

Reflect checksums before final XOR, specified as `false` or `true`. Set this property to `true` to flip the CRC checksums around their centers after the input data are completely through the shift register.

Data Types: `logical`

`FinalXOR` — Final XOR

0 (default) | binary scalar | binary vector

Final XOR, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. The XOR operation runs using the value of the `FinalXOR` property and the CRC checksum before comparing with the input checksum. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial. A setting of `0` is equivalent to no XOR operation.

Data Types: `logical`

`ChecksumsPerFrame` — Number of checksums calculated for each frame

1 (default) | positive integer

Number of checksums calculated for each frame, specified as a positive integer. For more information, see "CRC Generator Operation" on page 3-399.

Data Types: `double`

Usage

Syntax

```
codeword = crcgenerator(x)
```

Description

`codeword = crcgenerator(x)` generates CRC code bits for each input frame and appends them to the frame.

Input Arguments

x — Input signal
binary column vector

Input signal, specified as a binary column vector. The length of the input frame must be a multiple of the value of the `ChecksumsPerFrame` property. If the input data type is double, the least significant bit is used as the binary value. For more information, see “CRC Generator Operation” on page 3-399.

Data Types: `double` | `logical`

Output Arguments

`codeword` — Output codeword frame
binary column vector

Output codeword frame, returned as a binary column vector that inherits the data type of the input signal. The output contains the input frames with the CRC code sequence bits appended.

The length of the output codeword frame is $m + k * r$, where m is the size of the input message, k is the number of checksums per input frame, and r is the degree of the generator polynomial. For more information, see “CRC Generator Operation” on page 3-399.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate CRC-8 Checksum

Generate a CRC-8 checksum for the example shown in 802.11™-2016[1] on page 3-0 , section 21.3.10.3 and compare with the expected CRC.

Create a CRC Generator System object™. To align with the CRC calculation in 802.11-20016, the System object™ sets the generator polynomial as $z^8 + z^2 + z + 1$, initial states to 1, direct method, and final XOR to 1.

```
crc8 = comm.CRCGenerator('Polynomial','z^8 + z^2 + z + 1', ...
    'InitialConditions',1,'DirectMethod',true,'FinalXOR',1)
```

```
crc8 = 
  comm.CRCGenerator with properties:
```

```
           Polynomial: 'z^8 + z^2 + z + 1'
    InitialConditions: 1
         DirectMethod: true
    ReflectInputBytes: false
     ReflectChecksums: false
             FinalXOR: 1
    ChecksumsPerFrame: 1
```

Process one input frame according to the example from the 802.11-2016 standard in section 21.3.10.3. In the example, the input bit stream {m0, … m22} is {1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1} and the expected CRC checksum {c7, … c0} is {0 0 0 1 1 1 0 0}.

```
x = [1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1]';
expectedChecksum = [0 0 0 1 1 1 0 0]';
checksumLength = length(expectedChecksum);
```

The generated CRC checksum is compared to the expected checksum.

```
codeword = crc8(x);
checksum = codeword(end-checksumLength+1:end);
isequal(checksum,expectedChecksum)
```

```
ans = logical
   1
```

References

[1] IEEE Std 802.11™-2016 IEEE Standard for Information Technology—Local and Metropolitan Area Networks—Specific Requirements Part 11: Wireless LAN MAC and PHY Specifications.

CRC Detection of Errors in a Random Message

Pass binary data through a CRC generator, introduce a bit error, and detect the error using a CRC detector.

Create a random binary vector.

```
x = randi([0 1],12,1);
```

Encode the input message frame using a CRC generator with the `ChecksumsPerFrame` property set to `2`. This subdivides the incoming frame into two equal-length subframes.

```
crcgenerator = comm.CRCGenerator([1 0 0 1],'ChecksumsPerFrame',2);
codeword = crcgenerator(x);
```

Decode the codeword and verify that there are no errors in either subframe.

```
crcdetector = comm.CRCDetector([1 0 0 1],'ChecksumsPerFrame',2);
[~, err] = crcdetector(codeword)
```

```
err = 2×1

     0
     0
```

Introduce an error in the second subframe by inverting the last element of subframe 2. Pass the corrupted codeword through the CRC detector and verify that the error is detected in the second subframe.

```
codeword(end) = not(codeword(end));
[~,err] = crcdetector(codeword)
```

```
err = 2×1

     0
     1
```

Cyclic Redundancy Check of Noisy BPSK Data Frames

Use a CRC code to detect frame errors in a noisy BPSK signal.

Create a CRC generator and detector pair using a standard CRC-4 polynomial, $z^4 + z^3 + z^2 + z + 1$.

```
poly = 'z4+z3+z2+z+1';
crcgenerator = comm.CRCGenerator(poly);
crcdetector = comm.CRCDetector(poly);
```

Generate 12-bit frames of binary data and append the CRC bits. Based on the degree of the polynomial, 4 bits are appended to each frame. Apply BPSK modulation and pass the signal through an AWGN channel. Demodulate and use the CRC detector to determine if the frame is in error.

```
numFrames = 20;
frmError = zeros(numFrames,1);

for k = 1:numFrames
    data = randi([0 1],12,1);                 % Generate binary data
    encData = crcgenerator(data);                   % Append CRC bits
    modData = pskmod(encData,2);              % BPSK modulate
    rxSig = awgn(modData,5);                  % AWGN channel, SNR = 5 dB
    demodData = pskdemod(rxSig,2);            % BPSK demodulate
    [~,frmError(k)] = crcdetector(demodData); % Detect CRC errors
end
```

Identify the frames in which CRC code bit errors are detected.

```
find(frmError)
```

```
ans = 6
```

CRC-16-CCITT Generator for X.25

Create a CRC-16-CCITT generator as described in Section 2.2.7.4 of ITU-T Recommendation X-25[1] on page 3-0 using the input data and expected frame check sequence (FCS) from Example 2 in Appendix I, I.1.

Create an unnumbered acknowledgement (UA) response frame where address = B and F = 1.

```
Address = [1 0 0 0 0 0 0 0];
UA = [1 1 0 0 1 1 1 0];
input = [Address UA]';
expectedChecksum = [1 1 0 0 0 0 0 1 1 1 1 0 1 0 1 0]'; % Expected FCS
checksumLength = 16;

crcGen = comm.CRCGenerator(...
    'Polynomial','X^16 + X^12 + X^5 + 1',...
    'InitialConditions',1,...
    'DirectMethod',true,...
    'FinalXOR',1);
crcSeq = crcGen(input);
checkSum =  crcSeq(end-checksumLength+1:end);
```

Compare calculated checksum with the expected checksum.

```
isequal(expectedChecksum,checkSum)
```

```
ans = logical
   1
```

References

[1] ITU Telecommunication Standardization Sector. *Series X: Data Networks And Open System Communication. Public data networks – Interfaces. 1997*

CRC-32 Generator for Ethernet

Create a CRC-32 code for the frame check sequence (FCS) field for Ethernet as described in Section 3.2.9 of the IEEE Standard for Ethernet[1] on page 3-0   .

```
rng(1865);  % Seed the random number generator for repeatable results
```

Initialize a message with random data to represent the protected fields of the MAC frame, specifically the destination address, source address, length or type field, MAC client data, and padding.

```
data = randi([0,1],100,1);
```

Specify the CRC-32 generating polynomial used for encoding Ethernet messages.

```
poly = [32,26,23,22,16,12,11,10,8,7,5,4,2,1,0];
```

Calculate the CRC by following the steps specified in the standard and using the nondirect method to generate the CRC code.

```
dataN = [not(data(1:32));data(33:end)];    % Section 3.2.9 step a) and b)
crcGen1 = comm.CRCGenerator(...
    'Polynomial', poly, ...
    'InitialConditions', 0, ...
    'DirectMethod', false, ...
    'FinalXOR', 1);
seq = crcGen1(dataN);                      % Section 3.2.9 step c), d) and e)
csNondirect = seq(end-31:end);
```

Calculate the CRC by following the steps specified in the standard and using the direct method to generate the CRC code.

```
crcGen2 = comm.CRCGenerator(...
    'Polynomial', poly, ...
    'InitialConditions', 1, ...
    'DirectMethod', true, ...
    'FinalXOR', 1);
txSeq = crcGen2(data);
csDirect = txSeq(end-31:end);
```

Compare the generated CRC codes by using the nondirect and direct methods.

```
disp([csNondirect';csDirect']);
```

```
  Columns 1 through 13

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

  Columns 14 through 26

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

  Columns 27 through 32

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

```
isequal(csNondirect,csDirect)
```

```
ans = logical
   1
```

```
rng('default');  % Reset the random number generator
```

References

[1] IEEE Computer Society. *IEEE Standard for Ethernet: Std 802.3-2012*. New York, NY: 2012.

More About

Cyclic Redundancy Check Coding

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

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

When the number of checksums per frame is greater than 1, the input data frame is divided into subframes, the rule is applied to each data subframe, and individual checksums are appended to each subframe. The subframe codewords are concatenated to output one frame.

For a discussion of the supported CRC algorithms, see “Cyclic Redundancy Check Codes”.

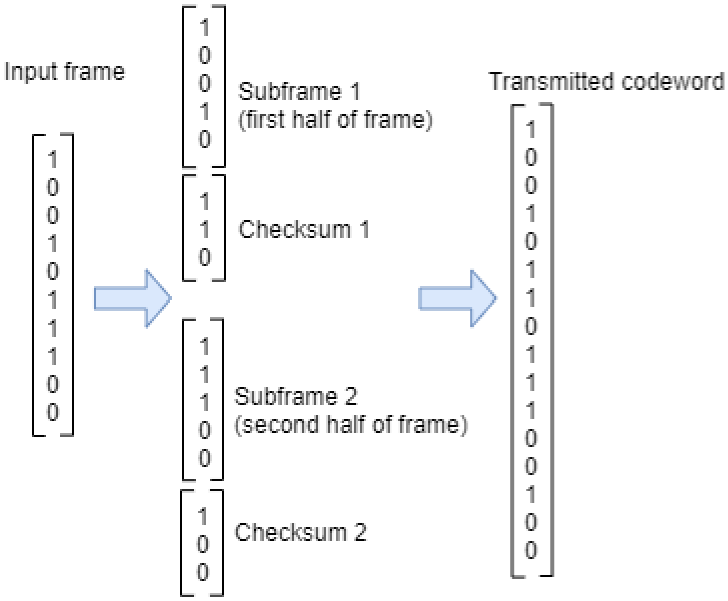
CRC Generator Operation

The CRC generator appends CRC checksums to the input frame according to the specified generator polynomial and number of checksums per frame.

For a specific initial state of the internal shift register and *k* checksums per input frame:

1. The input signal is divided into *k* subframes of equal size.
2. Each of the *k* subframes are prefixed with the initial states vector.
3. The CRC algorithm is applied to each subframe.
4. The resulting checksums are appended to the end of each subframe.
5. The subframes are concatenated and output as a column vector.

For the scenario shown here, a 10-bit frame is input, a third degree generator polynomial computes the CRC checksum, the initial state is 0, and the number of checksums per frame is 2.



The input frame is divided into two subframes of size 5 and checksums of size 3 are computed and appended to each subframe. The initial states are not shown, because an initial state of `[0]` does not affect the output of the CRC algorithm. The output transmitted codeword frame has the size 5 + 3 + 5 + 3 = 16.

References

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

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.CRCDetector` | `comm.HDLCRCGenerator`

Blocks
General CRC Generator

Topics
“Cyclic Redundancy Check Codes”

comm.DBPSKDemodulator

Package: comm

Demodulate using DBPSK method

Description

The DBPSKDemodulator object demodulates a signal that was modulated using the differential binary phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using differential binary phase shift keying:

1. Define and set up your DBPSK demodulator object. See “Construction” on page 3-401.
2. Call step to demodulate a signal according to the properties of comm.DBPSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.DBPSKDemodulator creates a demodulator System object, H. This object demodulates the input signal using the differential binary phase shift keying (DBPSK) method.

H = comm.DBPSKDemodulator(Name,Value) creates a DBPSK demodulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.DBPSKDemodulator(PHASE,Name,Value) creates a DBPSK demodulator object, H. This object has the PhaseRotation property set to PHASE and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated bits in radians as a real scalar. The default is 0. This value corresponds to the phase difference between previous and current modulated bits when the input is zero.

OutputDataType

Data type of output

Specify output data type as one of Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical. The default is Full precision. When you set this property to Full precision, the output data type has the same data type as the input. In this case, that value must be a double- or single-precision data type.

Methods

reset	Reset states of DBPSK demodulator object
step	Demodulate using DBPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

DBPSK Signal in AWGN

Create a DBPSK modulator and demodulator pair.

```
dbpskmod = comm.DBPSKModulator(pi/4);
dpbpskdemod = comm.DBPSKDemodulator(pi/4);
```

Create an error rate calculator. Set the ComputationDelay property to 1 to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50-bit frames
- DBPSK modulate
- Pass through AWGN channel
- DBPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],50,1);
    modSig = dbpskmod(txData);
    rxSig = awgn(modSig,7);
    rxData = dpbpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0040
```

```
numErrors = errorStats(2)
```

```
numErrors = 20
```

```
numBits = errorStats(3)
numBits = 4999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DBPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: `comm.DBPSKDemodulator`
Package: `comm`

Reset states of DBPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DBPSKDemodulator` object, `H`.

step

System object: `comm.DBPSKDemodulator`
Package: `comm`

Demodulate using DBPSK method

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the DBPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision data type scalar or column vector.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DBPSKModulator

Package: `comm`

Modulate using DBPSK method

Description

The `DBPSKModulator` object modulates using the differential binary phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using differential binary phase shift keying:

1. Define and set up your DBPSK modulator object. See "Construction" on page 3-406.
2. Call `step` to modulate a signal according to the properties of `comm.DBPSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.DBPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the differential binary phase shift keying (DBPSK) method.

`H = comm.DBPSKModulator(Name,Value)` creates a DBPSK modulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.DBPSKModulator(PHASE,Name,Value)` creates a DBPSK modulator object, `H`. This object has the `PhaseRotation` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

`PhaseRotation`

Additional phase shift

Specify the additional phase difference between previous and current modulated bits in radians as a real scalar value. The default is `0`. This value corresponds to the phase difference between previous and current modulated bits when the input is zero.

`OutputDataType`

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of DBPSK modulator object
step	Modulate using DBPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

DBPSK Signal in AWGN

Create a DBPSK modulator and demodulator pair.

```
dbpskmod = comm.DBPSKModulator(pi/4);
dpbpskdemod = comm.DBPSKDemodulator(pi/4);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50-bit frames
- DBPSK modulate
- Pass through AWGN channel
- DBPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],50,1);
    modSig = dbpskmod(txData);
    rxSig = awgn(modSig,7);
    rxData = dpbpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0040
```

```
numErrors = errorStats(2)
```

```
numErrors = 20
```

```
numBits = errorStats(3)
```

```
numBits = 4999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DBPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: `comm.DBPSKModulator`
Package: `comm`

Reset states of DBPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DBPSKModulator` object, `H`.

step

System object: `comm.DBPSKModulator`
Package: `comm`

Modulate using DBPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the DBPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be a numeric or logical data type column vector of bits.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.Descrambler

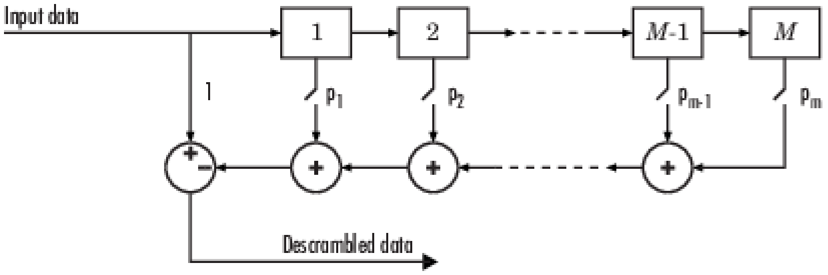
Package: `comm`

Descramble input signal

Description

The comm.Descrambler object descrambles a scalar or column vector input signal. The comm.Descrambler object is the inverse of the `comm.Scrambler` object. If you use the `comm.Scrambler` object in a transmitter, then you use the comm.Descrambler object in the related receiver.

This schematic shows the descrambler operation. The adders and subtracter operate modulo *N*, where *N* is the value specified by the Calculation base property.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Polynomial property, you specify the on or off state for each switch in the descrambler. To make the `comm.Descrambler` object reverse the operation of the `comm.Scrambler` object, use the same property settings in both objects. If there is no signal delay between the scrambler and the descrambler, then the InitialConditions in the two objects must be the same.

To descramble an input signal:

1. Create the `comm.Descrambler` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
descrambler = comm.Descrambler
descrambler = comm.Descrambler(base,poly,cond)
descrambler = comm.Descrambler(___,Name,Value)
```

Description

`descrambler = comm.Descrambler` creates a descrambler System object. This object descrambles the input data by using a linear feedback shift register that you specify with the Polynomial property.

`descrambler = comm.Descrambler(base,poly,cond)` creates the descrambler object with the CalculationBase property set to `base`, the Polynomial property set to `poly`, and the InitialConditions property set to `cond`.

Example: `comm.Descrambler(8,'1 + z^-2 + z^-3 + z^-5 + z^-7',[0 3 2 2 5 1 7])` sets the calculation base to `8`, and the descrambler polynomial and initial conditions as specified.

`descrambler = comm.Descrambler(___,Name,Value)` sets properties using one or more name-value pairs and either of the previous syntaxes. Enclose each property name in single quotes.

Example: `comm.Descrambler('CalculationBase',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`CalculationBase` — Range of input data

4 (default) | nonnegative integer

Range of input data used in the descrambler for modulo operations, specified as a nonnegative integer. The input and output of this object are integers from `0` to `CalculationBase – 1`.

Data Types: `double`

`Polynomial` — Connections for linear feedback shift registers

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Connections for linear feedback shift registers in the descrambler, specified as a character vector, integer vector, or binary vector. The `Polynomial` property defines if each switch in the descrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1z^{-1} + p_2z^{-2} + ...$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that have a coefficient of 1. In this case, the order of the descramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Data Types: `double` | `char`

`InitialConditionsSource` — Initial conditions source

`'Property'` (default) | `'Input port'`

- `'Property'` – Specify descrambler initial conditions by using the InitialConditions property.
- `'Input port'` – Specify descrambler initial conditions by using an additional input argument, initcond, when calling the object.

Data Types: `char`

`InitialConditions` — Initial conditions of descrambler registers

`[0 1 2 3]` (default) | nonnegative integer vector

Initial conditions of descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of `InitialConditions` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

`ResetInputPort` — Descrambler state reset port

`false` (default) | `true`

Descrambler state reset port, specified as `false` or `true`. If `ResetInputPort` is `true`, you can reset the descrambler object by using an additional input argument, reset, when calling the object.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

Usage

Syntax

```
descrambledOut = descrambler(signal)
descrambledOut = descrambler(signal,initcond)
descrambledOut = descrambler(signal,reset)
```

Description

`descrambledOut = descrambler(signal)` descrambles the input signal. The output is the same data type and length as the input vector.

`descrambledOut = descrambler(signal,initcond)`provides an additional input with values specifying the initial conditions of the linear feedback shift register.

This syntax applies when you set the InitialConditionsSource property of the object to `'Input port'`.

`descrambledOut = descrambler(signal,reset)` provides an additional input indicating whether to reset the state of the descrambler.

This syntax applies when you set the InitialConditionsSource property of the object to `'Property'` and the ResetInputPort to `true`.

Input Arguments

`signal` — Input signal
column vector

Input signal, specified as a column vector.

Example: `descrambledOut = descrambler([0 1 1 0 1 0])`

Data Types: `double` | `logical`

`initcond` — Initial register condition
nonnegative integer column vector

Initial descrambler register conditions when the simulation starts, specified as a nonnegative integer column vector. The length of `initcond` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Example: `descrambledOut = descrambler(signal,[0 1 1 0])` corresponds to possible initial register states for a descrambler with a polynomial order of 4 and a calculation base of 2 or higher.

Data Types: `double`

`reset` — Reset initial state of descrambler
scalar

Reset initial state of the descrambler when the simulation starts, specified as a scalar. When the value of `reset` is nonzero, the object is reset before it is called.

Example: `descrambledOut = descrambler(signal,0)` descrambles the input signal without resetting the descrambler states.

Data Types: `double`

Output Arguments

`out` — Descrambled output
column vector

Descrambled output, returned as a column vector with the same data type and length as signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Scramble and Descramble Data

Scramble and descramble 8-ary data using `comm.Scrambler` and `comm.Descrambler` System objects™ having a calculation base of 8.

Create scrambler and descrambler objects, specifying the calculation base, polynomial, and initial conditions using input arguments. The scrambler and descrambler polynomials are specified with different but equivalent data formats.

```
N = 8;
scrambler = comm.Scrambler(N,'1 + z^-2 + z^-3 + z^-5 + z^-7', ...
    [0 3 2 2 5 1 7]);
descrambler = comm.Descrambler(N,[1 0 1 1 0 1 0 1], ...
    [0 3 2 2 5 1 7]);
```

Scramble and descramble random integers. Display the original data, scrambled data, and descrambled data sequences.

```
data = randi([0 N-1],5,1);
scrData = scrambler(data);
deScrData = descrambler(scrData);
[data scrData deScrData]
```

```
ans = 5×3

     6     7     6
     7     5     7
     1     7     1
     7     0     7
     5     3     5
```

Verify that the descrambled data matches the original data.

```
isequal(data,deScrData)
```

```
ans = logical
   1
```

Scramble and Descramble Data with Changing Initial Conditions

Scramble and descramble quaternary data while changing the initial conditions between function calls.

Create scrambler and descrambler System objects having a calculation base of 4. Set the `InitialConditionsSource` property to `'Input port'` so you can set the initial conditions as an argument to the object.

```
N = 4;
scrambler = comm.Scrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
descrambler = comm.Descrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
```

Preallocate memory for the error vector which will be used to store errors output by the `symerr` function.

```
errVec = zeros(10,1);
```

Scramble and descramble random integers while changing the initial conditions, `initCond`, each time the loop executes. Use the `symerr` function to determine if the scrambling and descrambling operations result in symbol errors.

```
for k = 1:10
    initCond = randperm(3)';
    data = randi([0 N-1],5,1);
    scrData = scrambler(data,initCond);
    deScrData = descrambler(scrData,initCond);
    errVec(k) = symerr(data,deScrData);
end
```

Examine `errVec` to verify that the output from the descrambler matches the original data.

```
errVec
```

```
errVec = 10×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.PNSequence` | `comm.Scrambler`

Blocks

Descrambler

Introduced in R2012a

comm.DifferentialDecoder

Package: `comm`

Decode binary signal using differential decoding

Description

The `DifferentialDecoder` object decodes the binary input signal. The output is the logical difference between the consecutive input element within a channel.

To decode a binary signal using differential decoding:

1. Define and set up your differential decoder object. See “Construction” on page 3-417.
2. Call `step` to decode a binary signal according to the properties of `comm.DifferentialDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.DifferentialDecoder` creates a differential decoder System object, `H`. This object decodes a binary input signal that was previously encoded using a differential encoder.

`H = comm.DifferentialDecoder(Name,Value)` creates object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`InitialCondition`

Initial value used to generate initial output

Specify the initial condition as a real scalar. This property can have a logical, numeric, or fixed-point (embedded.fi object) data type. The default is `0`. The object treats nonbinary values as binary signals.

Methods

reset	Reset states of differential decoder object
step	Decode binary signal using differential decoding

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Decode Differentially Encoded Signal

Create a differential encoder and decoder pair.

```
diffEnc = comm.DifferentialEncoder;
diffDec = comm.DifferentialDecoder;
```

Generate random binary data. Differentially encode and decode the data.

```
data = randi([0 1],100,1);
encData = diffEnc(data);
decData = diffDec(encData);
```

Determine the number of errors between the original data and the decoded data.

```
numErrors = biterr(data,decData)
```

```
numErrors = 0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Differential Decoder block reference page. The object properties correspond to the block parameters, except:
The object only supports single channel, column vector inputs.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DifferentialEncoder`

Introduced in R2012a

reset

System object: `comm.DifferentialDecoder`
Package: `comm`

Reset states of differential decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DifferentialDecoder` object, `H`.

step

System object: `comm.DifferentialDecoder`
Package: `comm`

Decode binary signal using differential decoding

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` decodes the differentially encoded input data, `X`, and returns the decoded data, `Y`. The input `X` must be a column vector of data type logical, numeric, or fixed-point (embedded.fi objects). `Y` has the same data type as `X`. The object treats non-binary inputs as binary signals. The object computes the initial output value by performing an Xor operation of the value in the `InitialCondition` property and the first element of the vector you input the first time you call the step method.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DifferentialEncoder

Package: comm

Encode binary signal using differential coding

Description

The DifferentialEncoder object encodes the binary input signal within a channel. The output is the logical difference between the current input element and the previous output element.

To encode a binary signal using differential coding:

1. Define and set up your differential encoder object. See "Construction" on page 3-421.
2. Call step to encode a binary signal according to the properties of comm.DifferentialEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.DifferentialEncoder creates a differential encoder System object, H. This object encodes a binary input signal by calculating its logical difference with the previously encoded data.

H = comm.DifferentialEncoder(Name,Value) creates object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

InitialCondition

Initial value used to generate initial output

Specify the initial condition as a real scalar. This property can have a logical, numeric, or fixed-point (embedded.fi object) data type. The default is 0. The object treats nonbinary values as binary signals.

Methods

reset	Reset states of differential encoder object
step	Encode binary signal using differential coding

Common to All System Objects	
release	Allow System object property value changes

Examples

Differentially Encode Binary Data

Create a differential encoder object.

```
diffEnc = comm.DifferentialEncoder;
```

Generate random binary data. Encode the data.

```
data = randi([0 1],10,1);
encData = diffEnc(data)
```

```
encData = 10×1

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

Algorithms

This object implements the algorithm, inputs, and outputs described on the Differential Encoder block reference page. The object properties correspond to the block parameters, except:
The object only supports single channel, column vector inputs.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DifferentialDecoder`

Introduced in R2012a

reset

System object: `comm.DifferentialEncoder`
Package: `comm`

Reset states of differential encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DifferentialEncoder` object, `H`.

step

System object: comm.DifferentialEncoder
Package: comm

Encode binary signal using differential coding

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) encodes the binary input data, X, and returns the differentially encoded data, Y. The input X must be a column vector of data type logical, numeric, or fixed-point (embedded.fi objects). Y has the same data type as X. The object treats non-binary inputs as binary signals. The object computes the initial output value by performing an Xor operation of the value in the InitialCondition property and the first element of the vector you input the first time you call the step method.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.DiscreteTimeVCO

Package: `comm`

Generate variable frequency sinusoid

Description

The `DiscreteTimeVCO` (voltage-controlled oscillator) object generates a signal whose frequency shift from the quiescent frequency property is proportional to the input signal. The input signal is interpreted as a voltage.

To generate a variable frequency sinusoid:

1. Define and set up your discrete time voltage-controlled oscillator object. See "Construction" on page 3-425 .
2. Call `step` to generate a variable frequency sinusoid according to the properties of `comm.DiscreteTimeVCO`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.DiscreteTimeVCO` creates a discrete-time voltage-controlled oscillator (VCO) System object, `H`. This object generates a sinusoidal signal with the frequency shifted from the specified quiescent frequency to a value proportional to the input signal.

`H = comm.DiscreteTimeVCO(Name,Value)` creates a discrete-time VCO object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`OutputAmplitude`

Amplitude of output signal

Specify the amplitude of the output signal as a double- or single-precision, scalar value. The default is `1`. This property is tunable.

`QuiescentFrequency`

Frequency of output signal when input is zero

Specify the quiescent frequency of the output signal in Hertz, as a double- or single-precision, real, scalar value. The default is `10`. This property is tunable.

`Sensitivity`

Sensitivity of frequency shift of output signal

Specify the sensitivity of the output signal frequency shift to the input as a double- or single-precision, real, scalar value. The default is `1`. This value scales the input voltage and, consequently, the shift from the quiescent frequency value. The property measures Sensitivity in Hertz per volt. This property is tunable.

`InitialPhase`

Initial phase of output signal

Specify the initial phase of the output signal, in radians, as a double or single precision, real, scalar value. The default is `0`.

`SampleRate`

Sample rate of input

Specify the sample rate of the input, in Hertz, as a double- or single-precision, positive, scalar value. The default is `100`.

Methods

reset	Reset states of discrete-time VCO object
step	Generate variable frequency sinusoid

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate FSK Signal Using Discrete Time VCO

Create a signal source System object™.

```
reader = dsp.SignalSource;
```

Generate random data and apply rectangular pulse shaping.

```
reader.Signal = randi([0 7],10,1);
reader.Signal = rectpulse(reader.Signal,100);
```

Create a signal logger and discrete time VCO System objects.

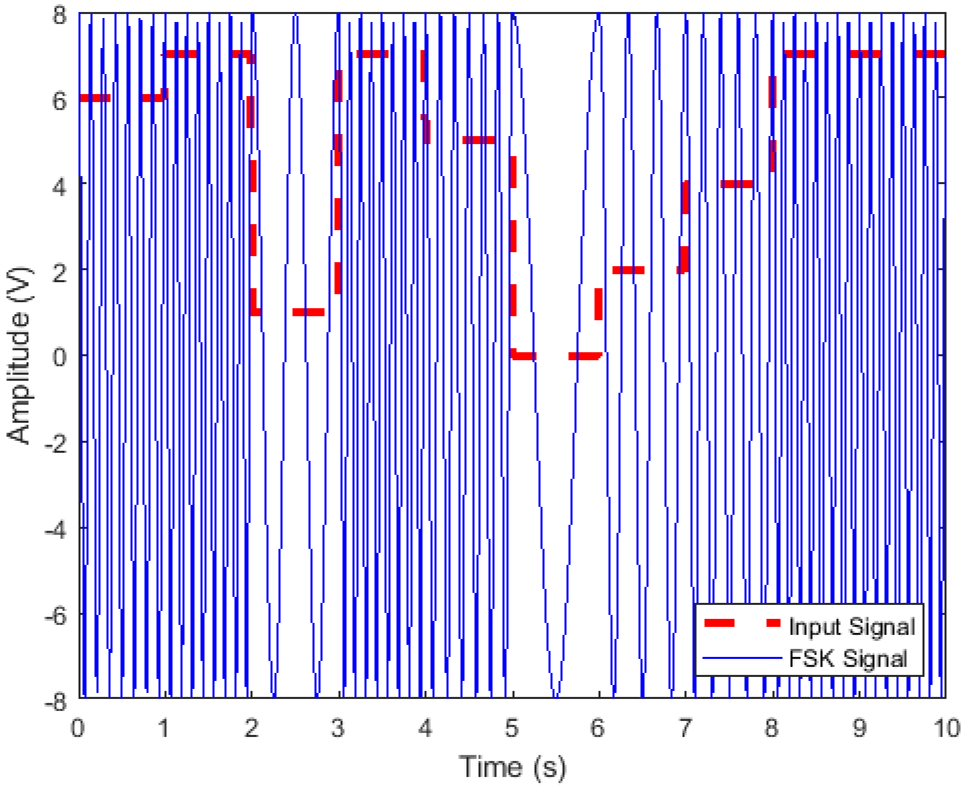
```
logger = dsp.SignalSink;
discreteVCO = comm.DiscreteTimeVCO('OutputAmplitude',8,'QuiescentFrequency',1);
```

Generate an FSK signal.

```
while(~isDone(reader))
    sig = reader();
    y = discreteVCO(sig);
    logger(y);
end
oscsig = logger.Buffer;
```

Plot the generated FSK signal.

```
t = (0:length(oscsig)-1)'/discreteVCO.SampleRate;
plot(t,reader.Signal,'--r', 'LineWidth',3)
hold on
plot(t,oscsig,'-b');
hold off
xlabel('Time (s)')
ylabel('Amplitude (V)')
legend('Input Signal','FSK Signal','location','se')
```



Algorithms

This object implements the algorithm, inputs, and outputs as described on the Discrete-Time VCO block reference page. However, this object and the corresponding block may not generate the exact same outputs for single-precision inputs or property values due to the following differences in casting strategies and arithmetic precision issues:

- The block always casts the result of intermediate mathematical operations to the input data type. The object does not cast intermediate results and MATLAB decides the data type. The object casts the final output to the input data type.

- You can specify the `SampleRate` object property in single-precision or double-precision. The block does not allow this.
- In arithmetic operations with more than two operands with mixed data types, the result may differ depending on the order of operation. Thus, the following calculation may also contribute to the difference in the output of the block and the object:

  input * sensitivity * `sampleTime`
- The block performs this calculation from left to right. However, since sensitivity * `sampleTime` is a one-time calculation, the object calculates this in the following manner:

  input * (sensitivity * `sampleTime`)

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer`

Introduced in R2012a

reset

System object: `comm.DiscreteTimeVCO`
Package: `comm`

Reset states of discrete-time VCO object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DiscreteTimeVCO` object, `H`.

step

System object: `comm.DiscreteTimeVCO`
Package: `comm`

Generate variable frequency sinusoid

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` generates a sinusoidal signal, `Y`, with frequency shifted, from the value you specify in the `QuiescentFrequency` property, to a value proportional to the input signal, `X`. The input, `X`, must be a double or single precision, real, scalar value. The output, `Y`, has the same data type and size as the input, `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DPD

Package: `comm`

Digital predistorter

Description

The `comm.DPD` System object applies digital predistortion (DPD) to a complex baseband signal by using a memory polynomial to compensate for nonlinearities in a power amplifier. For more information, see “Digital Predistortion” on page 3-435.

To predistort signals:

1. Create the `comm.DPD` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
dpd = comm.DPD
dpd = comm.DPD(Name,Value)
```

Description

`dpd = comm.DPD` creates a digital predistorter System object to predistort a signal.

`dpd = comm.DPD(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.DPD('PolynomialType','Cross-term memory polynomial')` configures the predistorter System object to predistort the input signal by using a memory polynomial with cross terms. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`PolynomialType` — Polynomial type
`'Memory polynomial'` (default) | `'Cross-term memory polynomial'`

Polynomial type used for predistortion, specified as one of these values:

- `'Memory polynomial'` — Predistorts the input signal by using a memory polynomial without cross terms.
- `'Cross-term memory polynomial'` — Predistorts the input signal by using a memory polynomial with cross terms.

For more information, see “Digital Predistortion” on page 3-435.

`Coefficients` — Memory-polynomial coefficients
`complex([1 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0])` (default) | matrix

Memory-polynomial coefficients, specified as a matrix. The number of rows in the matrix must equal the memory depth of the memory polynomial.

- If `PolynomialType` is `'Memory polynomial'`, the number of columns in the matrix is the degree of the memory polynomial.
- If `PolynomialType` is `'Cross-term memory polynomial'`, the number of columns in the matrix must equal $m(n\text{-}1)+1$. m is the memory depth of the polynomial, and n is the degree of the memory polynomial.

For more information, see “Digital Predistortion” on page 3-435.

Data Types: `double`
Complex Number Support: Yes

Usage

Syntax

`out = dpd(in)`

Description

`out = dpd(in)` predistorts a complex baseband signal by using a memory polynomial to compensate for nonlinearities in a power amplifier.

Input Arguments

`in` — Input baseband signal
column vector

Input baseband signal, specified as a column vector.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`out` — Predistorted baseband signal
column vector

Predistorted baseband signal, returned as a column vector of the same length as the input signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Predistort Power Amplifier Input Signal

Apply digital predistortion (DPD) to a power amplifier input signal. The DPD coefficient estimator System object uses a captured signal containing power amplifier input and output signals to determine the predistortion coefficient matrix.

Load a file containing the input and output signals for the power amplifier.

```
load('commpowamp_dpd_data.mat','PA_input','PA_output')
```

Generate a DPD coefficient estimator System object and a raised cosine transmit filter System object.

```
estimator = comm.DPDCoefficientEstimator( ...
    'DesiredAmplitudeGaindB',10, ...
    'PolynomialType','Memory polynomial', ...
    'Degree',5,'MemoryDepth',3,'Algorithm','Least squares');

rctFilt = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',2);
```

Estimate the digital predistortion memory-polynomial coefficients.

```
coef = estimator(PA_input,PA_output);
```

Generate a DPD System object using `coef`, the estimated coefficients output from the DPD coefficient estimator, as for the coefficient matrix.

```
dpd = comm.DPD('PolynomialType','Memory polynomial', ...
    'Coefficients',coef);
```

Generate 2000 random symbols and apply 16-QAM modulation to the signal. Apply raised cosine transmit filtering to the modulated signal.

```
s = randi([0,15],2000,1);
u = qammod(s,16);
x = rctFilt(u);
```

Apply digital predistortion to the data. The DPD System object returns a predistorted signal to provide as input to the power amplifier.

```
y = dpd(x);
```

Format of Coefficient Matrix for Digital Predistortion Memory Polynomial

This examples shows the format of the coefficient matrix for the DPD memory polynomial by using a randomly generated coefficient matrix. The example:

- Creates a digital predistortion System object configured using a memory polynomial coefficient matrix with the memory depth set to 3 and the polynomial degree set to 5 consisting of random values.
- Predistorts a signal using the memory-polynomial coefficient matrix.
- Compares one predistorted output element to the corresponding input element that has been manually computed using the memory-polynomial coefficient matrix.

Create a coefficient matrix representing a predistorter with the output equal to the input by generating a 3-by-5 coefficient matrix of zeros and setting the `coef(1,1)` element to `1`. Add small random complex nonlinear terms to the coefficient matrix.

```
coef = zeros(3,5);
coef(1,1) = 1;
coef = coef + 0.01*(randn(3,5)+1j*randn(3,5));
```

Create a DPD System object using the memory polynomial coefficient matrix, `coef`.

```
dpd = comm.DPD('PolynomialType','Memory polynomial','Coefficients',coef);
```

Generate an input signal and predistort it using the `dpd` System object.

```
x = randn(20,1) + 1j*randn(20,1);
y = dpd(x);
```

Compare the manually distorted output for an input corresponding output element `y(18)` to show how the coefficient matrix is used to calculate that particular output value.

```
u = x(18:-1:(18-3+1));
isequal(y(18),sum(sum(coef.*[u u.*abs(u) u.*(abs(u).^2) u.*(abs(u).^3) u.*(abs(u).^4)])))
```

```
ans = logical
   1
```

Format of Cross-Term Coefficient Matrix for Digital Predistortion Memory Polynomial

This examples shows the format of the coefficient matrix for the DPD memory polynomial by using a randomly generated coefficient matrix. The example:

- Creates a digital predistorter System object configured using a cross-term memory polynomial coefficient matrix with the memory depth set to 3 and the polynomial degree set to 5 consisting of random values.
- Predistorts a signal using the cross-term memory polynomial coefficient matrix.
- Compares one predistorted output element to the corresponding input element that has been manually computed using the cross-term memory polynomial coefficient matrix.

Create a coefficient matrix representing a predistorter with the output equal to the input by generating a 3-by-5 coefficient matrix of zeros and setting the `coef(1,1)` element to `1`. Add small random complex nonlinear terms to the coefficient matrix.

```
coef = zeros(3,3*(5-1)+1);
coef(1,1) = 1;
coef = coef + 0.01*(randn(3,13) + 1j*randn(3,13));
```

Create a DPD System object using the cross-term memory polynomial coefficient matrix, `coef`.

```
dpd = comm.DPD('PolynomialType','Cross-term memory polynomial','Coefficients',coef);
```

Generate an input signal and predistort it using the `dpd` System object.

```
x = randn(20,1) + 1j*randn(20,1);
y = dpd(x);
```

Compare the manually distorted output for an input corresponding output element `y(18)` to show how the coefficient matrix is used to calculate that particular output value.

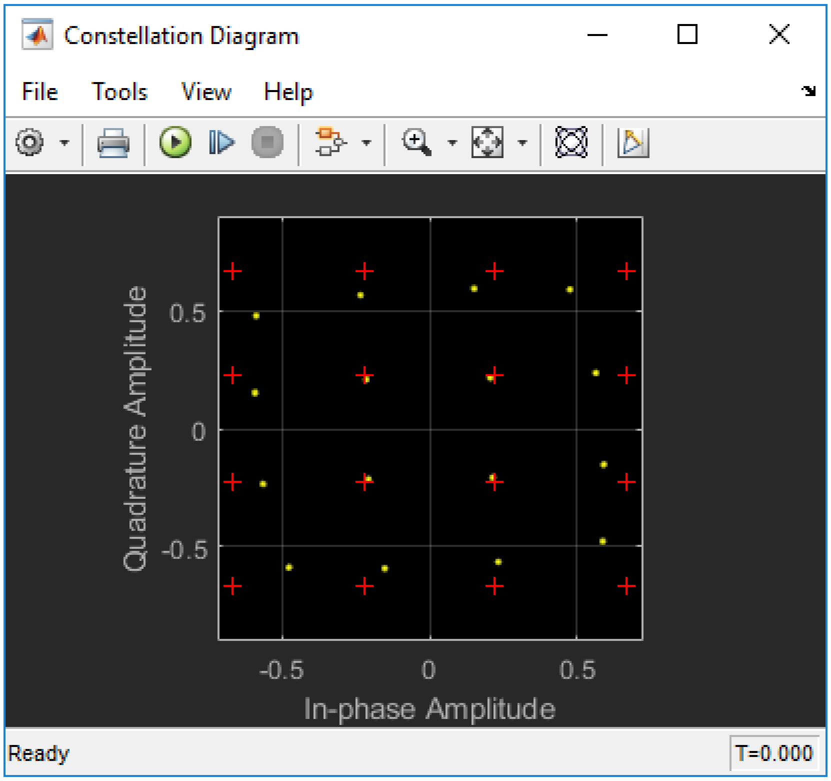
```
u = x(18:-1:(18-3+1));
isequal(y(18),sum(sum(coef.*[u u*abs(u.') u*(abs(u.').^2) u*(abs(u.').^3) u*(abs(u.').^4)])))
```

```
ans = logical
   1
```

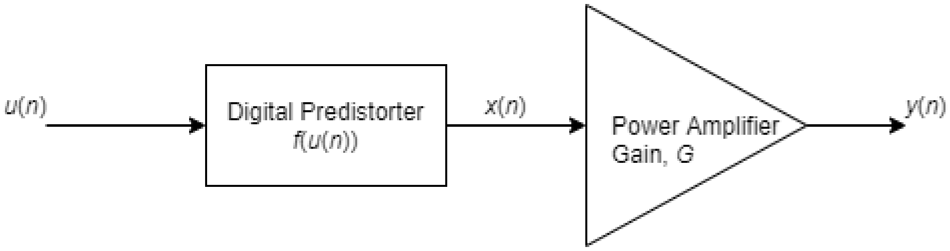
More About

Digital Predistortion

Wireless communication transmissions commonly require wide bandwidth signal transmission over a wide signal dynamic range. To transmit signals over a wide dynamic range and achieve high efficiency, RF power amplifiers (PAs) commonly operate in their nonlinear region. As this constellation diagram shows, the nonlinear behavior of a PA causes signal constellation distortions that pinch the amplitude (AM-AM distortion) and twist phase (AM-PM distortion) of constellation points proportional to the amplitude of the constellation point.



The goal of digital predistortion is to find a nonlinear function that linearizes the net effect of the PA nonlinear behavior at the PA output across the PA operating range. When the PA input is $x(n)$, and the predistortion function is $f(u(n))$, where $u(n)$ is the true signal to be amplified, the PA output is approximately equal to $G{\times}u(n)$, where G is the desired amplitude gain of the PA.



The digital predistorter can be configured to use a memory polynomial with or without cross terms.

- The memory polynomial with cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1} c_m \times u(n\text{-}m) + \sum_{m=0}^{M-1} \sum_{j=0}^{M-1} \sum_{k=0}^{K-1} a_{mjk} \times u(n\text{-}m) \times \left|u(n\text{-}j)\right|^k .$$

  The memory polynomial with cross terms has $(M{+}M{\times}M{\times}(K\text{-}1))$ coefficients for c_m and a_{mjk}.
- The memory polynomial without cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1} \sum_{k=0}^{K-1} a_{mk} \times u(n\text{-}m) \times \left|u(n\text{-}m)\right|^k .$$

The polynomial without cross terms has $M \times K$ coefficients for a_{mk}.

Estimating Predistortion Function and Coefficients

The DPD coefficient estimation uses an indirect learning architecture to find function $f(u(n))$ to predistort input signal $u(n)$ which precedes the PA input.

The DPD coefficient estimation algorithm models nonlinear PA memory effects based on the work in reference papers by Morgan, et al [1], and by Schetzen [2], using the theoretical foundation developed for Volterra systems.

Specifically, the inverse mapping from the PA output normalized by the PA gain, $\{y(n)/G\}$, to the PA input, $\{x(n)\}$, provides a good approximation to the function $f(u(n))$, needed to predistort $\{u(n)\}$ to produce $\{x(n)\}$.

Referring to the memory polynomial equations above, estimates are computed for the memory-polynomial coefficients:

- c_m and a_{mjk} for a memory polynomial with cross terms
- a_{mk} for a memory polynomial without cross terms

The memory-polynomial coefficients are estimated by using a least squares fit algorithm or a recursive least squares algorithm. The least squares fit algorithm or a recursive least squares algorithms use the memory polynomial equations above for a memory polynomial with or without cross terms, by replacing $\{u(n)\}$ with $\{y(n)/G\}$. The function order and dimension of the coefficient matrix are defined by the degree and depth of the memory polynomial.

For an example that details the process of accurately estimating memory-polynomial coefficients and predistorting a PA input signal, see “Digital Predistortion to Compensate for Power Amplifier Nonlinearities”.

For background reference material, see the works listed in [1] and [2].

References

[1] Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A Generalized Memory Polynomial Model for Digital Predistortion of Power Amplifiers." *IEEE Transactions on Signal Processing*. Vol. 54, Number 10, October 2006, pp. 3852–3860.

[2] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems.* New York: Wiley, 1980.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.DPDCoefficientEstimator`

Blocks
DPD

Topics
"Digital Predistortion to Compensate for Power Amplifier Nonlinearities"

Introduced in R2019a

comm.DPDCoefficientEstimator

Package: `comm`

Estimate memory-polynomial coefficients for digital predistortion

Description

The `comm.DPDCoefficientEstimator` System object estimates the coefficients of a memory polynomial for digital pre-distortion (DPD) of a nonlinear power amplifier, given the baseband equivalent input and baseband equivalent output of the power amplifier. For more information, see "Digital Predistortion" on page 3-444.

To compute predistortion coefficients:

1. Create the `comm.DPDCoefficientEstimator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
estimator = comm.DPDCoefficientEstimator
estimator = comm.DPDCoefficientEstimator(Name,Value)
```

Description

`estimator = comm.DPDCoefficientEstimator` creates a digital predistortion coefficient estimator System object to estimate the coefficients of a memory polynomial for digital predistortion (DPD) of a nonlinear power amplifier.

`estimator = comm.DPDCoefficientEstimator(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.DPDCoefficientEstimator('PolynomialType','Cross-term memory polynomial')` configures the predistortion coefficient estimator System object to estimate the coefficients for a memory-polynomial with cross terms. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

DesiredAmplitudeGaindB — Desired amplitude gain
10 (default) | scalar

Desired amplitude gain in dB, specified as a scalar. This property value expresses the desired signal gain at the compensated amplifier output.

Tunable: Yes

Data Types: double

PolynomialType — Polynomial type
'Memory polynomial' (default) | 'Cross-term memory polynomial'

Polynomial type used for predistortion, specified as one of these values:

- 'Memory polynomial' — Computes predistortion coefficients by using a memory polynomial without cross terms
- 'Cross-term memory polynomial' — Computes predistortion coefficients by using a memory polynomial with cross terms

For more information, see “Digital Predistortion” on page 3-444.

Degree — Memory-polynomial degree
5 (default) | positive integer

Memory-polynomial degree, specified as a positive integer.

Data Types: double

MemoryDepth — Memory-polynomial depth
3 (default) | positive integer

Memory-polynomial depth in samples, specified as a positive integer.

Data Types: double

Algorithm — Estimation algorithm
'Least squares' (default) | 'Recursive least squares'

Adaptive algorithm used for equalization, specified as one of these values:

- 'Least squares' — Estimate the memory polynomial coefficients by using a least squares algorithm
- 'Recursive least squares' — Estimate the memory polynomial coefficients by using a recursive least squares algorithm

For algorithm reference material, see the works listed in [1] and [2].

Data Types: char | string

ForgettingFactor — Forgetting factor
0.99 (default) | scalar in the range (0, 1]

Forgetting factor used by the recursive least squares algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the convergence time but causes the output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this property, set Algorithm to `'Recursive least squares'`.

Data Types: `double`

`InitialCoefficientEstimate` — Initial coefficient estimate

`[]` (default) | matrix

Initial coefficient estimate for the recursive least squares algorithm, specified as a matrix.

- If `InitialCoefficientEstimate` is an empty matrix, the initial coefficient estimate for the recursive least squares algorithm is chosen automatically to correspond to a memory polynomial that is an identity function, so that the output is equal to input.
- If `InitialCoefficientEstimate` is a nonempty matrix, the number of rows must be equal to `MemoryDepth`.

  - If `PolynomialType` is `'Memory polynomial'`, the number of columns is the degree of the memory polynomial.
  - If `PolynomialType` is `'Cross-term memory polynomial'`, the number of columns must equal $m(n\text{-}1)+1$. m is the memory depth of the polynomial, and n is the degree of the memory polynomial.

For more information, see “Digital Predistortion” on page 3-444.

Dependencies

To enable this property, set Algorithm to `'Recursive least squares'`.

Data Types: `double`
Complex Number Support: Yes

Usage

Syntax

```
coef = estimator(paIn,paOut)
```

Description

`coef = estimator(paIn,paOut)` estimates the coefficients of a memory polynomial for use by the `comm.DPD` System object to predistort a complex baseband signal by using a memory-polynomial to compensate for nonlinearities in a power amplifier.

Input Arguments

`paIn` — Power amplifier baseband equivalent input

column vector

Power amplifier baseband equivalent input, specified as a column vector.

Data Types: `double`
Complex Number Support: Yes

paOut — Power amplifier baseband equivalent output

column vector

Power amplifier baseband equivalent output, specified as a column vector of the same length as `paIn`.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

coef — Memory-polynomial coefficients

matrix

Memory-polynomial coefficients, returned as a matrix. For more information, see “Digital Predistortion” on page 3-444.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Predistort Power Amplifier Input Signal

Apply digital predistortion (DPD) to a power amplifier input signal. The DPD coefficient estimator System object uses a captured signal containing power amplifier input and output signals to determine the predistortion coefficient matrix.

Load a file containing the input and output signals for the power amplifier.

```
load('commpowamp_dpd_data.mat','PA_input','PA_output')
```

Generate a DPD coefficient estimator System object and a raised cosine transmit filter System object.

```
estimator = comm.DPDCoefficientEstimator( ...
    'DesiredAmplitudeGaindB',10, ...
    'PolynomialType','Memory polynomial', ...
    'Degree',5,'MemoryDepth',3,'Algorithm','Least squares');

rctFilt = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',2);
```

Estimate the digital predistortion memory-polynomial coefficients.

```
coef = estimator(PA_input,PA_output);
```

Generate a DPD System object using `coef`, the estimated coefficients output from the DPD coefficient estimator, as for the coefficient matrix.

```
dpd = comm.DPD('PolynomialType','Memory polynomial', ...
    'Coefficients',coef);
```

Generate 2000 random symbols and apply 16-QAM modulation to the signal. Apply raised cosine transmit filtering to the modulated signal.

```
s = randi([0,15],2000,1);
u = qammod(s,16);
x = rctFilt(u);
```

Apply digital predistortion to the data. The DPD System object returns a predistorted signal to provide as input to the power amplifier.

```
y = dpd(x);
```

Format of Coefficient Matrix for Digital Predistortion Memory Polynomial

This examples shows the format of the coefficient matrix for the DPD memory polynomial by using a randomly generated coefficient matrix. The example:

- Creates a digital predistortion System object configured using a memory polynomial coefficient matrix with the memory depth set to `3` and the polynomial degree set to `5` consisting of random values.
- Predistorts a signal using the memory-polynomial coefficient matrix.
- Compares one predistorted output element to the corresponding input element that has been manually computed using the memory-polynomial coefficient matrix.

Create a coefficient matrix representing a predistorter with the output equal to the input by generating a 3-by-5 coefficient matrix of zeros and setting the `coef(1,1)` element to `1`. Add small random complex nonlinear terms to the coefficient matrix.

```
coef = zeros(3,5);
coef(1,1) = 1;
coef = coef + 0.01*(randn(3,5)+1j*randn(3,5));
```

Create a DPD System object using the memory polynomial coefficient matrix, `coef`.

```
dpd = comm.DPD('PolynomialType','Memory polynomial','Coefficients',coef);
```

Generate an input signal and predistort it using the `dpd` System object.

```
x = randn(20,1) + 1j*randn(20,1);
y = dpd(x);
```

Compare the manually distorted output for an input corresponding output element `y(18)` to show how the coefficient matrix is used to calculate that particular output value.

```
u = x(18:-1:(18-3+1));
isequal(y(18),sum(sum(coef.*[u u.*abs(u) u.*(abs(u).^2) u.*(abs(u).^3) u.*(abs(u).^4)])))
```

```
ans = logical
   1
```

Format of Cross-Term Coefficient Matrix for Digital Predistortion Memory Polynomial

This examples shows the format of the coefficient matrix for the DPD memory polynomial by using a randomly generated coefficient matrix. The example:

- Creates a digital predistorter System object configured using a cross-term memory polynomial coefficient matrix with the memory depth set to 3 and the polynomial degree set to 5 consisting of random values.
- Predistorts a signal using the cross-term memory polynomial coefficient matrix.
- Compares one predistorted output element to the corresponding input element that has been manually computed using the cross-term memory polynomial coefficient matrix.

Create a coefficient matrix representing a predistorter with the output equal to the input by generating a 3-by-5 coefficient matrix of zeros and setting the `coef(1,1)` element to 1. Add small random complex nonlinear terms to the coefficient matrix.

```
coef = zeros(3,3*(5-1)+1);
coef(1,1) = 1;
coef = coef + 0.01*(randn(3,13) + 1j*randn(3,13));
```

Create a DPD System object using the cross-term memory polynomial coefficient matrix, `coef`.

```
dpd = comm.DPD('PolynomialType','Cross-term memory polynomial','Coefficients',coef);
```

Generate an input signal and predistort it using the `dpd` System object.

```
x = randn(20,1) + 1j*randn(20,1);
y = dpd(x);
```

Compare the manually distorted output for an input corresponding output element `y(18)` to show how the coefficient matrix is used to calculate that particular output value.

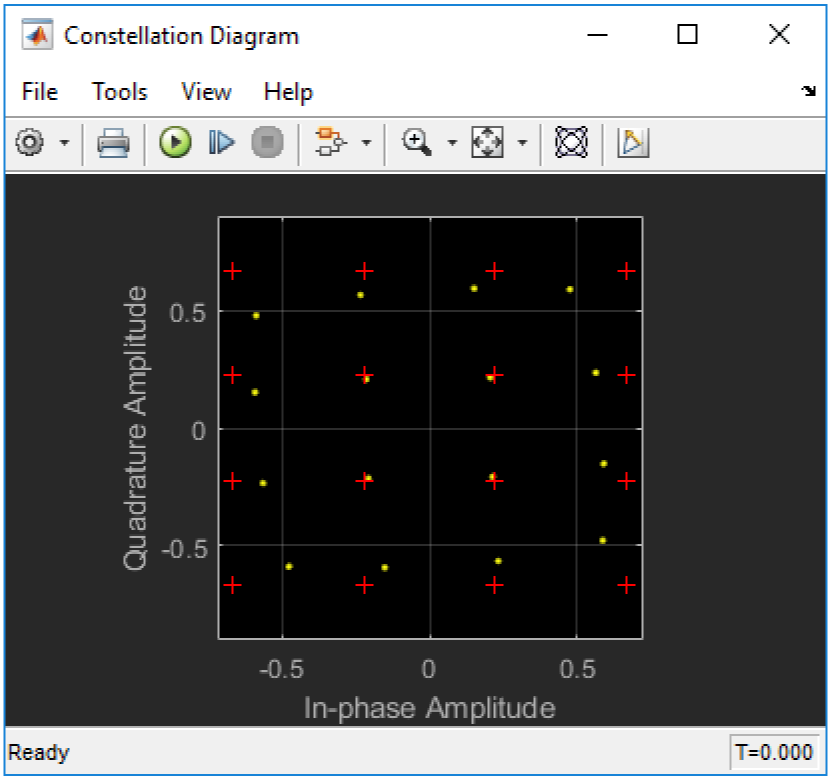
```
u = x(18:-1:(18-3+1));
isequal(y(18),sum(sum(coef.*[u u*abs(u.') u*(abs(u.').^2) u*(abs(u.').^3) u*(abs(u.').^4)])))
```

```
ans = logical
   1
```

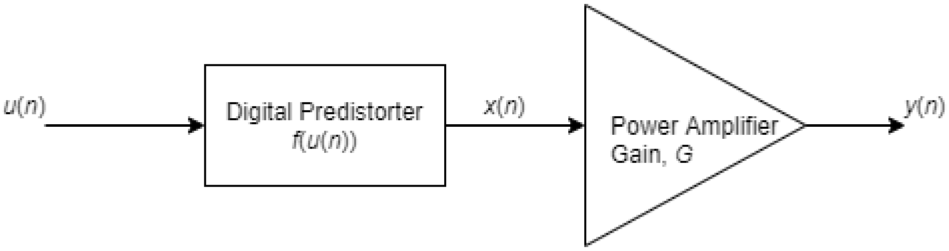
More About

Digital Predistortion

Wireless communication transmissions commonly require wide bandwidth signal transmission over a wide signal dynamic range. To transmit signals over a wide dynamic range and achieve high efficiency, RF power amplifiers (PAs) commonly operate in their nonlinear region. As this constellation diagram shows, the nonlinear behavior of a PA causes signal constellation distortions that pinch the amplitude (AM-AM distortion) and twist phase (AM-PM distortion) of constellation points proportional to the amplitude of the constellation point.



The goal of digital predistortion is to find a nonlinear function that linearizes the net effect of the PA nonlinear behavior at the PA output across the PA operating range. When the PA input is $x(n)$, and the predistortion function is $f(u(n))$, where $u(n)$ is the true signal to be amplified, the PA output is approximately equal to $G \times u(n)$, where G is the desired amplitude gain of the PA.



The digital predistorter can be configured to use a memory polynomial with or without cross terms.

- The memory polynomial with cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1} c_m \times u(n\text{-}m) + \sum_{m=0}^{M-1} \sum_{j=0}^{M-1} \sum_{k=0}^{K-1} a_{mjk} \times u(n\text{-}m) \times \left|u(n\text{-}j)\right|^{k}.$$

  The memory polynomial with cross terms has $(M{+}M{\times}M{\times}(K\text{-}1))$ coefficients for c_m and a_{mjk}.
- The memory polynomial without cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1}\sum_{k=0}^{K-1} a_{mk} \times u(n\text{-}m) \times |u(n\text{-}m)|^{k}.$$

The polynomial without cross terms has $M \times K$ coefficients for a_{mk}.

Estimating Predistortion Function and Coefficients

The DPD coefficient estimation uses an indirect learning architecture to find function *f*(u(n)) to predistort input signal $u(n)$ which precedes the PA input.

The DPD coefficient estimation algorithm models nonlinear PA memory effects based on the work in reference papers by Morgan, et al [1], and by Schetzen [2], using the theoretical foundation developed for Volterra systems.

Specifically, the inverse mapping from the PA output normalized by the PA gain, $\{y(n)/G\}$, to the PA input, $\{x(n)\}$, provides a good approximation to the function *f*(u(n)), needed to predistort $\{u(n)\}$ to produce $\{x(n)\}$.

Referring to the memory polynomial equations above, estimates are computed for the memory-polynomial coefficients:

- c_m and a_{mjk} for a memory polynomial with cross terms
- a_{mk} for a memory polynomial without cross terms

The memory-polynomial coefficients are estimated by using a least squares fit algorithm or a recursive least squares algorithm. The least squares fit algorithm or a recursive least squares algorithms use the memory polynomial equations above for a memory polynomial with or without cross terms, by replacing $\{u(n)\}$ with $\{y(n)/G\}$. The function order and dimension of the coefficient matrix are defined by the degree and depth of the memory polynomial.

For an example that details the process of accurately estimating memory-polynomial coefficients and predistorting a PA input signal, see “Digital Predistortion to Compensate for Power Amplifier Nonlinearities”.

For background reference material, see the works listed in [1] and [2].

References

[1] Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A Generalized Memory Polynomial Model for Digital Predistortion of Power Amplifiers." *IEEE Transactions on Signal Processing*. Vol. 54, Number 10, October 2006, pp. 3852–3860.

[2] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems.* New York: Wiley, 1980.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.DPD`

Blocks
DPD Coefficient Estimator

Topics
“Digital Predistortion to Compensate for Power Amplifier Nonlinearities”

Introduced in R2019a

comm.DPSKDemodulator

Package: `comm`

Demodulate using M-ary DPSK method

Description

The `DPSKDemodulator` object demodulates a signal that was modulated using the M-ary differential phase shift keying method. The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals. This object accepts a scalar-valued or column vector input signal.

To demodulate a signal that was modulated using differential phase shift keying:

1. Define and set up your DPSK modulator object. See “Construction” on page 3-448.
2. Call `step` to demodulate a signal according to the properties of `DPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.DPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the *M*-ary differential phase shift keying (M-DPSK) method.

`H = comm.DPSKDemodulator(Name,Value)` creates an M-DPSK demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.DPSKDemodulator(M,PHASE,Name,Value)` creates an M-DPSK demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `PhaseRotation` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `8`.

`PhaseRotation`

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is `pi/8`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`. When you set this property to `true` the `step` method outputs a column vector of bit values. The length of this column vector is equal to **log2**(`ModulationOrder` on page 3-0) times the number of demodulated symbols.

When you set this property to `false`, the `step` method outputs a column vector. The length of this column vector is equal to that of the input data vector. The output contains integer symbol values between `0` and `ModulationOrder-1`.

`SymbolMapping`

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer *m*, between (0 ≤ *m* ≤ `ModulationOrder`–1) maps to the current symbol. This mapping uses **exp**(*j*×`PhaseRotation` + *j*×2 × *π* ×m/ModulationOrder)×(previously modulated symbol).

`OutputDataType`

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`. When you set this property to `Full precision`, the input data type is single or double precision, the output data is the same as that of the input. When you set the `BitOutput` on page 3-0 property to true, `logical` data type becomes a valid option.

Methods

reset	Reset states of M-DPSK demodulator object
step	Demodulate using M-ary DPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

8-DPSK Signal in AWGN

Create a DPSK modulator and demodulator pair. Create an AWGN channel object having three bits per symbol.

```
dpskmod = comm.DPSKModulator(8,pi/8,'BitInput',true);
dpskdemod = comm.DPSKDemodulator(8,pi/8,'BitOutput',true);
channel = comm.AWGNChannel('EbNo',10,'BitsPerSymbol',3);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 3-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],150,1);
    modSig = dpskmod(txData);
    rxSig = channel(modSig);
    rxData = dpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0098
```

```
numErrors = errorStats(2)
```

```
numErrors = 147
```

```
numBits = errorStats(3)
```

```
numBits = 14999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-DPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DPSKModulator` | `comm.DQPSKDemodulator`

Introduced in R2012a

reset

System object: `comm.DPSKDemodulator`
Package: `comm`

Reset states of M-DPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DPSKDemodulator` object, `H`.

step

System object: `comm.DPSKDemodulator`
Package: `comm`

Demodulate using M-ary DPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the DPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a double or single precision data type scalar or column vector. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DPSKModulator

Package: comm

Modulate using M-ary DPSK method

Description

The DPSKModulator object modulates using the M-ary differential phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using differential phase shift keying:

1. Define and set up your DPSK modulator object. See “Construction” on page 3-454.
2. Call step to modulate a signal according to the properties of comm.DPSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.DPSKModulator creates a modulator System object, H. This object modulates the input signal using the M-ary differential phase shift keying (M-DPSK) method.

H = comm.DPSKModulator(Name,Value) creates an M-DPSK modulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.DPSKModulator(M,PHASE,Name,Value) creates an M-DPSK modulator object, H. This object has the ModulationOrder property set to M, the PhaseRotation property set to PHASE, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is 8.

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is pi/8. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to `true`, the `step` method input must be a column vector of bit values whose length is an integer multiple of **log2**(`ModulationOrder` on page 3-0). This vector contains bit representations of integers between `0` and `ModulationOrder–1`. When you set this property to `false`, the `step` method input requires a column vector of integer symbol values between `0` and `ModulationOrder–1`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) input bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer *m*, between (0 ≤ *m* ≤ `ModulationOrder`–1) shifts the output phase. This shift is (`PhaseRotation` on page 3-0 + 2 × *π* ×*m*/`ModulationOrder`) radians from the previous output phase. The output symbol uses **exp**(*j*×`PhaseRotation` + *j*×2 × *π* ×m/ModulationOrder)× (previously modulated symbol).

OutputDataType

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of M-DPSK modulator object
step	Modulate using M-ary DPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

8-DPSK Signal in AWGN

Create a DPSK modulator and demodulator pair. Create an AWGN channel object having three bits per symbol.

```
dpskmod = comm.DPSKModulator(8,pi/8,'BitInput',true);
dpskdemod = comm.DPSKDemodulator(8,pi/8,'BitOutput',true);
channel = comm.AWGNChannel('EbNo',10,'BitsPerSymbol',3);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 3-bit frames
- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],150,1);
    modSig = dpskmod(txData);
    rxSig = channel(modSig);
    rxData = dpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0098
```

```
numErrors = errorStats(2)
```

```
numErrors = 147
```

```
numBits = errorStats(3)
```

```
numBits = 14999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-DPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: `comm.DPSKModulator`
Package: `comm`

Reset states of M-DPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DPSKModulator` object, `H`.

step

System object: `comm.DPSKModulator`
Package: `comm`

Modulate using M-ary DPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the DPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric or logical data types.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DQPSKDemodulator

Package: comm

Demodulate using DQPSK method

Description

The DQPSKDemodulator object demodulates a signal that was modulated using the differential quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using differential quadrature phase shift keying:

1. Define and set up your DQPSK modulator object. See “Construction” on page 3-459.
2. Call step to demodulate a signal according to the properties of DQPSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.DQPSKDemodulator creates a demodulator System object, H. This object demodulates the input signal using the differential quadrature phase shift keying (DQPSK) method.

H = comm.DQPSKDemodulator(Name,Value) creates a DQPSK demodulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.DQPSKDemodulator(PHASE,Name,Value) creates a DQPSK demodulator object, H. This object has the PhaseRotation property set to PHASE and the other specified properties set to the specified values.

Properties

PhaseRotation

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar. The default is pi/4. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is false. When you set this property to true the step method outputs a column vector of bit values with length equal to twice the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector, of length equal to the input data vector, that contains integer symbol values between `0` and `3`.

`SymbolMapping`

Constellation encoding

Specify how the object maps an integer or group of 2 bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer *m*, between $0 \leq m \leq 3$ maps to the current symbol as $\mathbf{exp}(j \times$`PhaseRotation` on page 3-0 $+ j \times 2 \times \pi \times {}^{m}\!/_{4}) \times$ (previously modulated symbol).

`OutputDataType`

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`. When you set this property to `Full precision` the output has the same data type as that of the input. In this case, the input data type is single- or double-precision value. When you set the `BitOutput` on page 3-0 property to `true`, `logical` data type becomes a valid option.

Methods

reset	Reset states of DQPSK demodulator object
step	Demodulate using DQPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

DQPSK Signal in AWGN

Create a DQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
dqpskmod = comm.DQPSKModulator('BitInput',true);
dqpskdemod = comm.DQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',6,'BitsPerSymbol',2);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 2-bit frames

- 8-DPSK modulate
- Pass through AWGN channel
- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],100,1);
    modSig = dqpskmod(txData);
    rxSig = channel(modSig);
    rxData = dqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0170
```

```
numErrors = errorStats(2)
```

```
numErrors = 170
```

```
numBits = errorStats(3)
```

```
numBits = 9999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DQPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKDemodulator` | `comm.DPSKDemodulator` | `comm.DQPSKModulator`

Introduced in R2012a

reset

System object: `comm.DQPSKDemodulator`
Package: `comm`

Reset states of DQPSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DQPSKDemodulator` object, `H`.

step

System object: `comm.DQPSKDemodulator`
Package: `comm`

Demodulate using DQPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the DQPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a single or double precision data type scalar or column vector. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.DQPSKModulator

Package: `comm`

Modulate using DQPSK method

Description

The `DQPSKModulator` object modulates using the differential quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using differential quadrature phase shift keying:

1. Define and set up your DQPSK modulator object. See “Construction” on page 3-464.
2. Call `step` to modulate a signal according to the properties of `comm.DQPSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.DQPSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the differential quadrature phase shift keying (DQPSK) method.

`H = comm.DQPSKModulator(Name,Value)` creates a DQPSK modulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.DQPSKModulator(PHASE,Name,Value)` creates a DQPSK modulator object, `H`. This object has the `PhaseRotation` property set to `PHASE` and the other specified properties set to the specified values.

Properties

`PhaseRotation`

Additional phase shift

Specify the additional phase difference between previous and current modulated symbols in radians as a real scalar value. The default is `pi/4`. This value corresponds to the phase difference between previous and current modulated symbols when the input is zero.

`BitInput`

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`. When you set this property to true, the step method input must be a column vector of bit values. The length of this vector is an integer multiple of two. This vector contains bit representations of integers between `0` and `3`. When you set this property to `false`, the `step` method input must be a column vector of integer symbol values between `0` and `3`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of two input bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the input integer *m*, between $0 \leq m \leq 3$ shifts the output phase. This shift is (`PhaseRotation` on page 3-0 $+ 2 \times \pi \times {}^{m}\!/_{4}$) radians from the previous output phase. The output symbol is $\mathbf{exp}(j \times$`PhaseRotation`$+ j \times 2 \times \pi \times {}^{m}\!/_{4}) \times$(previously modulated symbol).

OutputDataType

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of DQPSK modulator object
step	Modulate using DQPSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

DQPSK Signal in AWGN

Create a DQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
dqpskmod = comm.DQPSKModulator('BitInput',true);
dqpskdemod = comm.DQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',6,'BitsPerSymbol',2);
```

Create an error rate calculator. Set the `ComputationDelay` property to `1` to account for the one bit transient caused by the differential modulation

```
errorRate = comm.ErrorRate('ComputationDelay',1);
```

Main processing loop steps:

- Generate 50 2-bit frames
- 8-DPSK modulate
- Pass through AWGN channel

- 8-DPSK demodulate
- Collect error statistics

```
for counter = 1:100
    txData = randi([0 1],100,1);
    modSig = dqpskmod(txData);
    rxSig = channel(modSig);
    rxData = dqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 0.0170
```

```
numErrors = errorStats(2)
```

```
numErrors = 170
```

```
numBits = errorStats(3)
```

```
numBits = 9999
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the DQPSK Modulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DBPSKModulator` | `comm.DPSKModulator` | `comm.DQPSKDemodulator`

Introduced in R2012a

reset

System object: `comm.DQPSKModulator`
Package: `comm`

Reset states of DQPSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `DQPSKModulator` object, `H`.

step

System object: `comm.DQPSKModulator`
Package: `comm`

Modulate using DQPSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the DQPSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric or logical data types.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.ErrorRate

Package: `comm`

Compute bit or symbol error rate of input data

Description

The `ErrorRate` object compares input data from a transmitter with input data from a receiver and calculates the error rate as a running statistic. To obtain the error rate, the object divides the total number of unequal pairs of data elements by the total number of input data elements from one source.

To obtain the error rate:

1. Define and set up your error rate object. See “Construction” on page 3-469.
2. Call `step` to compare input data from a transmitter with input data from a receiver and calculate the error rate according to the properties of `comm.ErrorRate`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.ErrorRate` creates an error rate calculator System object, `H`. This object computes the error rate of the received data by comparing it to the transmitted data.

`H = comm.ErrorRate(Name,Value)` creates an error rate calculator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`ReceiveDelay`

Number of samples to delay transmitted signal

Specify the number of samples by which the received data lags behind the transmitted data. This value must be a real, nonnegative, double-precision, integer scalar. Use this property to align the samples for comparison in the transmitted and received input data vectors. Specify the delay in number of samples, regardless of whether the input is a scalar or a vector. The default is `0`.

`ComputationDelay`

Computation delay

Specify the number of data samples that the object should ignore at the beginning of the comparison. This value must be a real, nonnegative, double-precision, integer scalar. Use this property to ignore the transient behavior of both input signals. The default is `0`.

`Samples`

Samples to consider

Specify samples to consider as one of `Entire frame` | `Custom` | `Input port`. The property defines whether the object should consider all or only part of the input frames when computing error statistics. The default is `Entire frame`. Select `Entire frame` to compare all the samples of the RX frame to those of the TX frame. Select `Custom` or `Input port` to list the indices of the RX frame elements that the object should consider when making comparisons. When you set this property to `Custom`, you can list the indices as a scalar or a column vector of double-precision integers through the `CustomSamples` on page 3-0 property. When you set this property to `Input port`, you can list the indices as an input to the `step` method.

`CustomSamples`

Selected samples from frame

Specify a scalar or a column vector of double-precision, real, positive integers. This value lists the indices of the elements of the RX frame vector that the object uses when making comparisons. This property applies when you set the `Samples` on page 3-0 property to `Custom`. The default is an empty vector, which specifies that all samples are used.

`ResetInputPort`

Enable error rate reset input

Set this property to `true` to reset the error statistics via an input to the `step` method. The default is `false`.

Methods

reset	Reset states of error rate calculator object
step	Compute bit or symbol error rate of input data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Calculate Error Statistics

Create two binary vectors and determine the error statistics.

Create a bit error rate counter object.

```
errorRate = comm.ErrorRate;
```

Create an arbitrary binary data vector.

```
x = [1 0 1 0 1 0 1 0 1 0]';
```

Introduce errors to the first and last bits.

```
y = x;
y(1) = ~y(1);
y(end) = ~y(end);
```

Calculate the error statistics.

```
z = errorRate(x,y);
```

The first element of the vector z is the bit error rate.

```
z(1)
```

```
ans = 0.2000
```

The second element of z is the total error count.

```
z(2)
```

```
ans = 2
```

The third element of z is the total number of bits.

```
z(3)
```

```
ans = 10
```

Calculate BER between Transmitted and Received Signal

Create an 8-DPSK modulator and demodulator pair that work with binary data.

```
dpskModulator = comm.DPSKModulator('ModulationOrder',8,'BitInput',true);
dpskDemodulator = comm.DPSKDemodulator('ModulationOrder',8,'BitOutput',true);
```

Create an error rate calculator, accounting for the three bit (one symbol) transient caused by the differential modulation.

```
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Calculate the BER for 10 frames.

```
BER = zeros(10,1);

for i= 1:10
    txData = randi([0 1],96,1);           % Generate binary data
    modData = dpskModulator(txData);      % Modulate
    rxSig = awgn(modData,7);              % Pass through AWGN channel
    rxData = dpskDemodulator(rxSig);      % Demodulate
    errors = errorRate(txData,rxData);    % Compute error statistics
    BER(i) = errors(1);                   % Save BER data
end
```

Display the BER.

```
BER
```

```
BER = 10×1

    0.1613
    0.1640
    0.1614
    0.1496
    0.1488
    0.1309
    0.1405
    0.1399
    0.1370
    0.1411
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Error Rate Calculation block reference page. The object properties correspond to the block parameters, except:

- The **Output data** and **Variable name** block parameters do not have a corresponding properties. The object always returns the result as an output.
- The **Stop simulation** block parameter does not have a corresponding property. To implement similar behavior, use the output of the `step` method in a while loop, to programmatically stop the simulation. .
- The **Computation mode** parameter corresponds to the `Samples` on page 3-0 and `CustomSamples` on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`alignsignals` | `finddelay`

Introduced in R2012a

reset

System object: `comm.ErrorRate`
Package: `comm`

Reset states of error rate calculator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ErrorRate` object, `H`.

step

System object: comm.ErrorRate
Package: comm

Compute bit or symbol error rate of input data

Syntax

```
Y = step(H,TX,RX)
Y = step(H,TX,RX,SEL)
Y = step(H,TX,RX,RST)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,TX,RX) counts the number of differences between the transmitted data vector, TX, and received data vector, RX. The step method outputs a three-element vector consisting of the error rate, followed by the number of errors detected and the total number of samples compared. TX and RX inputs can be either scalars or column vectors of the same data type. Valid data types are single, double, integer or logical. If TX is a scalar and RX is a vector, or vice-versa, then the block compares the scalar with each element of the vector.

Y = step(H,TX,RX,SEL) calculates the errors based on selected samples from the input frame specified by the SEL input. SEL must be a real, double-precision integer-valued scalar or a column vector. The vector lists the indices of the elements of the RX input vector that the object should consider when making comparisons. This syntax applies when you set the Samples property to 'Input Port'.

Y = step(H,TX,RX,RST) resets the error count whenever the input RST is non-zero. RST must be a real, double, or logical scalar. When you set the RST input to a nonzero value, the object clears its error statistics and then recomputes them based on the current TX and RX inputs. This syntax applies when you set the ResetInputPort property to true. You can combine optional input arguments when their enabling properties are set. Optional inputs must be listed in the same order as the order of the enabling properties.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.LinearEqualizer

Package: `comm`

Equalize modulated signals using linear filtering

Description

The `comm.LinearEqualizer` System object uses a linear filter tap delay line with a weighted sum to equalize modulated signals transmitted through a dispersive channel. The equalizer object adaptively adjusts tap weights based on the selected algorithm. For more information, see “Algorithms” on page 3-509.

To equalize modulated signals using a linear filter:

1. Create the `comm.LinearEqualizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
lineq = comm.LinearEqualizer
lineq = comm.LinearEqualizer(Name,Value)
```

Description

`lineq = comm.LinearEqualizer` creates a linear equalizer System object to adaptively equalize a signal.

`lineq = comm.LinearEqualizer(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.LinearEqualizer('Algorithm','RLS')` configures the equalizer object to update tap weights using the recursive least squares (RLS) algorithm. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Algorithm` — Adaptive algorithm
`'LMS'` (default) | `'RLS'` | `'CMA'`

Adaptive algorithm used for equalization, specified as one of these values:

- `'LMS'` — Update the equalizer tap weights using the “Least Mean Square (LMS) Algorithm” on page 3-510.
- `'RLS'` — Update the equalizer tap weights using the “Recursive Least Square (RLS) Algorithm” on page 3-511.
- `'CMA'` — Update the equalizer tap weights using the “Constant Modulus Algorithm (CMA)” on page 3-511.

Data Types: `char` | `string`

`NumTaps` — Number of equalizer taps

`5` (default) | positive integer

Number of equalizer taps, specified as a positive integer. The number of equalizer taps must be greater than or equal to the value of the `InputSamplesPerSymbol` property.

Data Types: `double`

`StepSize` — Step size

`0.01` (default) | positive scalar

Step size used by the adaptive algorithm, specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tip To determine the maximum step size allowed, use the `maxstep` object function.

Tunable: Yes

Dependencies

To enable this property, set Algorithm to `'LMS'` or `'CMA'`.

Data Types: `double`

`ForgettingFactor` — Forgetting factor

`0.99` (default) | scalar in the range (0, 1]

Forgetting factor used by the adaptive algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this property, set Algorithm to `'RLS'`.

Data Types: `double`

`InitialInverseCorrelationMatrix` — Initial inverse correlation matrix

`0.1` (default) | scalar | matrix

Initial inverse correlation matrix, specified as a scalar or an N_{Taps}-by-N_{Taps} matrix. N_{Taps} is equal to the NumTaps property value. If you specify `InitialInverseCorrelationMatrix` as a scalar, *a*, the equalizer sets the initial inverse correlation matrix to *a* times the identity matrix: a(`eye`(N_{Taps})).

Tunable: Yes

Dependencies

To enable this property, set Algorithm to `'RLS'`.

Data Types: `double`

`Constellation` — Signal constellation

`pskmod(0:3,4,pi/4)` (default) | vector

Signal constellation, specified as a vector. The default value is a QPSK constellation generated using this code: `pskmod(0:3,4,pi/4)`.

Tunable: Yes

Data Types: `double`

`ReferenceTap` — Reference tap

`3` (default) | positive integer

Reference tap, specified as a positive integer less than or equal to the NumTaps property value. The equalizer uses the reference tap location to track the main energy of the channel.

Data Types: `double`

`InputDelay` — Input signal delay

`0` (default) | nonnegative integer

Input signal delay in samples relative to the reset time of the equalizer, specified as a nonnegative integer. If the input signal is a vector of length greater than 1, then the input delay is relative to the start of the input vector. If the input signal is a scalar, then the input delay is relative to the first call of the System object and to the first call of the System object after calling the `release` or `reset` object function.

Data Types: `double`

`InputSamplesPerSymbol` — Number of input samples per symbol

`1` (default) | positive integer

Number of input samples per symbol, specified as a positive integer. Setting this property to any number greater than one effectively creates a fractionally spaced equalizer. For more information, see “Symbol Tap Spacing” on page 3-509.

Data Types: `double`

`TrainingFlagInputPort` — Enable training control input

`false` (default) | `true`

Enable training control input, specified as `false` or `true`. Setting this property to `true` enables the equalizer training flag input tf.

Tunable: Yes

Data Types: `logical`

`AdaptAfterTraining` — Update tap weights when not training

`true` (default) | `false`

Update tap weights when not training, specified as `true` or `false`. If this property is set to `true`, the System object uses decision directed mode to update equalizer tap weights. If this property is set to `false`, the System object keeps the equalizer tap weights unchanged after training.

Tunable: Yes

Data Types: `logical`

`AdaptWeightsSource` — Source of adapt tap weights request

`'Property'` (default) | `'Input port'`

Source of adapt tap weights request, specified as one of these values:

- `'Property'` — Specify this value to use the AdaptWeights property to control when the System object adapts tap weights.
- `'Input port'` — Specify this value to use the `aw` input to control when the System object adapts tap weights.

Dependencies

To enable this property, set Algorithm to `'CMA'`.

Data Types: `char` | `string`

`AdaptWeights` — Adapt tap weights

`true` (default) | `false`

Adapt tap weights, specified as `true` or `false`. If this property is set to `true`, the System object updates the equalizer tap weights. If this property is set to `false`, the System object keeps the equalizer tap weights unchanged.

Tunable: Yes

Dependencies

To enable this property, set AdaptWeightsSource to `'Property'` and set AdaptAfterTraining to `true`.

Data Types: `logical`

`InitialWeightsSource` — Source for initial tap weights

`'Auto'` (default) | `'Property'`

Source for initial tap weights, specified as

- `'Auto'` — Initialize the tap weights to the algorithm-specific default values, as described in the InitialWeights property.
- `'Property'` — Initialize the tap weights using the InitialWeights property value.

Data Types: `char` | `string`

`InitialWeights` — Initial tap weights

`0` or `[0;0;1;0;0]` (default) | scalar | column vector

Initial tap weights used by the adaptive algorithm, specified as a scalar or vector. The default is `0` when the Algorithm property is set to `'LMS'` or `'RLS'`. The default is `[0;0;1;0;0]` when the Algorithm property is set to `'CMA'`.

If you specify `InitialWeights` as a vector, the vector length must be equal to the NumTaps property value. If you specify `InitialWeights` as a scalar, the equalizer uses scalar expansion to create a vector of length NumTaps with all values set to `InitialWeights`.

Tunable: Yes

Dependencies

To enable this property, set InitialWeightsSource to `'Property'`.

Data Types: `double`

`WeightUpdatePeriod` — Tap weight update period

`1` (default) | positive integer

Tap weight update period in symbols, specified as a positive integer. The equalizer updates the tap weights after processing this number of symbols.

Data Types: `double`

Usage

Syntax

```
y = lineq(x,tsym)
y = lineq(x,tsym,tf)

y = lineq(x)
y = lineq(x,aw)
```

Description

`y = lineq(x,tsym)` equalizes input signal `x` by using training symbols `tsym`. The output is the equalized symbols. To enable this syntax, set the Algorithm property to `'LMS'` or `'RLS'`.

`y = lineq(x,tsym,tf)` also specifies training flag `tf`. The System object starts training when `tf` changes from `false` to `true` (at the rising edge). The System object trains until all symbols in `tsym` are processed. The input `tsym` is ignored when `tf` is `false`. To enable this syntax, set the Algorithm property to `'LMS'` or `'RLS'` and TrainingFlagInputPort property to `true`.

`y = lineq(x)` equalizes input signal `x`. To enable this syntax, set the Algorithm property to `'CMA'`.

`y = lineq(x,aw)` also specifies adapts weights flag `aw`. The System object adapts the equalizer tap weights when `aw` is `true`. If `aw` is `false`, the System object keeps the weights unchanged. To enable this syntax, set the Algorithm property to `'CMA'` and AdaptWeightsSource property to `'Input port'`.

`[y,err] = lineq( ___ )` also returns error signal `err` using input arguments from any of the previous syntaxes.

`[y,err,weights] = lineq( ___ )` also returns `weights`, the tap weights from the last tap weight update, using input arguments from any of the previous syntaxes.

Input Arguments

`x` — Input signal
column vector

Input signal, specified as a column vector. The input signal vector length must be equal to an integer multiple of the InputSamplesPerSymbol property value. For more information, see “Symbol Tap Spacing” on page 3-509.

Data Types: `double`
Complex Number Support: Yes

`tsym` — Training symbols
column vector

Training symbols, specified as a column vector of length less than or equal to the length of input `x`. The input `tsym` is ignored when `tf` is `false`.

Dependencies

To enable this argument, set the Algorithm property to `'LMS'` or `'RLS'`.

Data Types: `double`
Complex Number Support: Yes

`tf` — Training flag
`true` | `false`

Training flag, specified as `true` or `false`. The System object starts training when `tf` changes from `false` to `true` (at the rising edge). The System object trains until all symbols in `tsym` are processed. The input `tsym` is ignored when `tf` is `false`.

Dependencies

To enable this argument, set the Algorithm property to `'LMS'` or `'RLS'` and TrainingFlagInputPort property to `true`.

Data Types: `logical`

`aw` — Adapt weights flag
`true` | `false`

Adapt weights flag, specified as `true` or `false`. If `aw` is `true`, the System object adapts weights. If `aw` is `false`, the System object keeps the weights unchanged.

Dependencies

To enable this argument, set the Algorithm property to `'CMA'` and AdaptWeightsSource property to `'Input port'`.

Data Types: `logical`

Output Arguments

`y` — Equalized symbols
column vector

Equalized symbols, returned as a column vector that has the same length as input signal `x`.

`err` — Error signal
column vector

Error signal, returned as a column vector that has the same length as input signal `x`.

`weights` — Tap weights
column vector

Tap weights, returned as a column vector that has NumTaps elements. `weights` contains the tap weights from the last tap weight update.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.LinearEqualizer`

isLocked	Determine if System object is in use
clone	Create duplicate System object
info	Characteristic information about the equalizer object
maxstep	Maximum step size for LMS equalizer convergence
mmseweights	Linear equalizer MMSE tap weights

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Linearly Equalize BPSK-Modulated Signal

Create a BPSK modulator and an equalizer System object™, specifying a linear LMS equalizer having eight taps and a step size of 0.03.

```
bpsk = comm.BPSKModulator;
eqlms = comm.LinearEqualizer('Algorithm','LMS','NumTaps',8,'StepSize',0.03);
```

Change the reference tap index of the equalizer.

```
eqlms.ReferenceTap = 4;
```

Build a set of test data. Receive the data by convolving the signal.

```
x = bpsk(randi([0 1],1000,1));
rxsig = conv(x,[1 0.8 0.3]);
```

Use `maxstep` to find the maximum permitted step size.

```
mxStep = maxstep(eqlms,rxsig)
```

```
mxStep = 0.1384
```

Equalize the received signal. Use the first 200 symbols as the training sequence.

```
y = eqlms(rxsig,x(1:200));
```

Linearly Equalize QPSK-Modulated Signal

Apply linear equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through a multipath AWGN channel.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
chtaps = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
```

Generate QPSK-modulated symbols. Apply multipath channel filtering and AWGN impairments to the symbols.

```
data = randi([0 M-1],numSymbols,1);
tx = pskmod(data,M,pi/4);
rx = awgn(filter(chtaps,1,tx),25,'measured');
```

Create a linear equalizer System object and display the default configuration. Adjust the reference tap to 1. Check the maximum permitted step size. Equalize the impaired symbols.

```
eq = comm.LinearEqualizer
```

```
eq = 
  comm.LinearEqualizer with properties:

                Algorithm: 'LMS'
                  NumTaps: 5
                 StepSize: 0.0100
            Constellation: [1x4 double]
             ReferenceTap: 3
               InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: true
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

```
eq.ReferenceTap = 1;
```

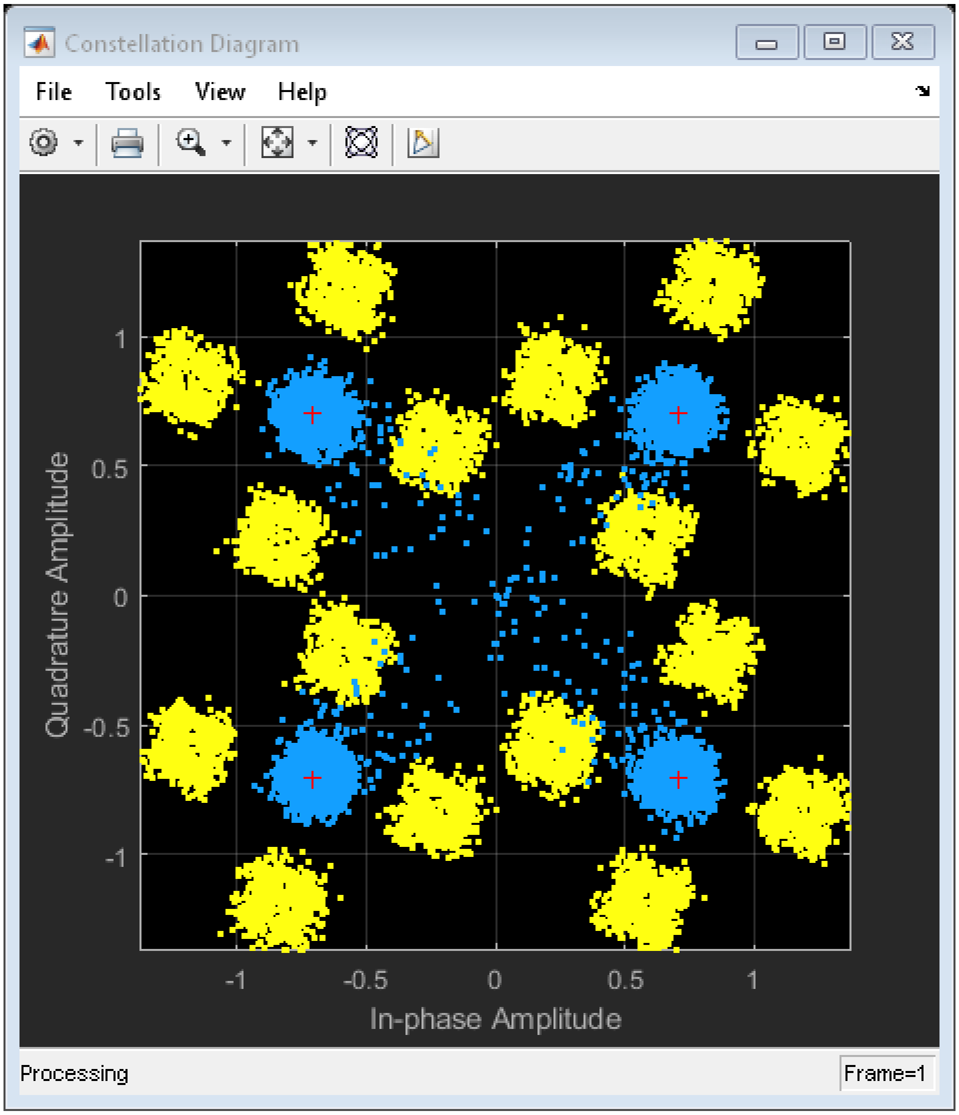
```
mxStep = maxstep(eq,rx)
```

```
mxStep = 0.3154
```

```
[y,err,weights] = eq(rx,tx(1:numTrainingSymbols));
```

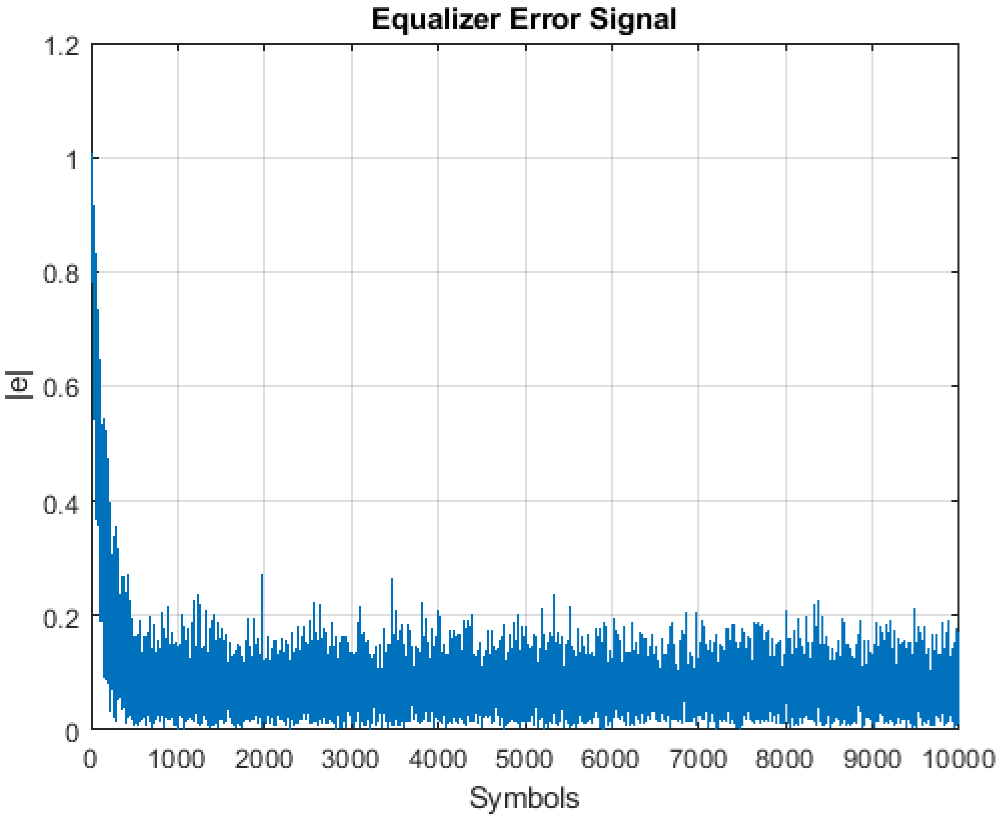
Plot the constellation of the impaired and equalized symbols.

```
constell = comm.ConstellationDiagram('NumInputPorts',2);
constell(rx,y)
```



Plot the equalizer error signal and compute the error vector magnitude (EVM) of the equalized symbols.

```
plot(abs(err))
grid on; xlabel('Symbols'); ylabel('|e|');title('Equalizer Error Signal')
```

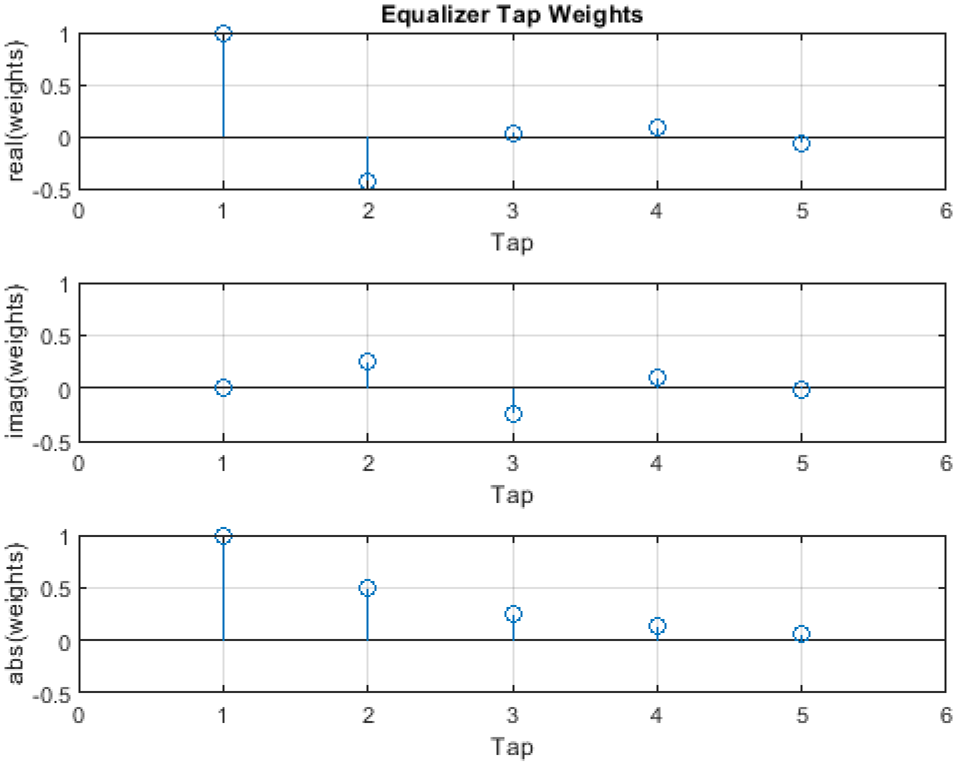


```
errevm = comm.EVM;
evm = errevm(tx,y)
```

```
evm = 11.7710
```

Plot the equalizer tap weights.

```
subplot(3,1,1);
stem(real(weights)); ylabel('real(weights)'); xlabel('Tap'); grid on; axis([0 6 -0.5 1])
title('Equalizer Tap Weights')
subplot(3,1,2);
stem(imag(weights)); ylabel('imag(weights)'); xlabel('Tap'); grid on; axis([0 6 -0.5 1])
subplot(3,1,3);
stem(abs(weights)); ylabel('abs(weights)'); xlabel('Tap'); grid on; axis([0 6 -0.5 1])
```



Linearly Equalize System By Using Different Training Schemes

Demonstrate linear equalization by using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel. Apply different equalizer training schemes and show the symbol error magnitude.

System Setup

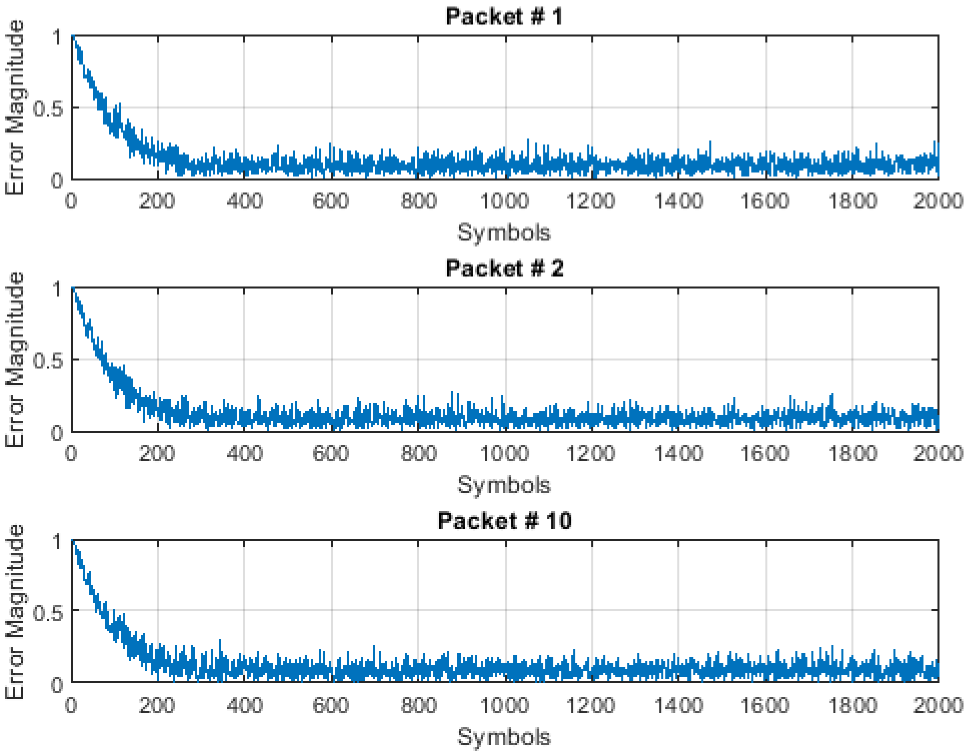
Simulate a QPSK-modulated system subject to AWGN. Transmit packets composed of 200 training symbols and 1800 random data symbols. Configure a linear LMS equalizer to recover the packet data.

```
M = 4;
numTrainSymbols = 200;
numDataSymbols = 1800;
SNR = 20;
trainingSymbols = pskmod(randi([0 M-1],numTrainSymbols,1),M,pi/4);
numPkts = 10;
lineq = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',5,'ReferenceTap',3,'StepSize',0.01);
```

Train the Equalizer at the Beginning of Each Packet With Reset

Use prepended training symbols when processing each packet. After processing each packet, reset the equalizer. This reset forces the equalizer to train the taps with no previous knowledge. Equalizer error signal plots for the first, second, and last packet show higher symbol errors at the start of each packet.

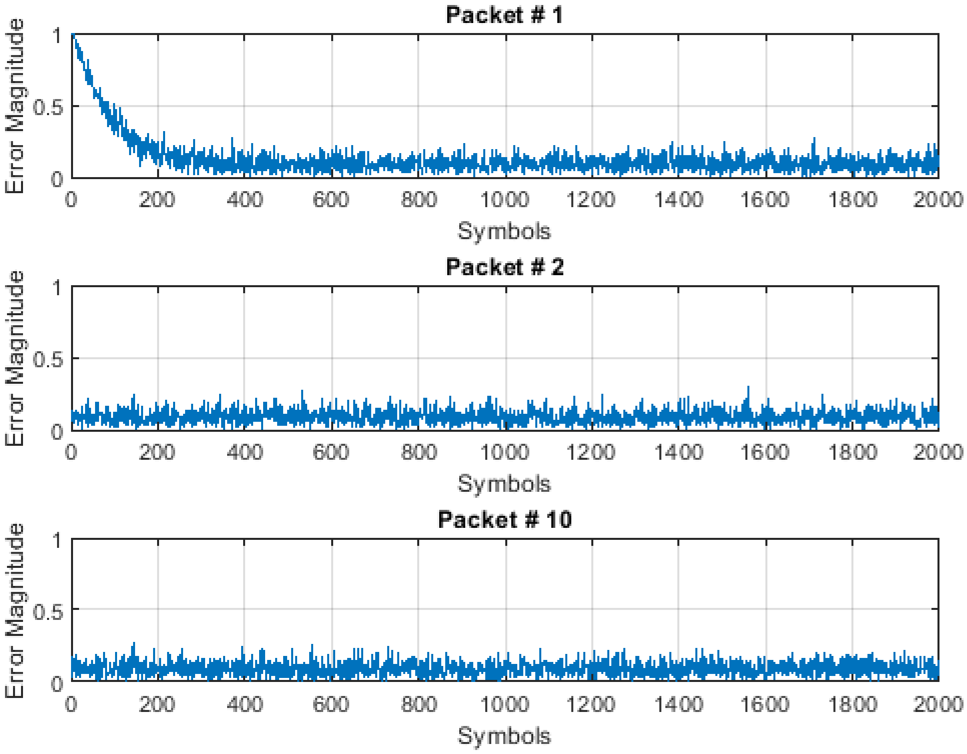
```
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    rx = awgn(packet,SNR);
    [~,err] = lineq(rx,trainingSymbols);
    reset(lineq)
    if (ii ==1 || ii == 2 ||ii == numPkts)
        subplot(3,1,jj)
        plot(abs(err))
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols')
        ylabel('Error Magnitude')
        axis([0,length(packet),0,1])
        grid on;
        jj = jj+1;
    end
end
```



Train the Equalizer at the Beginning of Each Packet Without Reset

Process each packet using prepended training symbols. Do not reset the equalizer after each packet is processed. By not resetting after each packet, the equalizer retains tap weights from training prior packets. Equalizer error signal plots for the first, second, and last packet show that after the initial training on the first packet, subsequent packets have fewer symbol errors at the start of each packet.

```
release(lineq)
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    channel = 1;
    rx = awgn(packet*channel,SNR);
    [~,err] = lineq(rx,trainingSymbols);
    if (ii ==1 || ii == 2 ||ii == numPkts)
        subplot(3,1,jj)
        plot(abs(err))
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols')
        ylabel('Error Magnitude')
        axis([0,length(packet),0,1])
        grid on;
        jj = jj+1;
    end
end
```



Train the Equalizer Periodically

Systems with signals subject to time-varying channels require periodic equalizer training to maintain lock on the channel variations. Specify a system that has 200 symbols of training for every 1800 data symbols. Between training, the equalizer does not update tap weights. The equalizer processes 200 symbols per packet.

```
Rs = 1e6;
fd = 20;
spp = 200; % Symbols per packet
b = randi([0 M-1],numDataSymbols,1);
dataSym = pskmod(b,M,pi/4);
packet = [trainingSymbols; dataSym];
stream = repmat(packet,10,1);
tx = (0:length(stream)-1)'/Rs;
channel = exp(1i*2*pi*fd*tx);
rx = awgn(stream.*channel,SNR);
```

Set the `AdaptAfterTraining` property to `false` to stop the equalizer tap weight updates after the training phase.

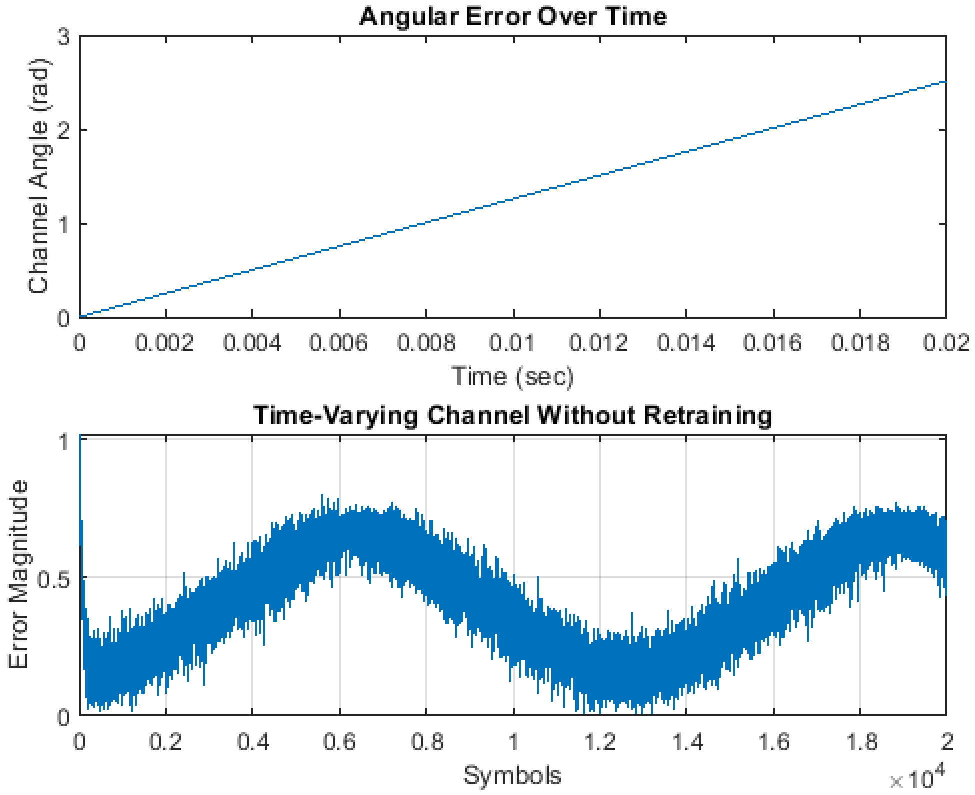
```
release(lineq)
lineq.AdaptAfterTraining = false
```

```
lineq = 
  comm.LinearEqualizer with properties:
```

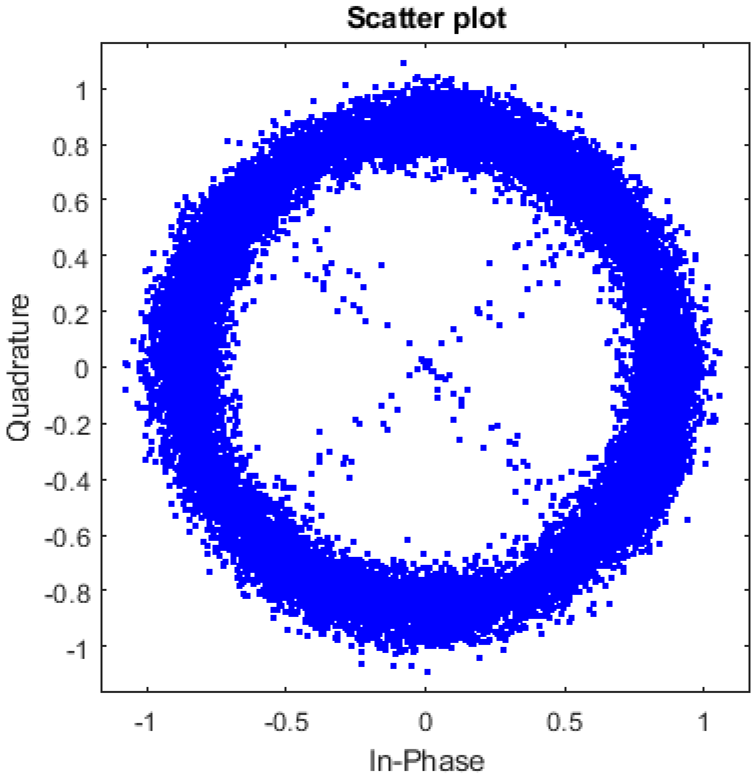
```
                Algorithm: 'LMS'
                  NumTaps: 5
                 StepSize: 0.0100
            Constellation: [1x4 double]
             ReferenceTap: 3
               InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: false
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

Equalize the impaired data. Plot the angular error from the channel, the equalizer error signal, and signal constellation. As the channel varies, the equalizer output does not remove the channel effects. The output constellation rotates out of sync, resulting in bit errors.

```
[y,err] = lineq(rx,trainingSymbols);

figure
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('Time (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(err))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel Without Retraining')
```



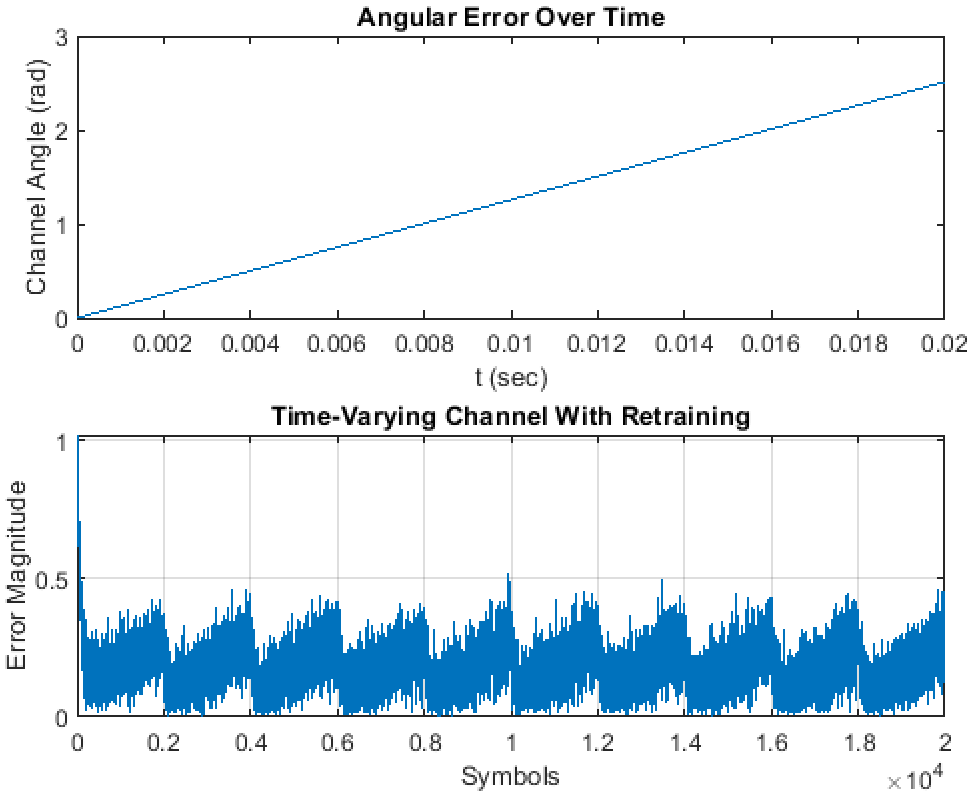
```
scatterplot(y)
```



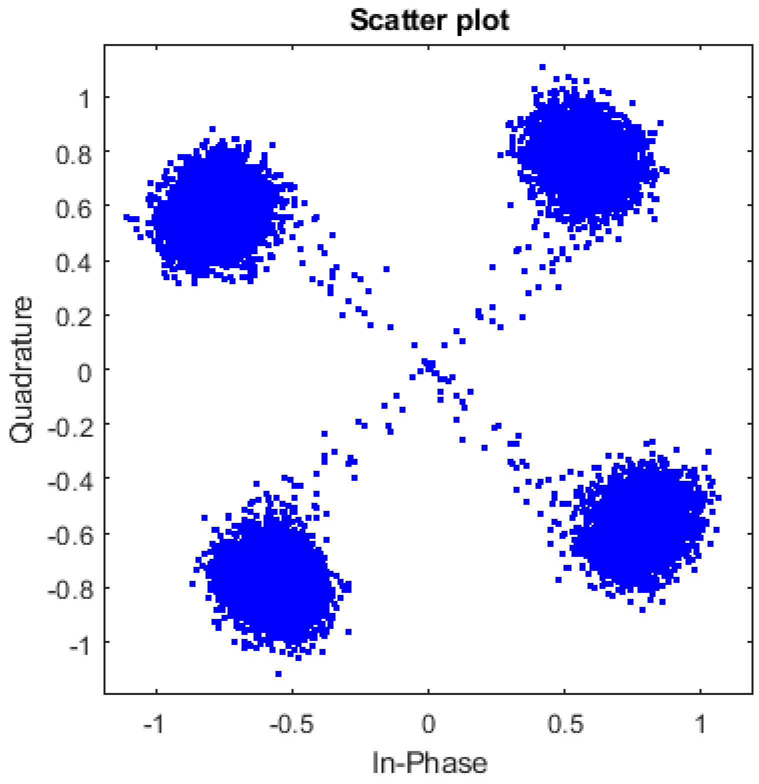
Set the `TrainingInputPort` property to `true` to configure the equalizer to retrain the taps when signaled by the `trainFlag` input. The equalizer trains only when `trainFlag` is `true`. After every 2000 symbols, the equalizer retrains the taps and keeps lock on variations of the channel. Plot the angular error from the channel, equalizer error signal, and signal constellation. As the channel varies, the equalizer output removes the channel effects. The output constellation does not rotate out of sync and bit errors are reduced.

```
release(lineq)
lineq.TrainingFlagInputPort = true;
symbolCnt = 0;
numPackets = length(rx)/spp;
trainFlag = true;
trainingPeriod = 2000;
eVec = zeros(size(rx));
yVec = zeros(size(rx));
for p=1:numPackets
    [yVec((p-1)*spp+1:p*spp,1),eVec((p-1)*spp+1:p*spp,1)] = ...
        lineq(rx((p-1)*spp+1:p*spp,1),trainingSymbols,trainFlag);
    symbolCnt = symbolCnt + spp;
    if symbolCnt >= trainingPeriod
        trainFlag = true;
        symbolCnt = 0;
    else
        trainFlag = false;
    end
end
figure
```

```
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('t (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(eVec))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel With Retraining')
```



```
scatterplot(yVec)
```



Linearly Equalize Delayed Signal

Simulate a system with delay between the transmitted symbols and received samples. Typical systems have transmitter and receiver filters that result in a delay. This delay must be accounted for to synchronize the system. In this example, the system delay is introduced without transmit and receive filters. Linear equalization, using the least mean squares (LMS) algorithm, recovers QPSK symbols.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
mpChan = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
systemDelay = dsp.Delay(20);
snr = 24;
```

Generate QPSK-modulated symbols. Apply multipath channel filtering, a system delay, and AWGN to the transmitted symbols.

```
data = randi([0 M-1],numSymbols,1);
tx = pskmod(data,M,pi/4); % OQPSK
delayedSym = systemDelay(filter(mpChan,1,tx));
rx = awgn(delayedSym,snr,'measured');
```

Create equalizer and EVM System objects. The equalizer System object specifies a linear equalizer that uses the LMS algorithm.

```
lineq = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',9,'ReferenceTap',5);
evm = comm.EVM('ReferenceSignalSource', ...
    'Estimated from reference constellation');
```

Equalize Without Adjusting Input Delay

Equalize the received symbols.

```
[y1,err1,wts1] = lineq(rx,tx(1:numTrainingSymbols,1));
```

Find the delay between the received symbols and the transmitted symbols by using the `finddelay` function.

```
rxDelay = finddelay(tx,rx)
```

```
rxDelay = 20
```

Display the equalizer information. The latency value indicates the delay introduced by the equalizer. Calculate the total delay as the sum of `rxDelay` and the equalizer latency.

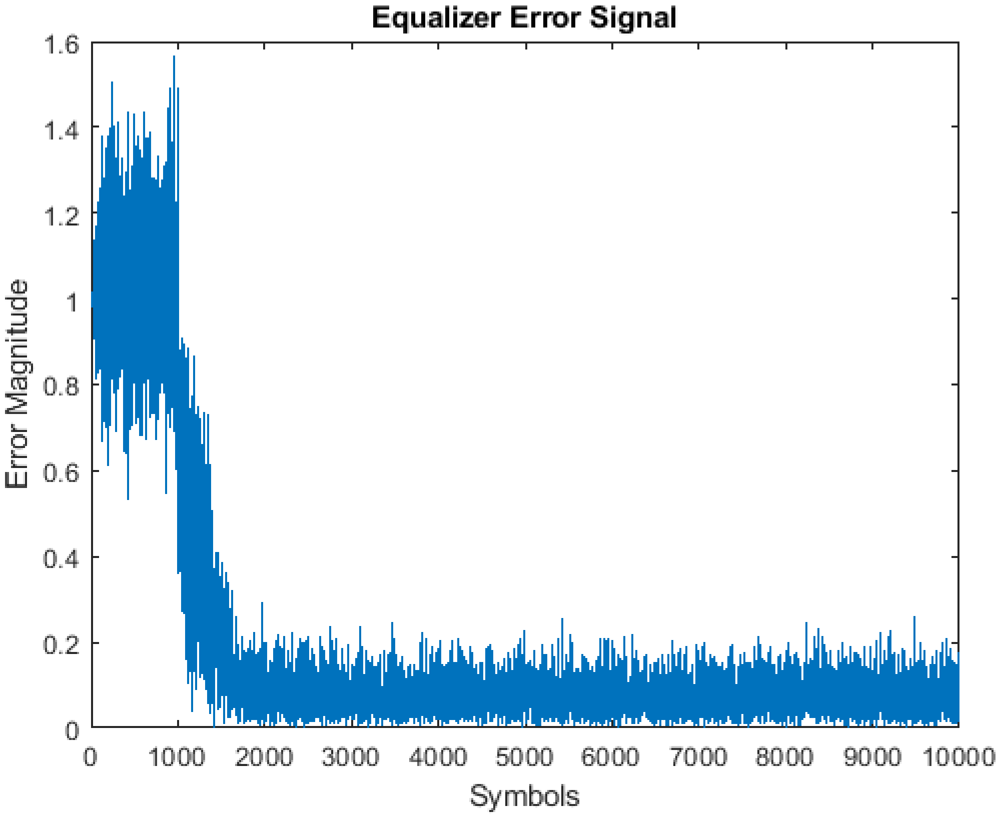
```
eqInfo = info(lineq)
```

```
eqInfo = struct with fields:
    Latency: 4
```

```
totalDelay = rxDelay + eqInfo.Latency;
```

Until the equalizer output converges, the symbol error rate is high. Plot the error output, `err1`, to determine when the equalized output converges.

```
plot(abs(err1))
xlabel('Symbols')
ylabel('Error Magnitude')
title('Equalizer Error Signal')
```



The plot shows excessive errors beyond the 1000 symbols training period. When demodulating symbols and computing symbol errors, to account for the unconverged output and the system delay between the equalizer output and transmitted symbols, skip the first 2000 symbols.

```
dataRec1 = pskdemod(y1(2000+totalDelay:end),M,pi/4);
symErrWithDelay = symerr(data(2000:end-totalDelay),dataRec1)
```

```
symErrWithDelay = 5999
```

```
evmWithDelay = evm(y1)
```

```
evmWithDelay = 29.5795
```

The error rate and EVM are high because the receive delay was not accounted for in the equalizer System object.

Adjust Input Delay in Equalizer

Equalize the received data by using the delay value to set the `InputDelay` property. Because `InputDelay` is a nontunable property, you must release the `lineq` System object to reconfigure the `InputDelay` property. Equalize the received symbols.

```
release(lineq)
lineq.InputDelay = rxDelay
```

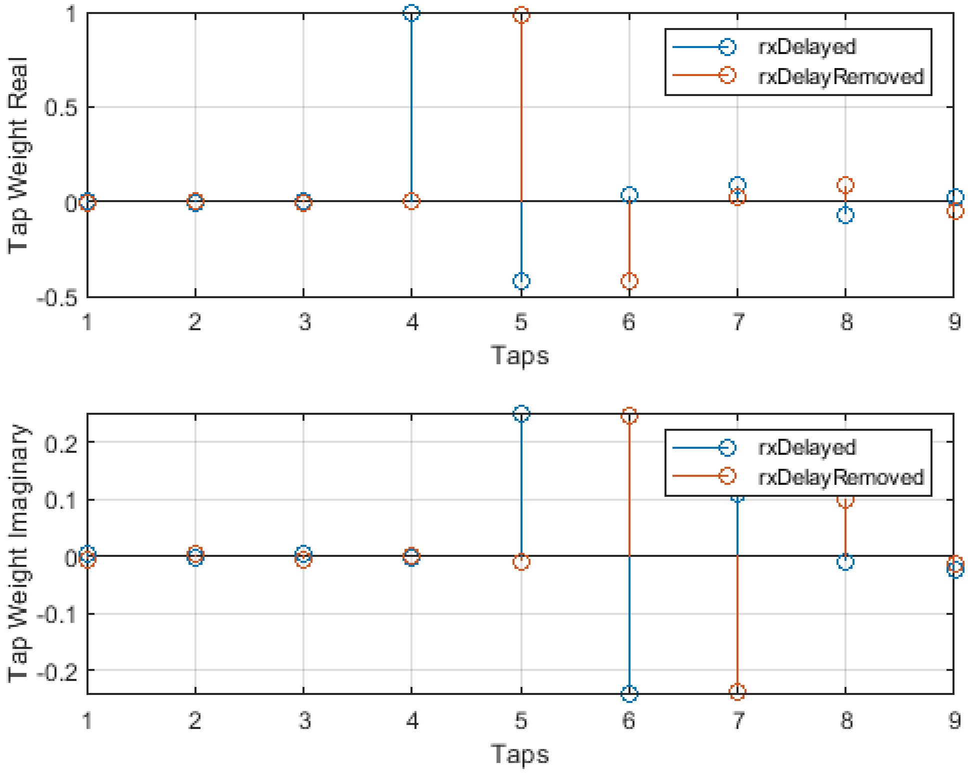
```
lineq = 
  comm.LinearEqualizer with properties:
```

```
                Algorithm: 'LMS'
                  NumTaps: 9
                 StepSize: 0.0100
            Constellation: [1x4 double]
             ReferenceTap: 5
               InputDelay: 20
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: true
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

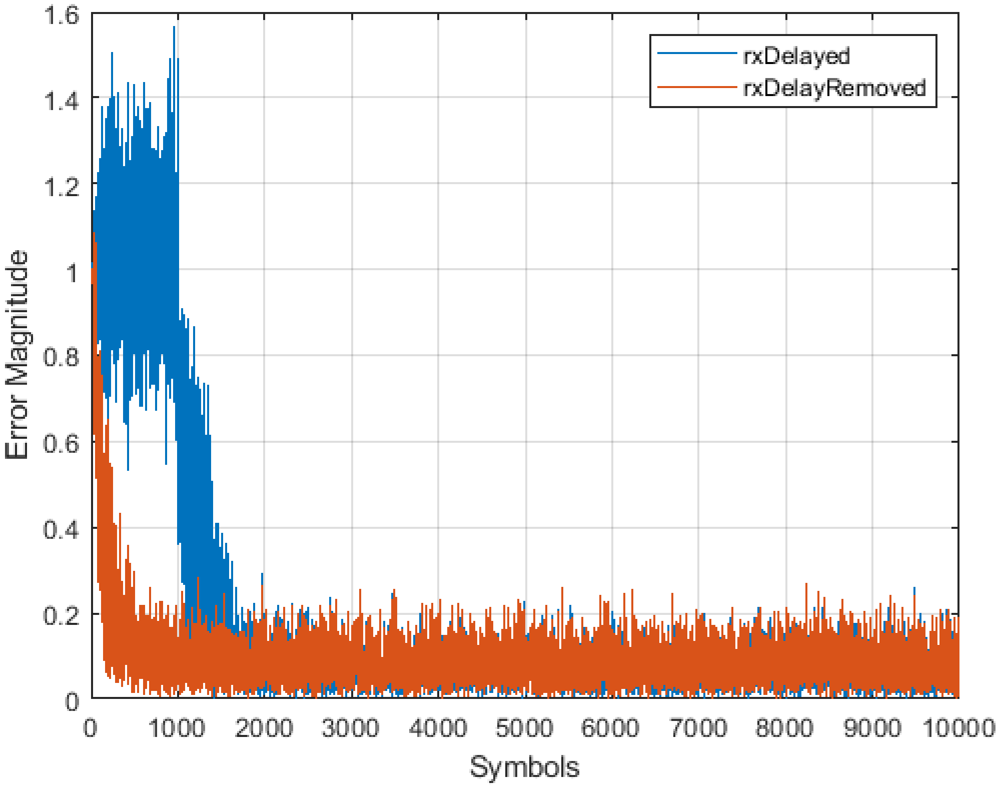
```
[y2,err2,wts2] = lineq(rx,tx(1:numTrainingSymbols,1));
```

Plot the tap weights and equalized error magnitude. A stem plot shows the equalizer tap weights before and after the system delay is removed. A 2-D line plot shows the slower equalizer convergence for the delayed signal as compared to the signal with the delay removed.

```
subplot(2,1,1)
stem([real(wts1),real(wts2)])
xlabel('Taps')
ylabel('Tap Weight Real')
legend('rxDelayed','rxDelayRemoved')
grid on
subplot(2,1,2)
stem([imag(wts1),imag(wts2)])
xlabel('Taps')
ylabel('Tap Weight Imaginary')
legend('rxDelayed','rxDelayRemoved')
grid on
```



```
figure
plot([abs(err1),abs(err2)])
xlabel('Symbols')
ylabel('Error Magnitude')
legend('rxDelayed','rxDelayRemoved')
grid on
```



Plot error output of the equalized signals, `rxDelayed` and `rxDelayRemoved`. For the signal that has the delay removed, the equalizer converges during the 1000 symbol training period. When demodulating symbols and computing symbol errors, to account for the unconverged output and the system delay between the equalizer output and transmitted symbols, skip the first 500 symbols. Reconfiguring the equalizer to account for the system delay enables better equalization of the signal, and reduces symbol errors and the EVM.

```
eqInfo = info(lineq)
```

```
eqInfo = struct with fields:
    Latency: 4
```

```
totalDelay = rxDelay + eqInfo.Latency;
dataRec2 = pskdemod(y2(500+totalDelay:end),M,pi/4);
symErrDelayRemoved = symerr(data(500:end-totalDelay),dataRec2)
```

```
symErrDelayRemoved = 0
```

```
evmDelayRemoved = evm(y2(500+totalDelay:end))
```

```
evmDelayRemoved = 9.4435
```

Linearly Equalize Symbols By Using EVM-Based Training

Recover QPSK symbols with a linear equalizer by using the constant modulus algorithm (CMA) and EVM-based taps training. When using blind equalizer algorithms, such as CMA, train the equalizer taps by using the AdaptWeights property to start and stop training. Helper functions are used to generate plots and apply phase correction.

Initialize system variables.

```
rng(123456);
M = 4; % QPSK
numSymbols = 100;
numPackets = 5000;
raylChan = comm.RayleighChannel('PathDelays',[0 1], ...
    'AveragePathGains',[0 -12],'MaximumDopplerShift',1e-5);
SNR = 50;
adaptWeights = true;
```

Create the equalizer and EVM System objects. The equalizer System object specifies a linear equalizer by using the CMA adaptive algorithm. Call the helper function to initialize figure plots.

```
lineq = comm.LinearEqualizer('Algorithm','CMA', ...
    'NumTaps',5,'ReferenceTap',3, ...
    'StepSize',0.03,'AdaptWeightsSource','Input port')
```

```
lineq = 
  comm.LinearEqualizer with properties:

                Algorithm: 'CMA'
                  NumTaps: 5
                 StepSize: 0.0300
            Constellation: [1x4 double]
             ReferenceTap: 3
    InputSamplesPerSymbol: 1
       AdaptWeightsSource: 'Input port'
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

```
info(lineq)
```

```
ans = struct with fields:
    Latency: 2
```

```
evm = comm.EVM('ReferenceSignalSource', ...
    'Estimated from reference constellation');
[errPlot,evmPlot,scatSym,adaptState] = initFigures(numPackets,lineq);
```

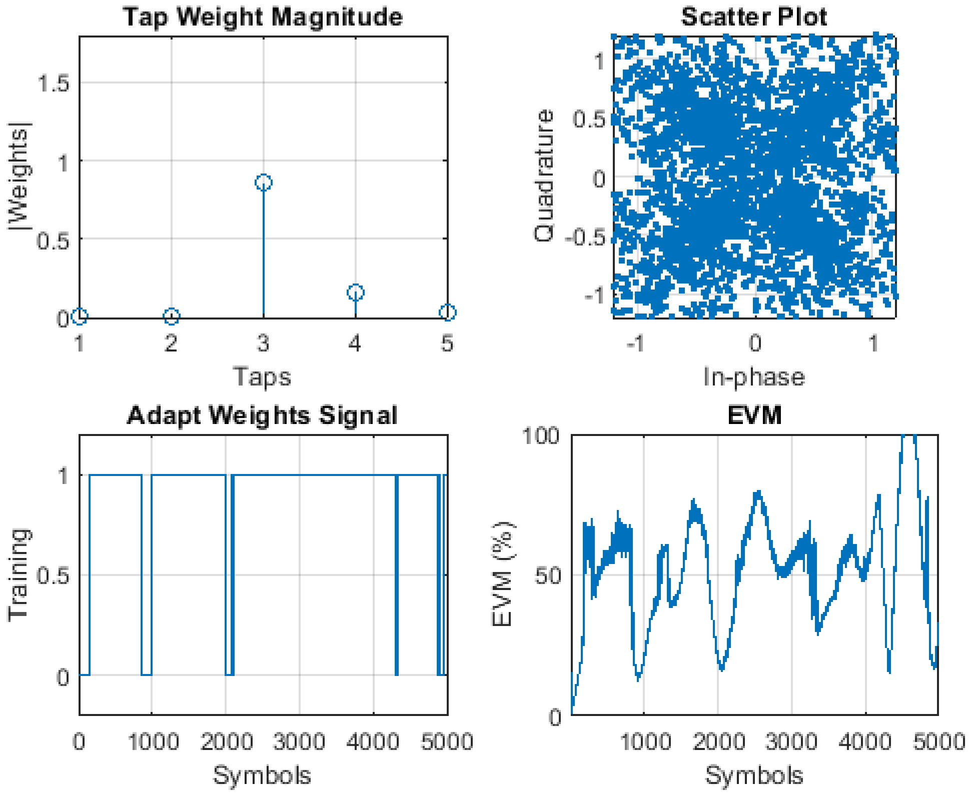
Equalization Loop

To implement the equalization loop:

1. Generate PSK data packets.
2. Apply Rayleigh fading and AWGN to the transmission data.
3. Apply equalization to the received data and phase correction to the equalizer output.
4. Estimate the EVM and toggle the adaptWeights flag to true or false based on the EVM level.
5. Update the figure plots.

```
for p=1:numPackets
```

```
    data = randi([0 M-1],numSymbols,1);
    tx = pskmod(data,M,pi/4);
    rx = awgn(raylChan(tx),SNR);
    rxDelay = finddelay(rx,tx);
    [y,err,wts] = lineq(rx,adaptWeights);
    y = phaseCorrection(y);
    evmEst = evm(y);
    adaptWeights = (evmEst > 20);

    updateFigures(errPlot,evmPlot,scatSym,adaptState, ...
        wts,y(end),evmEst,adaptWeights,p,numPackets)
end
```



```
rxDelay

rxDelay = 0
```

The figure plots show that, as the EVM varies, the equalizer toggles in and out of decision-directed weight adaptation mode.

Helper Functions

This helper function initializes figures that show a quad plot of simulation results.

```
function [errPlot,evmPlot,scatter,adaptState] = initFigures(numPkts,lineq)
yVec = nan(numPkts,1);
evmVec = nan(numPkts,1);
wVec = zeros(lineq.NumTaps,1);
```

```
adaptVec = nan(numPkts,1);

figure
subplot(2,2,1)
evmPlot = stem(wVec);
grid on; axis([1 lineq.NumTaps 0 1.8])
xlabel('Taps'); ylabel('|Weights|'); title('Tap Weight Magnitude')

subplot(2,2,2)
scatter = plot(yVec, '.');
axis square; axis([-1.2 1.2 -1.2 1.2]); grid on
xlabel('In-phase'); ylabel('Quadrature'); title('Scatter Plot');
subplot(2,2,3)
adaptState = plot(adaptVec);
grid on; axis([0 numPkts -0.2 1.2])
ylabel('Training'); xlabel('Symbols'); title('Adapt Weights Signal')
subplot(2,2,4)
errPlot = plot(evmVec);
grid on; axis([1 numPkts 0 100])
xlabel('Symbols'); ylabel('EVM (%)'); title('EVM')
end
```

This helper function updates figures.

```
function updateFigures(errPlot,evmPlot,scatSym, ...
    adaptState,w,y,evmEst,adaptWts,p,numFrames)
persistent yVec evmVec adaptVec

if p == 1
    yVec = nan(numFrames,1);
    evmVec = nan(numFrames,1);
    adaptVec = nan(numFrames,1);
end

yVec(p) = y;
evmVec(p) = evmEst;
adaptVec(p) = adaptWts;

errPlot.YData = abs(evmVec);
evmPlot.YData = abs(w);
scatSym.XData = real(yVec);
scatSym.YData = imag(yVec);
adaptState.YData = adaptVec;
drawnow limitrate
end
```

This helper function applies phase correction.

```
function y = phaseCorrection(y)
a = angle(y((real(y) > 0) & (imag(y) > 0)));
a(a < 0.1) = a(a < 0.1) + pi/2;
theta = mean(a) - pi/4;
y = y * exp(-1i*theta);
end
```

Linearly Equalize Packetized Signals in Fading Environments

Recover QPSK symbols in fading environments with a linear equalizer, using the least mean squares (LMS) algorithm. Use the `reset` object function to equalize independent packets. Use helper functions to generate plots. This example also shows symbol-based processing and frame-based processing.

Setup

Initialize system variables, create an equalizer System object, and initialize the plot figures.

```
M = 4; % QPSK
numSym = 1000;
numTrainingSym = 100;
numPackets = 5;
numTaps = 9;
ttlNumSym = numSym + numTrainingSym;
raylChan = comm.RayleighChannel('PathDelays',[0 1], ...
    'AveragePathGains',[0 -9], ...
    'MaximumDopplerShift',0, ...
    'PathGainsOutputPort',true);
SNR = 35;
rxVec = zeros(ttlNumSym,numPackets);
txVec = zeros(ttlNumSym,numPackets);
yVec = zeros(ttlNumSym,1);
eVec = zeros(ttlNumSym,1);

lineq1 = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',numTaps,'ReferenceTap',5, ...
    'StepSize',0.01,'TrainingFlagInputPort',true);

[errPlot,wStem,hStem,scatPlot] = initFigures(ttlNumSym,lineq1, ...
    raylChan.AveragePathGains);
```

Symbol-Based Processing

For symbol-based processing, provide one symbol at the input of the equalizer. Reset the equalizer state and channel after processing each packet.

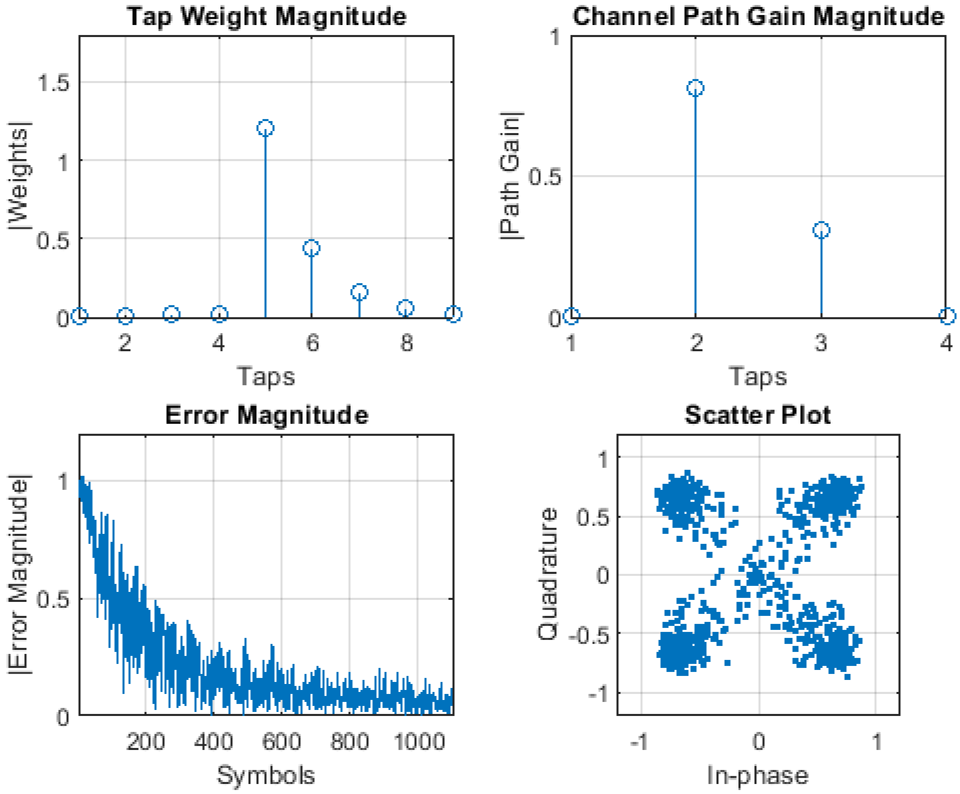
```
for p = 1:numPackets
    trainingFlag = true;
    for q=1:ttlNumSym
        data = randi([0 M-1],1,1);
        tx = pskmod(data,M,pi/4);
        [xc,pg] = raylChan(tx);
        rx = awgn(xc,25);
        [y,err,wts] = lineq1(rx,tx,trainingFlag);
```

Disable training after processing `numTrainingSym` training symbols.

```
        if q == numTrainingSym
            trainingFlag = false;
        end
        updateFigures(errPlot,wStem,hStem,scatPlot,err,wts,y,pg,q,ttlNumSym);
        txVec(q,p) = tx;
        rxVec(q,p) = rx;
    end
```

After processing each packet, reset the channel System object to get a new realization of channel taps and the equalizer System object to restore the default set of tap weights.

```
    reset(raylChan)
    reset(lineq1)
end
```



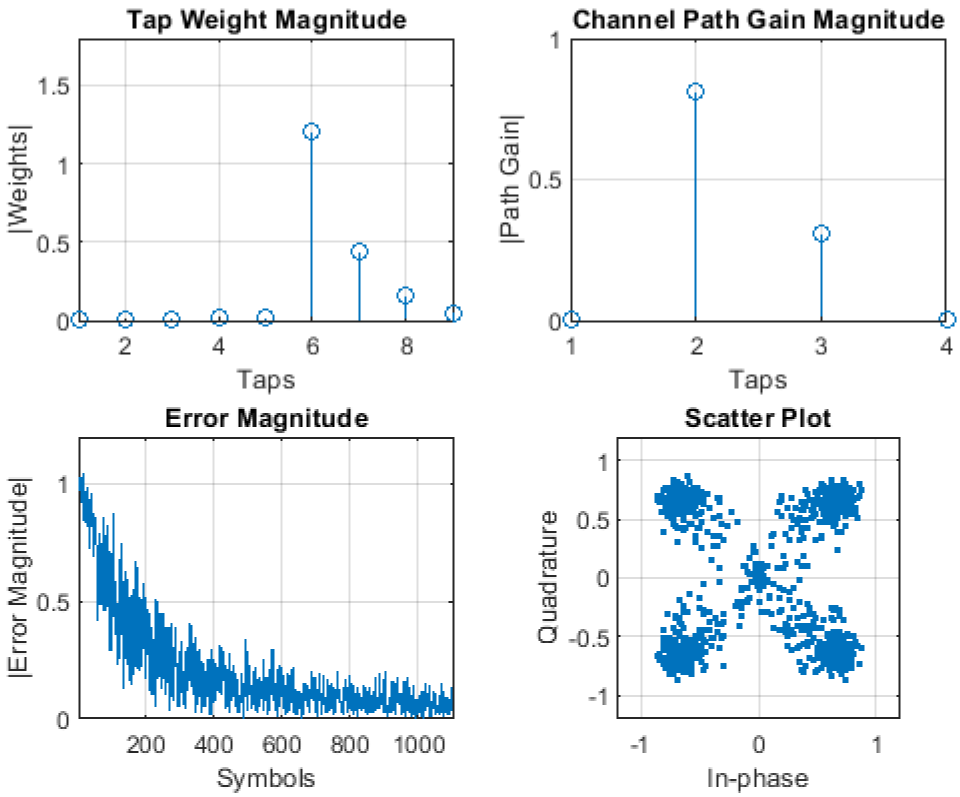
Packet-Based Processing

For packet-based processing, provide one packet at the input of the equalizer. Each packet contains `ttlNumSym` symbols. Because the training duration is less than the packet length, you do not need to specify the start-training input.

```
yVecPkt = zeros(ttlNumSym,numPackets);
errVecPkt = zeros(ttlNumSym,numPackets);
wgtVecPkt = zeros(numTaps,numPackets);
lineq2 = comm.LinearEqualizer('Algorithm','LMS', ...
    'NumTaps',9,'ReferenceTap',6,'StepSize',0.01);
for p = 1:numPackets
    [yVecPkt(:,p),errVecPkt(:,p),wgtVecPkt(:,p)] = ...
        lineq2(rxVec(:,p),txVec(1:numTrainingSym,p));
    for q=1:ttlNumSym
        updateFigures(errPlot,wStem,hStem,scatPlot, ...
            errVecPkt(q,p),wgtVecPkt(:,p),yVecPkt(q,p),pg,q,ttlNumSym);
    end
```

After processing each packet, reset the channel System object to get a new realization of channel taps and the equalizer System object to restore the default set of tap weights.

```
    reset(raylChan)
    reset(lineq2)
end
```



Helper Functions

The helper function initializes the figures.

```
function [errPlot,wStem,hStem,scatPlot] = initFigures(ttlNumSym,lineq,pg)
yVec = nan(ttlNumSym,1);
eVec = nan(ttlNumSym,1);
wVec = zeros(lineq.NumTaps,1);
figure;
subplot(2,2,1);
wStem = stem(wVec);
axis([1 lineq.NumTaps 0 1.8]); grid on
xlabel('Taps'); ylabel('|Weights|'); title('Tap Weight Magnitude')
subplot(2,2,2);
hStem = stem([0 abs(pg) 0]);
grid on;
xlabel('Taps'); ylabel('|Path Gain|'); title('Channel Path Gain Magnitude')
subplot(2,2,3);
errPlot = plot(eVec);
axis([1 ttlNumSym 0 1.2]); grid on
xlabel('Symbols'); ylabel('|Error Magnitude|'); title('Error Magnitude')
subplot(2,2,4);
scatPlot = plot(yVec,'.');
axis square; axis([-1.2 1.2 -1.2 1.2]); grid on;
```

```
xlabel('In-phase'); ylabel('Quadrature'); title(sprintf('Scatter Plot'));
end
```

This helper function updates the figures.

```
function updateFigures(errPlot,wStem,hStem,scatPlot, ...
    err,wts,y,pg,p,ttlNumSym)
persistent yVec eVec
if p == 1
    yVec = nan(ttlNumSym,1);
    eVec = nan(ttlNumSym,1);
end
yVec(p) = y;
eVec(p) = abs(err);
errPlot.YData = abs(eVec);
wStem.YData = abs(wts);
hStem.YData = [0 abs(pg) 0];
scatPlot.XData = real(yVec);
scatPlot.YData = imag(yVec);
drawnow limitrate
end
```

Nonadaptive Linear Equalization

Use the linear equalizer in nonadaptive mode. Use the `mmseweights` object function to calculate the minimum mean squared error (MMSE) solution and use the weights returned as the set of tap weights for the linear equalizer.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
chtaps = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
EbN0 = 20;
```

Generate QPSK modulated symbols. Apply delayed multipath channel filtering and AWGN impairments to the symbols.

```
data = randi([0 M-1], numSymbols, 1);
tx = pskmod(data, M, pi/4);
rx = awgn(filter(chtaps,1,tx),25,'measured');
```

Create a linear equalizer System object configured to use CMA algorithm, set the `AdaptWeights` property to `false`, and the `InitialWeightsSource` property to `Property`. Calculate the MMSE weights. Set the initial tap weights to the calculated MMSE weights. Equalize the impaired symbols.

```
eq = comm.LinearEqualizer('Algorithm','CMA','AdaptWeights',false,'InitialWeightsSource','Property
```

```
eq = 
  comm.LinearEqualizer with properties:

                Algorithm: 'CMA'
                  NumTaps: 5
                 StepSize: 0.0100
            Constellation: [1x4 double]
```

```
    InputSamplesPerSymbol: 1
       AdaptWeightsSource: 'Property'
             AdaptWeights: false
     InitialWeightsSource: 'Property'
           InitialWeights: [5x1 double]
       WeightUpdatePeriod: 1
```

```
wgts = mmseweights(eq,chtaps,EbN0)
```

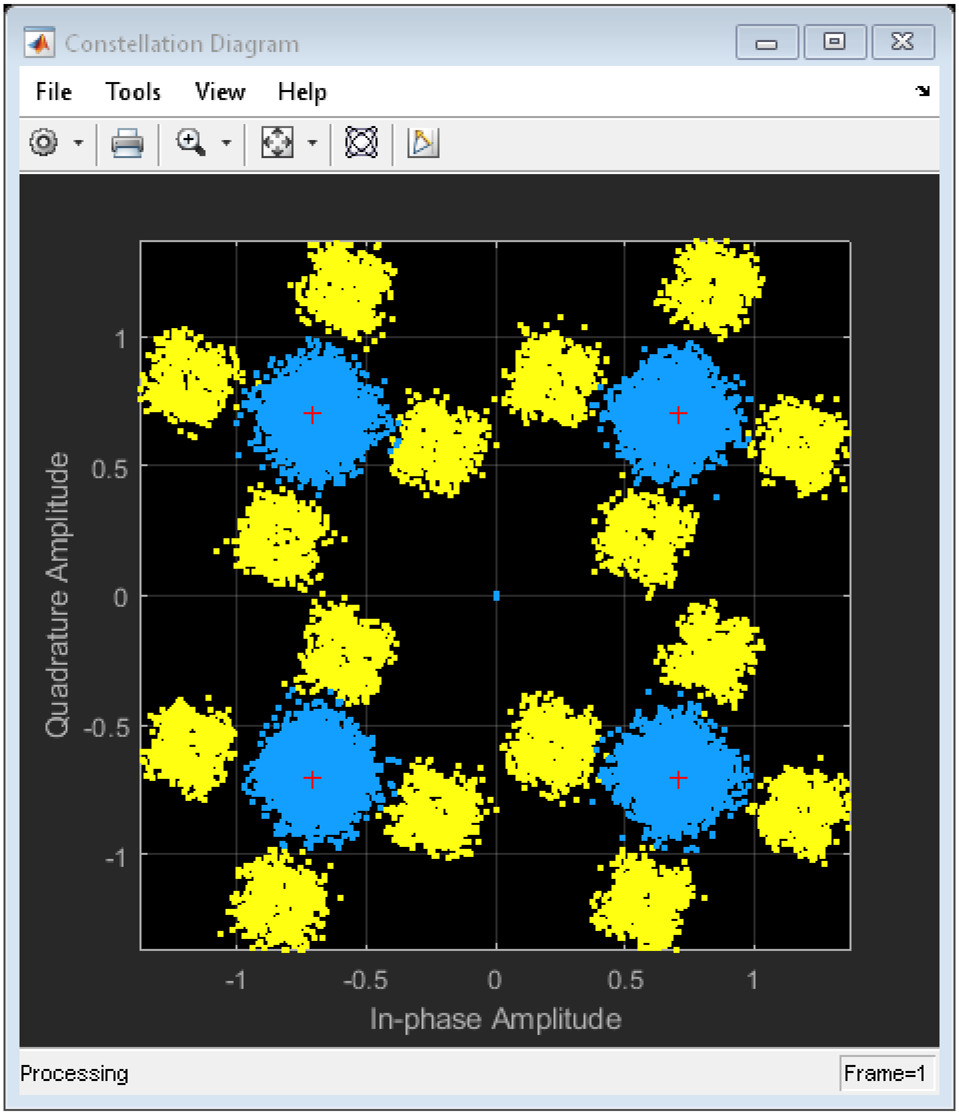
```
wgts = 5×1 complex

   0.0005 - 0.0068i
   0.0103 + 0.0117i
   0.9694 - 0.0019i
  -0.3987 + 0.2186i
   0.0389 - 0.1756i
```

```
eq.InitialWeights = wgts;
```

```
[y,err,weights] = eq(rx);
```

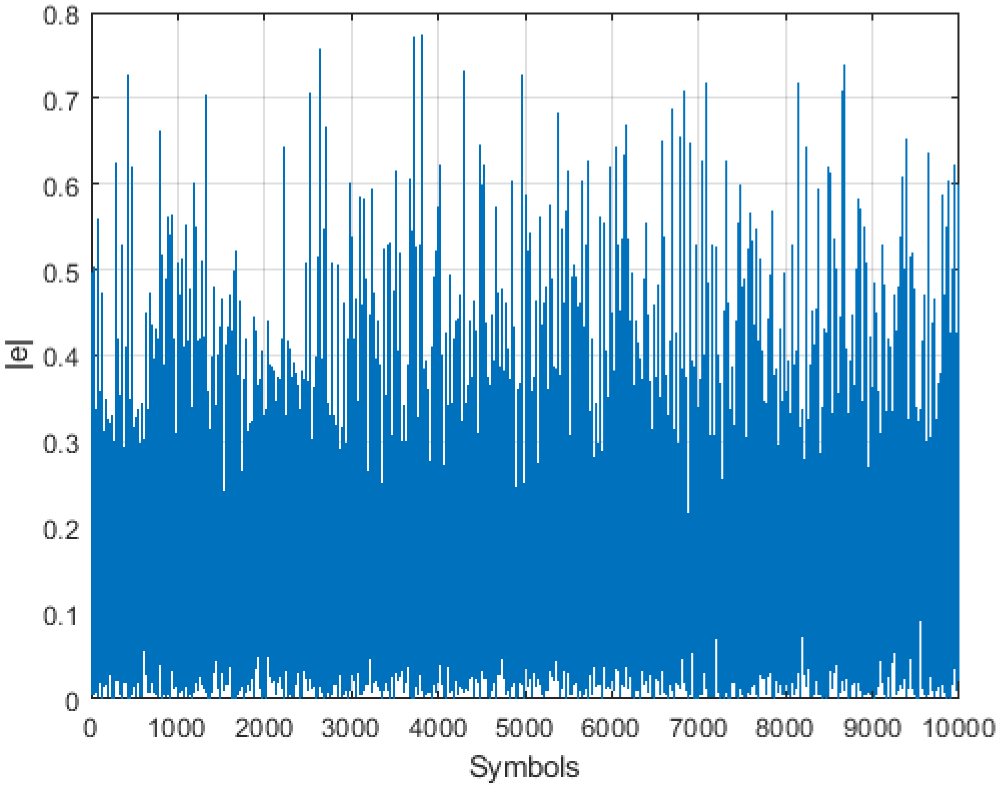
Plot constellation of impaired and equalized symbols.

```
constell = comm.ConstellationDiagram('NumInputPorts',2);
constell(rx,y)
```



Plot the equalizer error signal and compute the error vector magnitude of the equalized symbols.

```
plot(abs(err))
grid on; xlabel('Symbols'); ylabel('|e|')
```

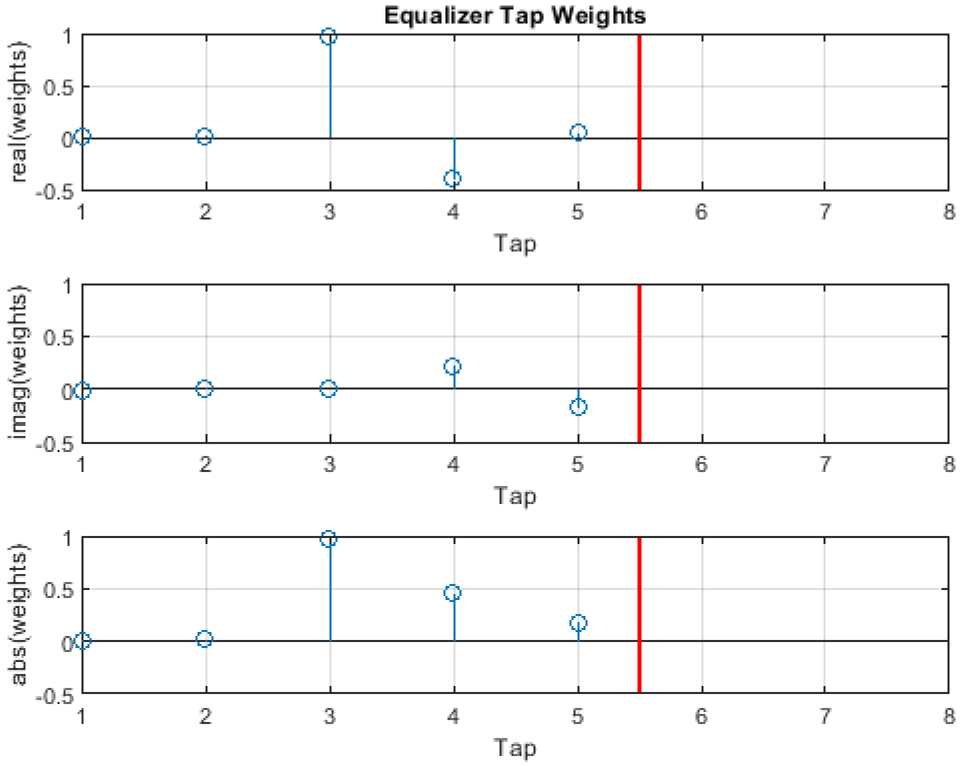


```
errevm = comm.EVM;
evm = errevm(tx,y)
```

```
evm = 140.6177
```

Plot equalizer tap weights.

```
subplot(3,1,1); stem(real(weights)); ylabel('real(weights)'); xlabel('Tap'); grid on; axis([1 8 -
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
title('Equalizer Tap Weights')
subplot(3,1,2); stem(imag(weights)); ylabel('imag(weights)'); xlabel('Tap'); grid on; axis([1 8 -
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
subplot(3,1,3); stem(abs(weights)); ylabel('abs(weights)'); xlabel('Tap'); grid on; axis([1 8 -0
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
```



More About

Symbol Tap Spacing

You can configure the equalizer to operate as a symbol-spaced equalizer or as a fractional symbol-spaced equalizer.

- To operate the equalizer at a symbol-spaced rate, specify the number of samples per symbol as 1. Symbol-rate equalizers have taps spaced at the symbol duration. Symbol-rate equalizers are sensitive to timing phase.
- To operate the equalizer at a fractional symbol-spaced rate, specify the number of input samples per symbol as an integer greater than 1 and provide an input signal oversampled at that sampling rate. Fractional symbol-spaced equalizers have taps spaced at an integer fraction of the input symbol duration. Fractional symbol-spaced equalizers are not sensitive to timing phase.

Algorithms

Linear Equalizers

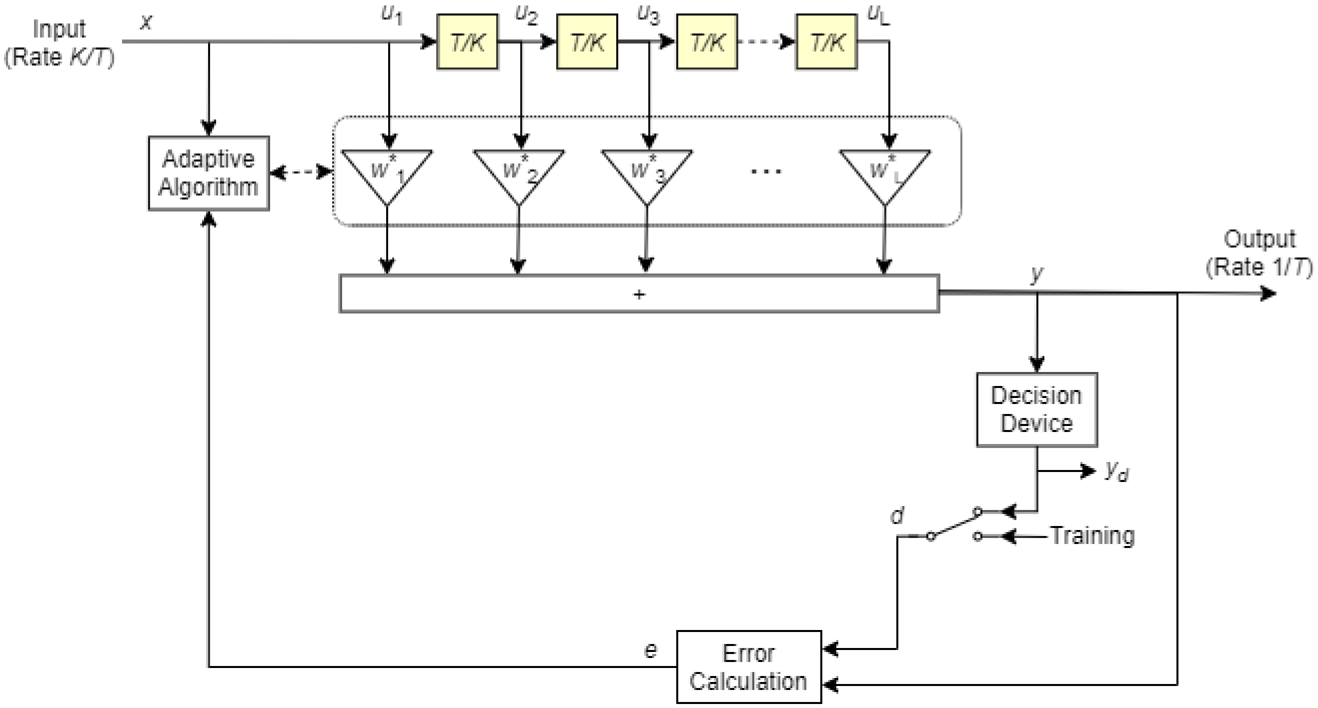
Linear equalizers can remove intersymbol interference (ISI) when the frequency response of a channel has no null. If a null exists in the frequency response of a channel, linear equalizers tend to enhance the noise. In this case, use decision feedback equalizers to avoid enhancing the noise.

A linear equalizer consists of a tapped delay line that stores samples from the input signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period.

Linear equalizers can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, *K*, is 1. The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, *K*, is an integer greater than 1. Typically, *K* is 4 for fractionally spaced equalizers. The output sample rate is 1/*T* and the input sample rate is *K/T*, where *T* is the symbol period. Tap-weight updating occurs at the output rate.

This schematic shows a linear equalizer with *L* weights, a symbol period of *T*, and *K* samples per symbol. If *K* is 1, the result is a symbol-spaced linear equalizer instead of a fractional symbol-spaced linear equalizer.



In each symbol period, the equalizer receives *K* input samples at the tapped delay line. The equalizer then outputs a weighted sum of the values in the tapped delay line and updates the weights to prepare for the next symbol period.

For more information, see “Equalization”.

Least Mean Square (LMS) Algorithm

For the LMS algorithm, in the previous schematic, *w* is a vector of all weights w_i, and *u* is a vector of all inputs u_i. Based on the current set of weights, the LMS algorithm creates the new set of weights as

$$w_{\text{new}} = w_{\text{current}} + (StepSize)\, ue^*.$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed when using the LMS adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Recursive Least Square (RLS) Algorithm

For the RLS algorithm, in the previous schematic, w is the vector of all weights w_i, and u is the vector of all inputs u_i. Based on the current set of inputs, u, and the inverse correlation matrix, P, the RLS algorithm first computes the Kalman gain vector, K, as

$$K = \frac{Pu}{(ForgettingFactor) + u^H Pu}\,.$$

The forgetting factor used by the adaptive algorithm is specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalized output signal to be less stable. H denotes the Hermitian transpose. Based on the current inverse correlation matrix, the new inverse correlation matrix is

$$P_{\text{new}} = \frac{(1 - Ku^H)P_{\text{current}}}{ForgettingFactor}\,.$$

Based on the current set of weights, the RLS algorithm creates the new set of weights as

$$w_{\text{new}} = w_{\text{current}} + K^* e.$$

The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Constant Modulus Algorithm (CMA)

For the CMA adaptive algorithm, in the previous schematic, w is the vector of all weights w_i, and u is the vector of all inputs u_i. Based on the current set of weights, the CMA adaptive algorithm creates the new set of weights as

$$w_{\text{new}} = w_{\text{current}} + (StepSize)\ u^* e.$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed by the CMA adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = y(R - |y|^2)$, where R is a constant related to the signal constellation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`comm.DecisionFeedback` | `comm.MLSEEqualizer`

Blocks

Linear Equalizer

Topics
"Equalization"
"Adaptive Equalizers"

Introduced in R2019a

comm.DecisionFeedbackEqualizer

Package: `comm`

Equalize modulated signals using decision feedback filtering

Description

The `comm.DecisionFeedbackEqualizer` System object uses a decision feedback filter tap delay line with a weighted sum to equalize modulated signals transmitted through a dispersive channel. The equalizer object adaptively adjusts tap weights based on the selected algorithm. For more information, see “Algorithms” on page 3-543.

To equalize modulated signals using a decision feedback filter:

1. Create the `comm.DecisionFeedbackEqualizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
dfe = comm.DecisionFeedbackEqualizer
dfe = comm.DecisionFeedbackEqualizer(Name,Value)
```

Description

`dfe = comm.DecisionFeedbackEqualizer` creates a decision feedback equalizer System object to adaptively equalize a signal.

`dfe = comm.DecisionFeedbackEqualizer(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.DecisionFeedbackEqualizer('Algorithm','RLS')` configures the equalizer object to update tap weights using the recursive least squares (RLS) algorithm. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Algorithm` — Adaptive algorithm
`'LMS'` (default) | `'RLS'` | `'CMA'`

Adaptive algorithm used for equalization, specified as one of these values:

- 'LMS' — Update the equalizer tap weights using the “Least Mean Square (LMS) Algorithm” on page 3-545.
- 'RLS' — Update the equalizer tap weights using the “Recursive Least Square (RLS) Algorithm” on page 3-545.
- 'CMA' — Update the equalizer tap weights using the “Constant Modulus Algorithm (CMA)” on page 3-545.

Data Types: char | string

NumForwardTaps — Number of forward equalizer taps

5 (default) | positive integer

Number of forward equalizer taps, specified as a positive integer. The number of forward equalizer taps must be greater than or equal to the value of the InputSamplesPerSymbol property.

Data Types: double

NumFeedbackTaps — Number of feedback equalizer taps

3 (default) | positive integer

Number of feedback equalizer taps, specified as a positive integer.

Data Types: double

StepSize — Step size

0.01 (default) | positive scalar

Step size used by the adaptive algorithm, specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tip To determine the maximum step size allowed, use the maxstep object function.

Tunable: Yes

Dependencies

To enable this property, set Algorithm to 'LMS' or 'CMA'.

Data Types: double

ForgettingFactor — Forgetting factor

0.99 (default) | scalar in the range (0, 1]

Forgetting factor used by the adaptive algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this property, set Algorithm to 'RLS'.

Data Types: double

`InitialInverseCorrelationMatrix` — Initial inverse correlation matrix

`0.1` (default) | scalar | matrix

Initial inverse correlation matrix, specified as a scalar or an N_{Taps}-by-N_{Taps} matrix. N_{Taps} is equal to the sum of the NumForwardTaps and NumFeedbackTaps property values. If you specify `InitialInverseCorrelationMatrix` as a scalar, *a*, the equalizer sets the initial inverse correlation matrix to *a* times the identity matrix: *a*(`eye`(N_{Taps})).

Tunable: Yes

Dependencies

To enable this property, set Algorithm to `'RLS'`.

Data Types: `double`

`Constellation` — Signal constellation

`pskmod(0:3,4,pi/4)` (default) | vector

Signal constellation, specified as a vector. The default value is a QPSK constellation generated using this code: `pskmod(0:3,4,pi/4)`.

Tunable: Yes

Data Types: `double`

`ReferenceTap` — Reference tap

`3` (default) | positive integer

Reference tap, specified as a positive integer less than or equal to the NumForwardTaps property value. The equalizer uses the reference tap location to track the main energy of the channel.

Data Types: `double`

`InputDelay` — Input signal delay

`0` (default) | nonnegative integer

Input signal delay in samples relative to the reset time of the equalizer, specified as a nonnegative integer. If the input signal is a vector of length greater than 1, then the input delay is relative to the start of the input vector. If the input signal is a scalar, then the input delay is relative to the first call of the System object and to the first call of the System object after calling the `release` or `reset` object function.

Data Types: `double`

`InputSamplesPerSymbol` — Number of input samples per symbol

`1` (default) | positive integer

Number of input samples per symbol, specified as a positive integer. Setting this property to any number greater than one effectively creates a fractionally spaced equalizer.

Data Types: `double`

`TrainingFlagInputPort` — Enable training control input

`false` (default) | `true`

Enable training control input, specified as `false` or `true`. Setting this property to `true` enables the equalizer training flag input `tf`.

Tunable: Yes

Data Types: `logical`

`AdaptAfterTraining` — Update tap weights when not training

`true` (default) | `false`

Update tap weights when not training, specified as `true` or `false`. If this property is set to `true`, the System object uses decision directed mode to update equalizer tap weights. If this property is set to `false`, the System object keeps the equalizer tap weights unchanged after training.

Tunable: Yes

Data Types: `logical`

`AdaptWeightsSource` — Source of adapt tap weights request

`'Property'` (default) | `'Input port'`

Source of adapt tap weights request, specified as one of these values:

- `'Property'` — Specify this value to use the AdaptWeights property to control when the System object adapts tap weights.
- `'Input port'` — Specify this value to use the `aw` input to control when the System object adapts tap weights.

Dependencies

To enable this property, set Algorithm to `'CMA'`.

Data Types: `char` | `string`

`AdaptWeights` — Adapt tap weights

`true` (default) | `false`

Adapt tap weights, specified as `true` or `false`. If this property is set to `true`, the System object updates the equalizer tap weights. If this property is set to `false`, the System object keeps the equalizer tap weights unchanged.

Tunable: Yes

Dependencies

To enable this property, set AdaptWeightsSource to `'Property'` and set AdaptAfterTraining to `true`.

Data Types: `logical`

`InitialWeightsSource` — Source for initial tap weights

`'Auto'` (default) | `'Property'`

Source for initial tap weights, specified as one of these values:

- `'Auto'` — Initialize the tap weights to the algorithm-specific default values, as described in the InitialWeights property.
- `'Property'` — Initialize the tap weights using the InitialWeights property value.

Data Types: `char` | `string`

`InitialWeights` — Initial weights
`0` or `[0;0;1;0;0]` (default) | scalar | vector

Initial weights used by the adaptive algorithm, specified as a scalar or vector. The default is `0` when the Algorithm property is set to `'LMS'` or `'RLS'`. The default is `[0;0;1;0;0]` when the Algorithm property is set to `'CMA'`.

If you specify `InitialWeights` as a scalar, the equalizer uses scalar expansion to create a vector of length N_{Taps} with all values set to `InitialWeights`. N_{Taps} is equal to the sum of the NumForwardTaps and NumFeedbackTaps property values. If you specify `InitialWeights` as a vector, the vector length must be N_{Taps}.

Tunable: Yes

Data Types: `double`

`WeightUpdatePeriod` — Tap weight update period
`1` (default) | positive integer

Tap weight update period in symbols, specified as a positive integer. The equalizer updates the tap weights after processing this number of symbols.

Data Types: `double`

Usage

Syntax

```
y = dfe(x,tsym)
y = dfe(x,tsym,tf)
y = dfe(x)
y = dfe(x,aw)
[y,err] = dfe( ___ )
[y,err,weights] = dfe( ___ )
```

Description

`y = dfe(x,tsym)` equalizes input signal `x` by using training symbols `tsym`. The output is the equalized symbols. To enable this syntax, set the Algorithm property to `'LMS'` or `'RLS'`.

`y = dfe(x,tsym,tf)` also specifies training flag `tf`. The System object starts training when `tf` changes from `false` to `true` (at the rising edge). The System object trains until all symbols in `tsym` are processed. The input `tsym` is ignored when `tf` is `false`. To enable this syntax, set the Algorithm property to `'LMS'` or `'RLS'` and TrainingFlagInputPort property to `true`.

`y = dfe(x)` equalizes input signal `x`. To enable this syntax, set the Algorithm property to `'CMA'`.

`y = dfe(x,aw)` also specifies adapts weights flag `aw`. The System object adapts the equalizer tap weights when `aw` is `true`. If `aw` is `false`, the System object keeps the weights unchanged. To enable this syntax, set the Algorithm property to `'CMA'` and AdaptWeightsSource property to `'Input port'`.

`[y,err] = dfe( ___ )` also returns error signal `err` using input arguments from any of the previous syntaxes.

[y,err,weights] = dfe(___) also returns weights, the tap weights from the last tap weight update, using input arguments from any of the previous syntaxes.

Input Arguments

x — Input signal

column vector

Input signal, specified as a column vector. The input signal vector length must be equal to an integer multiple of the InputSamplesPerSymbol property value. For more information, see “Symbol Tap Spacing” on page 3-543.

Data Types: double
Complex Number Support: Yes

tsym — Training symbols

column vector

Training symbols, specified as a column vector of length less than or equal to the length of input x. The input tsym is ignored when tf is false.

Dependencies

To enable this argument, set the Algorithm property to 'LMS' or 'RLS'.

Data Types: double

tf — Training flag

true | false

Training flag, specified as true or false. The System object starts training when tf changes from false to true (at the rising edge). The System object trains until all symbols in tsym are processed. The input tsym is ignored when tf is false.

Dependencies

To enable this argument, set the Algorithm property to 'LMS' or 'RLS' and TrainingFlagInputPort property to true.

Data Types: logical

aw — Adapt weights flag

true | false

Adapt weights flag, specified as true or false. If aw is true, the System object adapts weights. If aw is false, the System object keeps the weights unchanged.

Dependencies

To enable this argument, set the Algorithm property to 'CMA' and AdaptWeightsSource property to 'Input port'.

Data Types: logical

Output Arguments

y — Equalized symbols

column vector

Equalized symbols, returned as a column vector that has the same length as input signal `x`.

`err` — Error signal
column vector

Error signal, returned as a column vector that has the same length as input signal `x`.

`weights` — Tap weights
column vector

Tap weights, returned as a column vector that has N_{Taps} elements. N_{Taps} is equal to the sum of the NumForwardTaps and NumFeedbackTaps property values. `weights` contains the tap weights from the last tap weight update.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.DecisionFeedbackEqualizer`

isLocked	Determine if System object is in use
clone	Create duplicate System object
info	Characteristic information about the equalizer object
maxstep	Maximum step size for LMS equalizer convergence
mmseweights	Linear equalizer MMSE tap weights

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Decision Feedback Equalize BPSK-Modulated Signal

Create a BPSK modulator and an equalizer System object™, specifying a decision feedback LMS equalizer having eight forward taps, five feedback taps, and a step size of 0.03.

```
bpsk = comm.BPSKModulator;
eqdfe_lms = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',8,'NumFeedbackTaps',5,'StepSize',0.03);
```

Change the reference tap index of the equalizer.

```
eqdfe_lms.ReferenceTap = 4;
```

Build a set of test data. Receive the data by convolving the signal.

```
x = bpsk(randi([0 1],1000,1));
rxsig = conv(x,[1 0.8 0.3]);
```

Use `maxstep` to find the maximum permitted step size.

```
mxStep = maxstep(eqdfe_lms,rxsig)
```

```
mxStep = 0.1028
```

Equalize the received signal. Use the first 200 symbols as the training sequence.

```
y = eqdfe_lms(rxsig,x(1:200));
```

Decision Feedback Equalize QPSK-Modulated Signal

Apply decision feedback equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through a delayed multipath AWGN channel.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
chtaps = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
```

Generate QPSK-modulated symbols. Apply delayed multipath channel filtering and AWGN impairments to the symbols.

```
data = randi([0 M-1], numSymbols, 1);
tx = pskmod(data, M, pi/4);
rx = awgn(filter(chtaps,1,tx),25,'measured');
```

Create a decision feedback equalizer System object and display the default configuration. Adjust the reference tap to 1. Check the maximum permitted step size. Equalize the impaired symbols.

```
eq = comm.DecisionFeedbackEqualizer
```

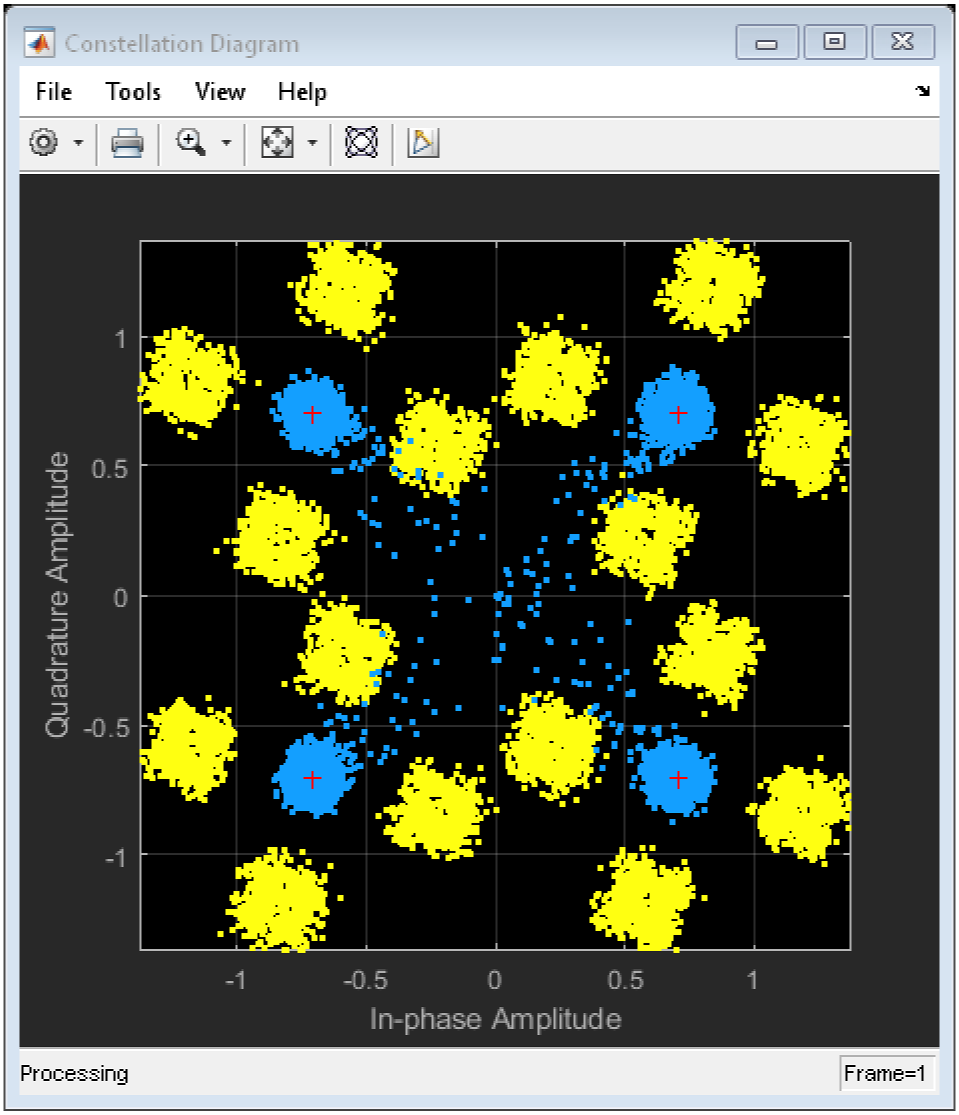
```
eq = 
  comm.DecisionFeedbackEqualizer with properties:

                Algorithm: 'LMS'
           NumForwardTaps: 5
          NumFeedbackTaps: 3
                 StepSize: 0.0100
            Constellation: [1x4 double]
             ReferenceTap: 3
               InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: true
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

```
eq.ReferenceTap = 1;

mxStep = maxstep(eq,rx)

mxStep = 0.2141
```

```
[y,err,weights] = eq(rx,tx(1:numTrainingSymbols));
```

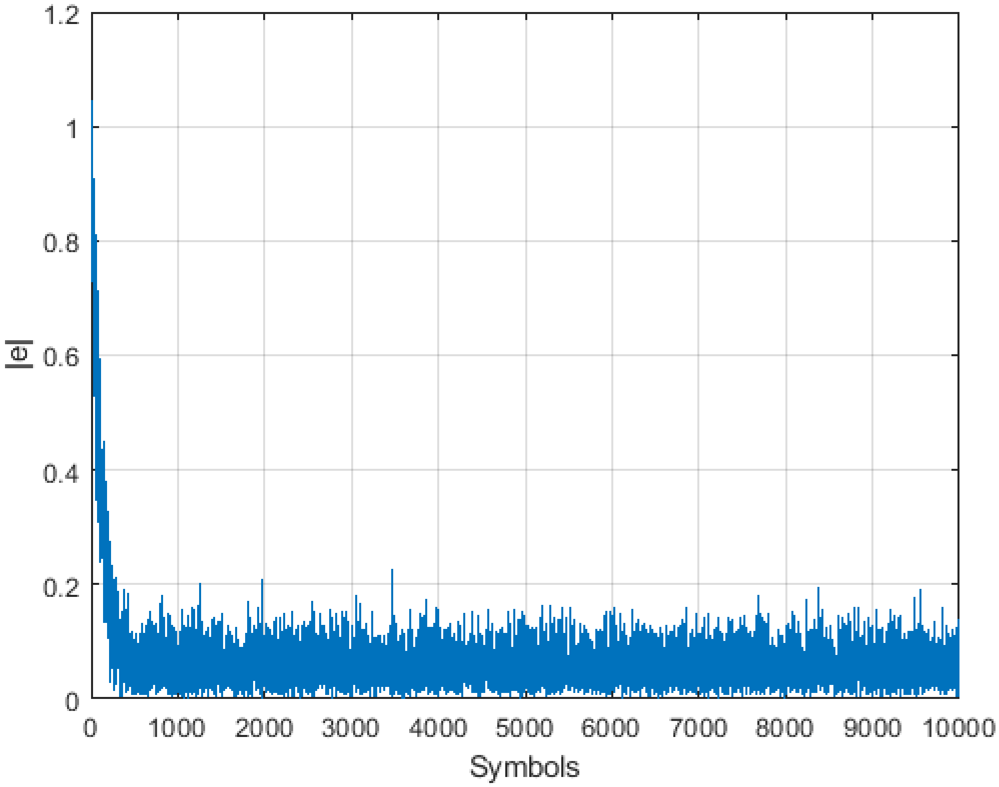
Plot the constellation of the impaired and equalized symbols.

```
constell = comm.ConstellationDiagram('NumInputPorts',2);
constell(rx,y)
```



Plot the equalizer error signal and compute the error vector magnitude of the equalized symbols.

```
plot(abs(err))
grid on; xlabel('Symbols'); ylabel('|e|')
```

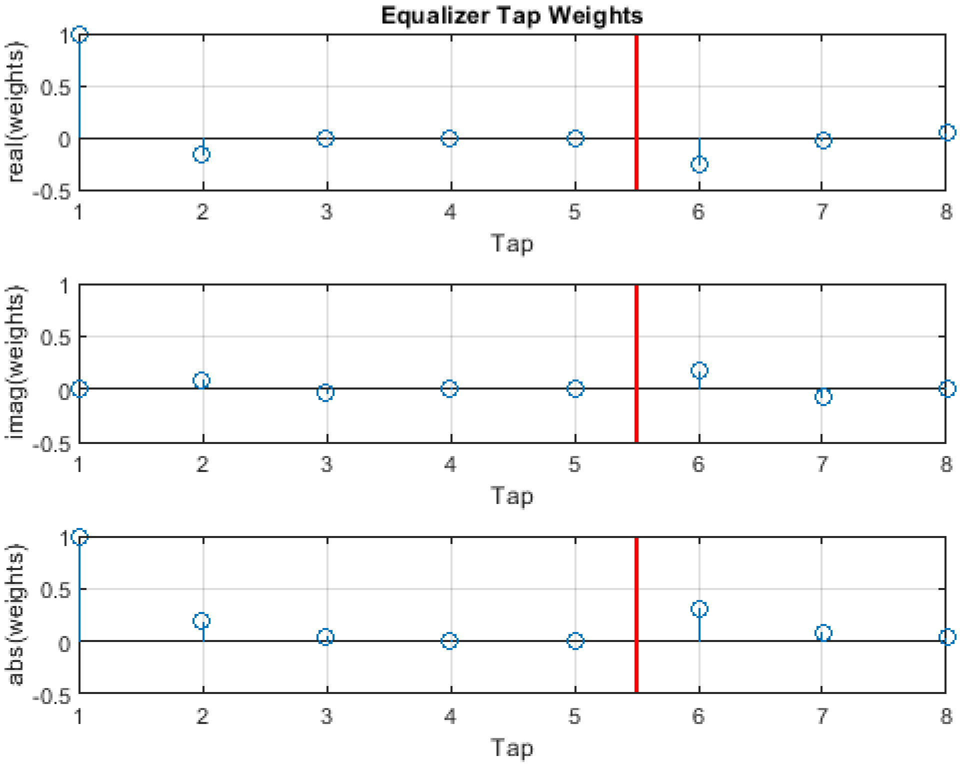


```
errevm = comm.EVM;
evm = errevm(tx,y)
```

```
evm = 10.1621
```

Plot the equalizer tap weights.

```
subplot(3,1,1); stem(real(weights)); ylabel('real(weights)'); xlabel('Tap'); grid on; axis([1 8 -
line([eq.NumForwardTaps+0.5 eq.NumForwardTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
title('Equalizer Tap Weights')
subplot(3,1,2); stem(imag(weights)); ylabel('imag(weights)'); xlabel('Tap'); grid on; axis([1 8 -
line([eq.NumForwardTaps+0.5 eq.NumForwardTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
subplot(3,1,3); stem(abs(weights)); ylabel('abs(weights)'); xlabel('Tap'); grid on; axis([1 8 -0
line([eq.NumForwardTaps+0.5 eq.NumForwardTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
```



Decision Feedback Equalize System Using Different Training Schemes

Demonstrate decision feedback equalization using the least mean squares (LMS) algorithm to recover QPSK symbols passed through an AWGN channel. Apply different equalizer training schemes and show the symbol error magnitude.

System Setup

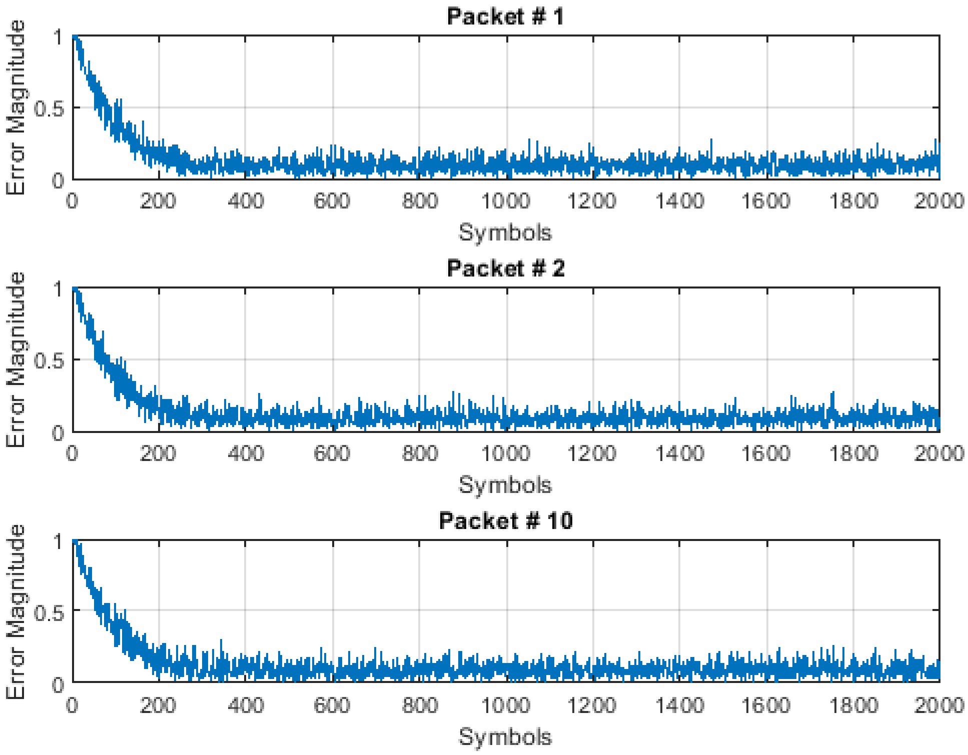
Simulate a QPSK-modulated system subject to AWGN. Transmit packets composed of 200 training symbols and 1800 random data symbols. Configure a decision feedback LMS equalizer to recover the packet data.

```
M = 4;
numTrainSymbols = 200;
numDataSymbols = 1800;
SNR = 20;
trainingSymbols = pskmod(randi([0 M-1],numTrainSymbols,1),M,pi/4);
numPkts = 10;
dfeq = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',5,'NumFeedbackTaps',4,'ReferenceTap',3,'StepSize',0.01);
```

Train the Equalizer at the Beginning of Each Packet With Reset

Process each packet using prepended training symbols. Reset the equalizer after processing each packet. Resetting the equalizer after each packet forces the equalizer to train taps with no a priori knowledge. Equalizer error signal plots for the first, second, and last packet show higher symbol errors at the start of each packet.

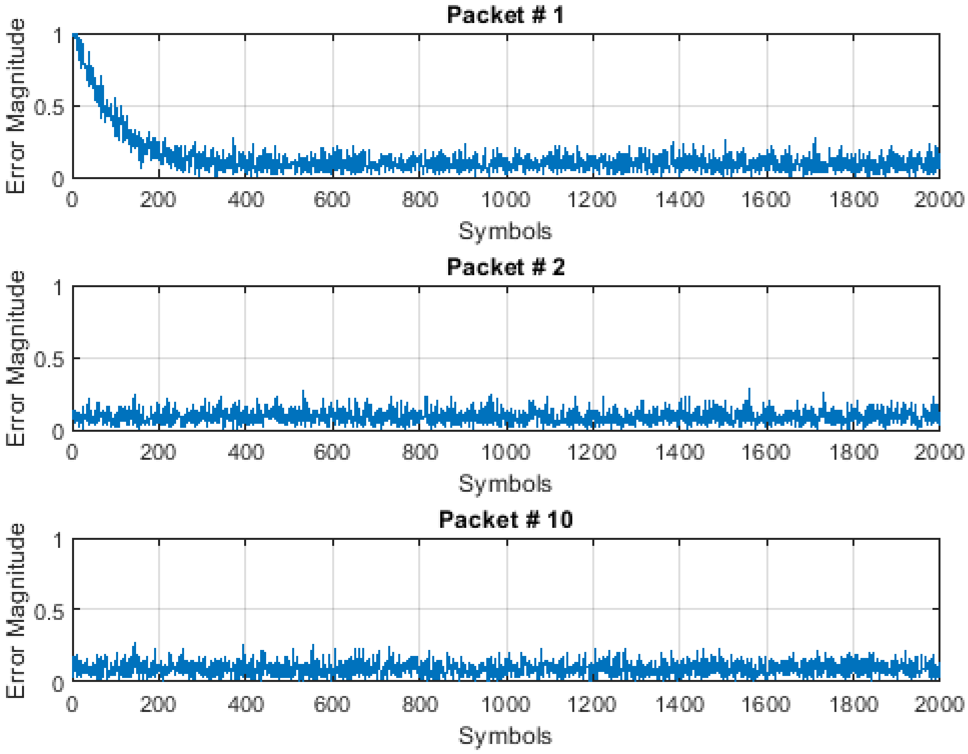
```
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    rx = awgn(packet,SNR);
    [~,err] = dfeq(rx,trainingSymbols);
    reset(dfeq)
    if (ii ==1 || ii == 2 ||ii == numPkts)
        subplot(3,1,jj)
        plot(abs(err))
        ylim([0 1])
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols');
        ylabel('Error Magnitude');
        grid on;
        jj = jj+1;
    end
end
```



Train the Equalizer at the Beginning of Each Packet Without Reset

Process each packet using prepended training symbols. Do not reset the equalizer after each packet is processed. By not resetting after each packet, the equalizer retains tap weights from training prior packets. Equalizer error signal plots for the first, second, and last packet show that after the initial training on the first packet, subsequent packets have less symbol errors at the start of each packet.

```
release(dfeq)
jj = 1;
figure
for ii = 1:numPkts
    b = randi([0 M-1],numDataSymbols,1);
    dataSym = pskmod(b,M,pi/4);
    packet = [trainingSymbols;dataSym];
    channel = 1;
    rx = awgn(packet*channel,SNR);
    [~,err] = dfeq(rx,trainingSymbols);
    if (ii ==1 || ii == 2 ||ii == numPkts)
        subplot(3,1,jj)
        plot(abs(err))
        ylim([0 1])
        title(['Packet # ',num2str(ii)])
        xlabel('Symbols');
        ylabel('Error Magnitude');
        grid on;
        jj = jj+1;
    end
end
```



Train the Equalizer Periodically

Systems with signals subject to time-varying channels require periodic equalizer training to maintain lock on the channel variations. Specify a system that has 200 symbols of training for every 1800 data symbols. Between training, the equalizer does not update tap weights. The equalizer processes 200 symbols per packet.

```
Rs = 1e6;
fd = 20;
spp = 200; % Symbols per packet
b = randi([0 M-1],numDataSymbols,1);
dataSym = pskmod(b,M,pi/4);
packet = [trainingSymbols; dataSym];
stream = repmat(packet,10,1);
tx = (0:length(stream)-1)'/Rs;
channel = exp(1i*2*pi*fd*tx);
rx = awgn(stream.*channel,SNR);
```

Set the `AdaptAfterTraining` property to `false` to stop the equalizer tap weight updates after the training phase.

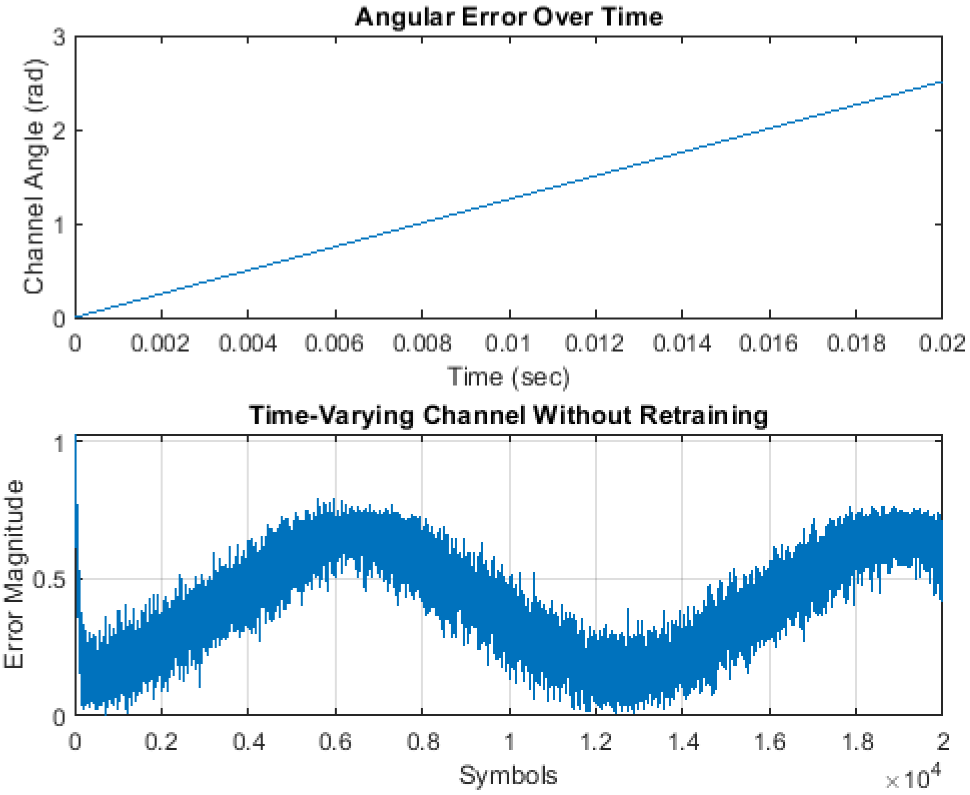
```
release(dfeq)
dfeq.AdaptAfterTraining = false
```

```
dfeq = 
  comm.DecisionFeedbackEqualizer with properties:
```

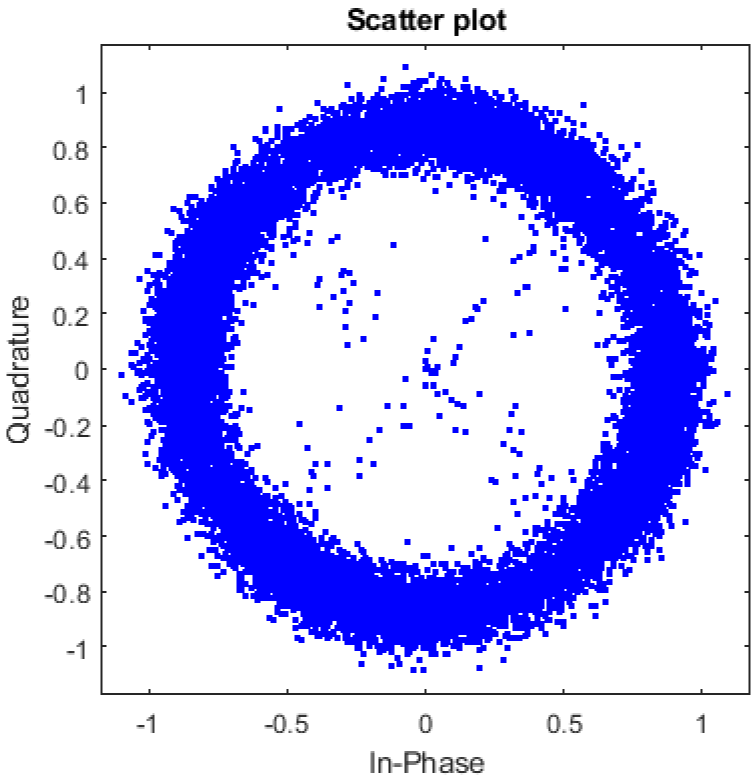
```
                Algorithm: 'LMS'
           NumForwardTaps: 5
          NumFeedbackTaps: 4
                 StepSize: 0.0100
            Constellation: [1x4 double]
             ReferenceTap: 3
               InputDelay: 0
    InputSamplesPerSymbol: 1
    TrainingFlagInputPort: false
       AdaptAfterTraining: false
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

Equalize the impaired data. Plot the angular error from the channel, the equalizer error signal, and signal constellation. As the channel varies, the equalizer output does not remove the channel effects. Also, the output constellation rotates out of sync, resulting in bit errors.

```
[y,err] = dfeq(rx,trainingSymbols);

figure
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('Time (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(err))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel Without Retraining')
```



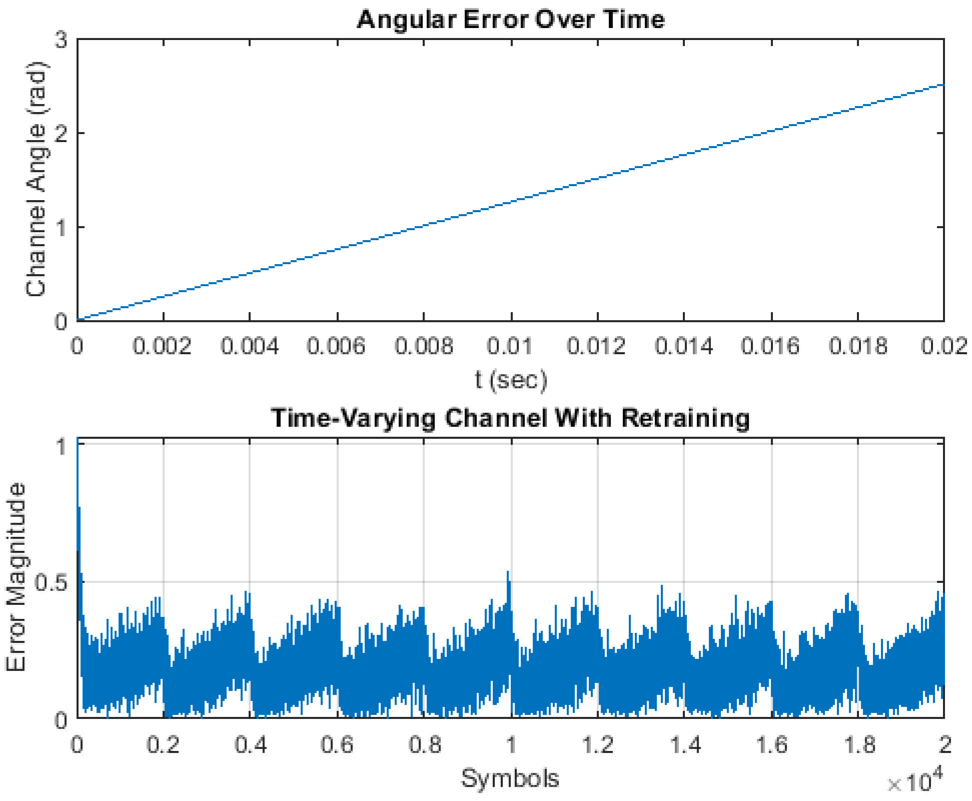
```
scatterplot(y)
```



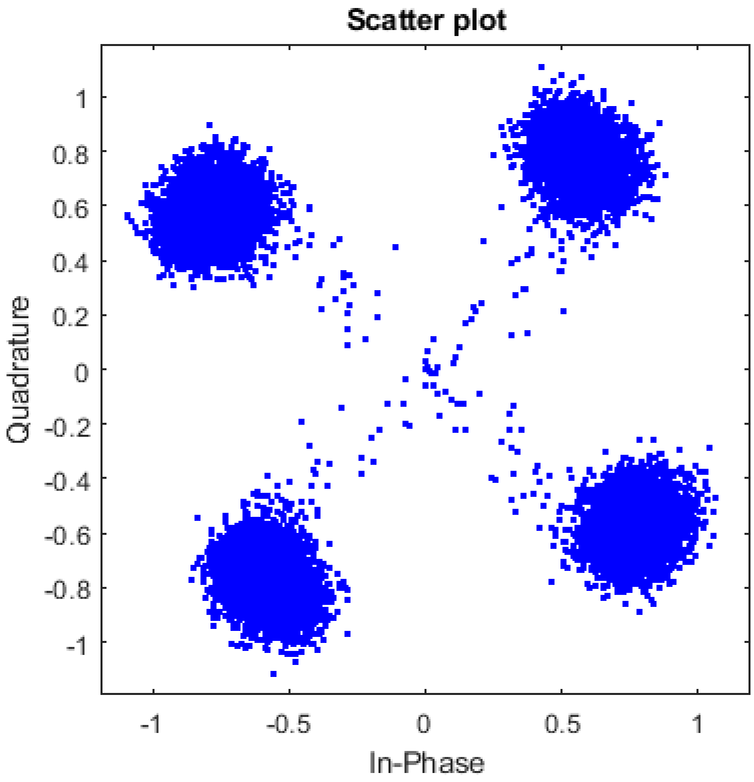
Set the `TrainingInputPort` property to `true` to configure the equalizer to retrain the taps when signaled by the `trainFlag` input. The equalizer trains only when `trainFlag` is `true`. After every 2000 symbols, the equalizer retrains the taps and keeps lock on variations of the channel. Plot the angular error from the channel, the equalizer error signal, and signal constellation. As the channel varies, the equalizer output removes the channel effects. Also, the output constellation does not rotate out of sync, and bit errors are reduced.

```
release(dfeq)
dfeq.TrainingFlagInputPort = true;
symbolCnt = 0;
numPackets = length(rx)/spp;
trainFlag = true;
trainingPeriod = 2000;
eVec = zeros(size(rx));
yVec = zeros(size(rx));
for p=1:numPackets
    [yVec((p-1)*spp+1:p*spp,1),eVec((p-1)*spp+1:p*spp,1)] = ...
        dfeq(rx((p-1)*spp+1:p*spp,1),trainingSymbols,trainFlag);
    symbolCnt = symbolCnt + spp;
    if symbolCnt >= trainingPeriod
        trainFlag = true;
        symbolCnt = 0;
    else
        trainFlag = false;
    end
end
figure
```

```
subplot(2,1,1)
plot(tx, unwrap(angle(channel)))
xlabel('t (sec)')
ylabel('Channel Angle (rad)')
title('Angular Error Over Time')
subplot(2,1,2)
plot(abs(eVec))
xlabel('Symbols')
ylabel('Error Magnitude')
grid on
title('Time-Varying Channel With Retraining')
```



```
scatterplot(yVec)
```



Decision Feedback Equalize Delayed Signal

Simulate a system with delay between the transmitted symbols and received samples. Typical systems have transmitter and receiver filters that result in a delay. This delay must be accounted for to synchronize the system. In this example, the system delay is introduced without transmit and receive filters. Decision feedback equalization, using the least mean squares (LMS) algorithm, recovers QPSK symbols.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
mpChan = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
systemDelay = dsp.Delay(20);
snr = 24;
```

Generate QPSK-modulated symbols. Apply multipath channel filtering, a system delay, and AWGN to the transmitted symbols.

```
data = randi([0 M-1],numSymbols,1);
tx = pskmod(data,M,pi/4); % OQPSK
delayedSym = systemDelay(filter(mpChan,1,tx));
rx = awgn(delayedSym,snr,'measured');
```

Create equalizer and EVM System objects. The equalizer System object specifies a decision feedback equalizer using the LMS algorithm.

```
dfeq = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',9,'NumFeedbackTaps',6,'ReferenceTap',5);
evm = comm.EVM('ReferenceSignalSource', ...
    'Estimated from reference constellation');
```

Equalize Without Adjusting Input Delay

Equalize the received symbols.

```
[y1,err1,wts1] = dfeq(rx,tx(1:numTrainingSymbols,1));
```

Find the delay between the received symbols and the transmitted symbols by using the `finddelay` function.

```
rxDelay = finddelay(tx,rx)
```

```
rxDelay = 20
```

Display the equalizer information. The latency value indicates the delay introduced by the equalizer. Calculate the total delay as the sum of `rxDelay` and the equalizer latency.

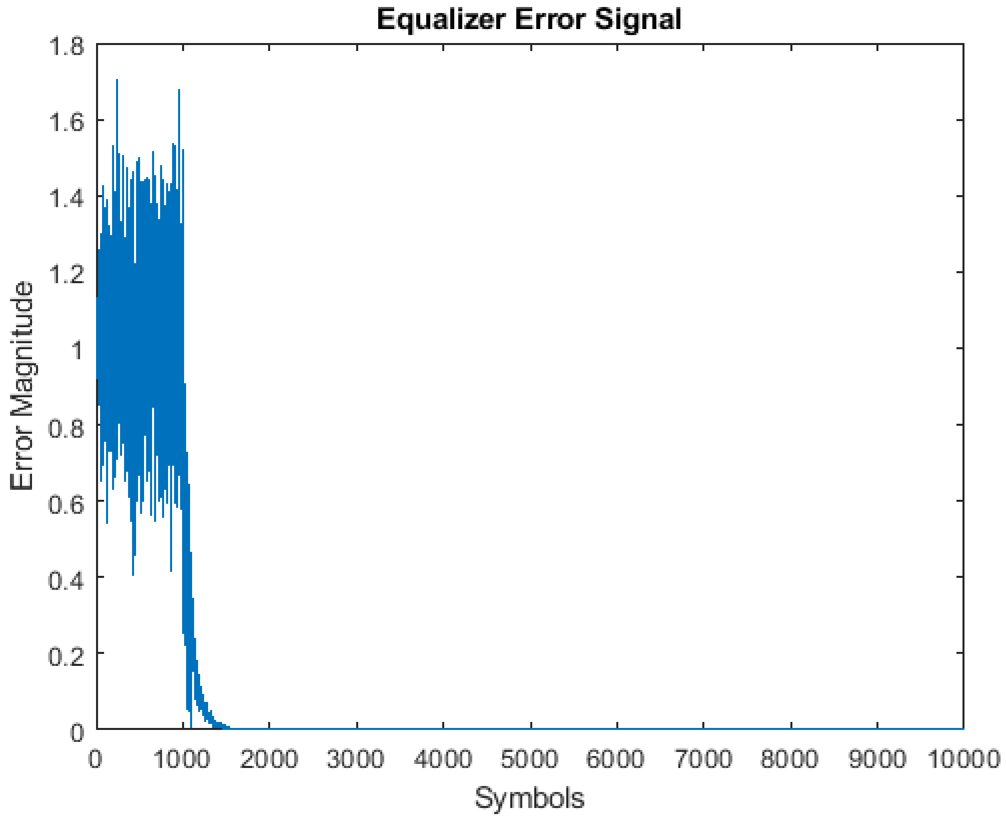
```
eqInfo = info(dfeq)
```

```
eqInfo = struct with fields:
    Latency: 4
```

```
totalDelay = rxDelay + eqInfo.Latency;
```

Until the equalizer output converges, the symbol error rate is high. Plot the error output, `err1`, to determine when the equalized output converges.

```
plot(abs(err1))
xlabel('Symbols')
ylabel('Error Magnitude')
title('Equalizer Error Signal')
```



The plot shows excessive errors for the first 2000 symbols. When demodulating symbols and computing symbol errors, account for the unconverged output and the system delay between the equalizer output and transmitted symbols.

```
dataRec1 = pskdemod(y1(2000+totalDelay:end),M,pi/4);
symErrWithDelay = symerr(data(2000:end-totalDelay),dataRec1)
```

```
symErrWithDelay = 6001
```

```
evmWithDelay = evm(y1)
```

```
evmWithDelay = 25.6868
```

The error rate and EVM are high because the receive delay was not accounted for in the equalizer System object.

Adjust Input Delay in Decision Feedback Equalizer

Equalize the received data by using the delay value to set the `InputDelay` property. Since `InputDelay` is a nontunable property, you must release the `dfeq` System object to reconfigure the `InputDelay` property. Equalize the received symbols.

```
release(dfeq)
dfeq.InputDelay = rxDelay
```

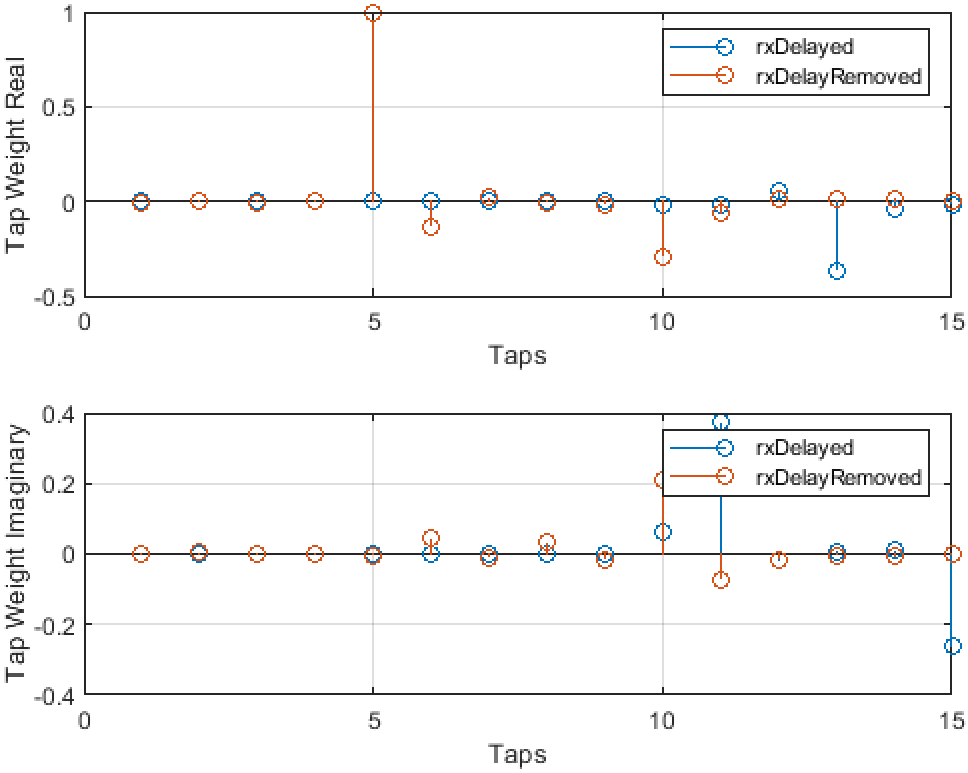
```
dfeq = 
  comm.DecisionFeedbackEqualizer with properties:
```

```
              Algorithm: 'LMS'
         NumForwardTaps: 9
        NumFeedbackTaps: 6
               StepSize: 0.0100
          Constellation: [1x4 double]
           ReferenceTap: 5
             InputDelay: 20
  InputSamplesPerSymbol: 1
  TrainingFlagInputPort: false
     AdaptAfterTraining: true
   InitialWeightsSource: 'Auto'
     WeightUpdatePeriod: 1
```

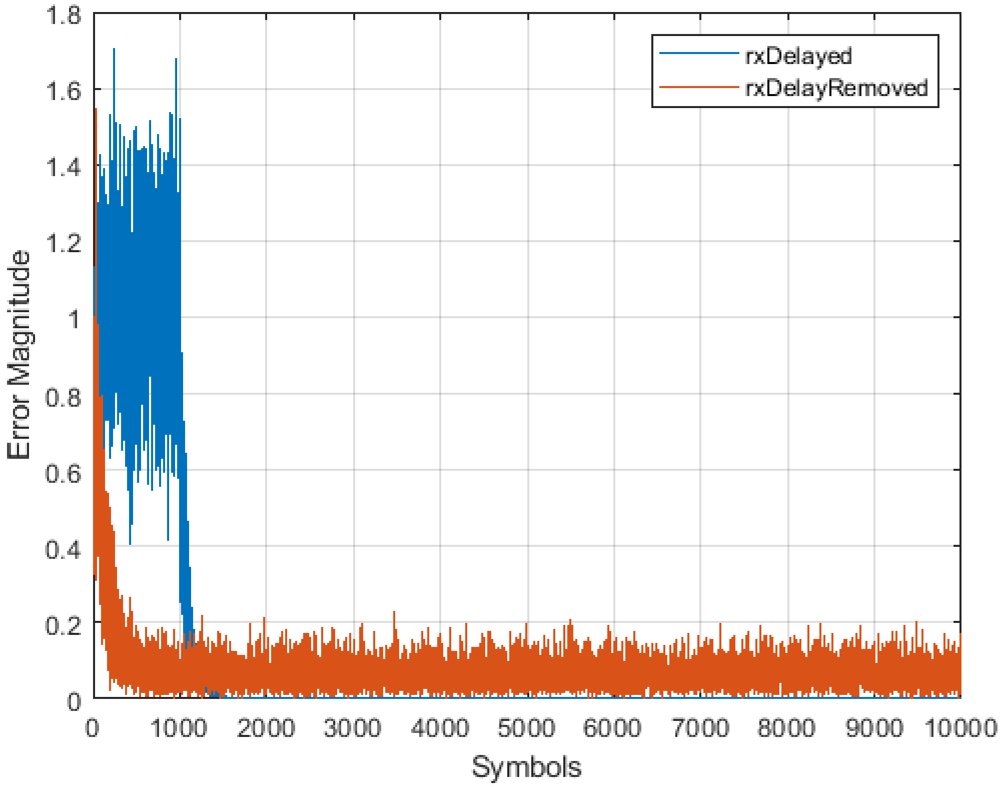
```
[y2,err2,wts2] = dfeq(rx,tx(1:numTrainingSymbols,1));
```

Plot the tap weights and equalized error magnitude. A stem plot shows the equalizer tap weights before and after the system delay is removed. A 2-D line plot shows the slower equalizer convergence for the delayed signal, as compared to the signal with the delay removed.

```
subplot(2,1,1)
stem([real(wts1),real(wts2)])
xlabel('Taps')
ylabel('Tap Weight Real')
legend('rxDelayed','rxDelayRemoved')
grid on
subplot(2,1,2)
stem([imag(wts1),imag(wts2)])
xlabel('Taps')
ylabel('Tap Weight Imaginary')
legend('rxDelayed','rxDelayRemoved')
grid on
```



```
figure
plot([abs(err1),abs(err2)])
xlabel('Symbols')
ylabel('Error Magnitude')
legend('rxDelayed','rxDelayRemoved')
grid on
```



Plot error output of the equalized signals, `rxDelayed` and `rxDelayRemoved`. For the signal that has the delay removed, the equalizer converges during the 1000 symbol training period. When demodulating symbols and computing symbol errors, to account for the unconverged output and the system delay between the equalizer output and transmitted symbols, skip the first 500 symbols. Reconfiguring the equalizer to account for the system delay enables better equalization of the signal, and reduces symbol errors and the EVM.

```
eqInfo = info(dfeq)
```

```
eqInfo = struct with fields:
    Latency: 4
```

```
totalDelay = rxDelay + eqInfo.Latency;
dataRec2 = pskdemod(y2(500+totalDelay:end),M,pi/4);
symErrDelayRemoved = symerr(data(500:end-totalDelay),dataRec2)
```

```
symErrDelayRemoved = 0
```

```
evmDelayRemoved = evm(y2(500+totalDelay:end))
```

```
evmDelayRemoved = 7.5147
```

Decision Feedback Equalize Symbols Using EVM-Based Training

Recover QPSK symbols with a decision equalizer, using the constant modulus algorithm (CMA) and EVM-based taps training. When using blind equalizer algorithms, such as CMA, you can train the equalizer taps using the `AdaptWeights` property to start and stop training. Use helper functions to generate plots and apply phase correction.

Initialize system variables.

```
rng(123456);
M = 4; % QPSK
numSymbols = 100;
numPackets = 5000;
refTap = 3;
nFwdTaps = 5;
nFdbkTaps = 4;
ttlTaps = nFwdTaps + nFdbkTaps;
raylChan = comm.RayleighChannel('PathDelays',[0 1], ...
    'AveragePathGains',[0 -12],'MaximumDopplerShift',1e-5);
SNR = 50;
adaptWeights = true;
```

Create the equalizer and EVM System objects. The equalizer System object specifies a decision feedback equalizer using the CMA adaptive algorithm. Call the helper function to initialize figure plots.

```
dfeq = comm.DecisionFeedbackEqualizer('Algorithm','CMA', ...
    'NumForwardTaps',nFwdTaps,'NumFeedbackTaps',nFdbkTaps,'ReferenceTap',refTap, ...
    'StepSize',0.03,'AdaptWeightsSource','Input port')
```

```
dfeq = 
  comm.DecisionFeedbackEqualizer with properties:

                Algorithm: 'CMA'
           NumForwardTaps: 5
          NumFeedbackTaps: 4
                 StepSize: 0.0300
            Constellation: [1x4 double]
             ReferenceTap: 3
    InputSamplesPerSymbol: 1
       AdaptWeightsSource: 'Input port'
     InitialWeightsSource: 'Auto'
       WeightUpdatePeriod: 1
```

```
info(dfeq)
```

```
ans = struct with fields:
    Latency: 2
```

```
evm = comm.EVM('ReferenceSignalSource', ...
    'Estimated from reference constellation');
[errPlot,evmPlot,scatSym,adaptState] = initFigures(numPackets,ttlTaps);
```

Equalization Loop

Follow these steps to implement the equalization loop.

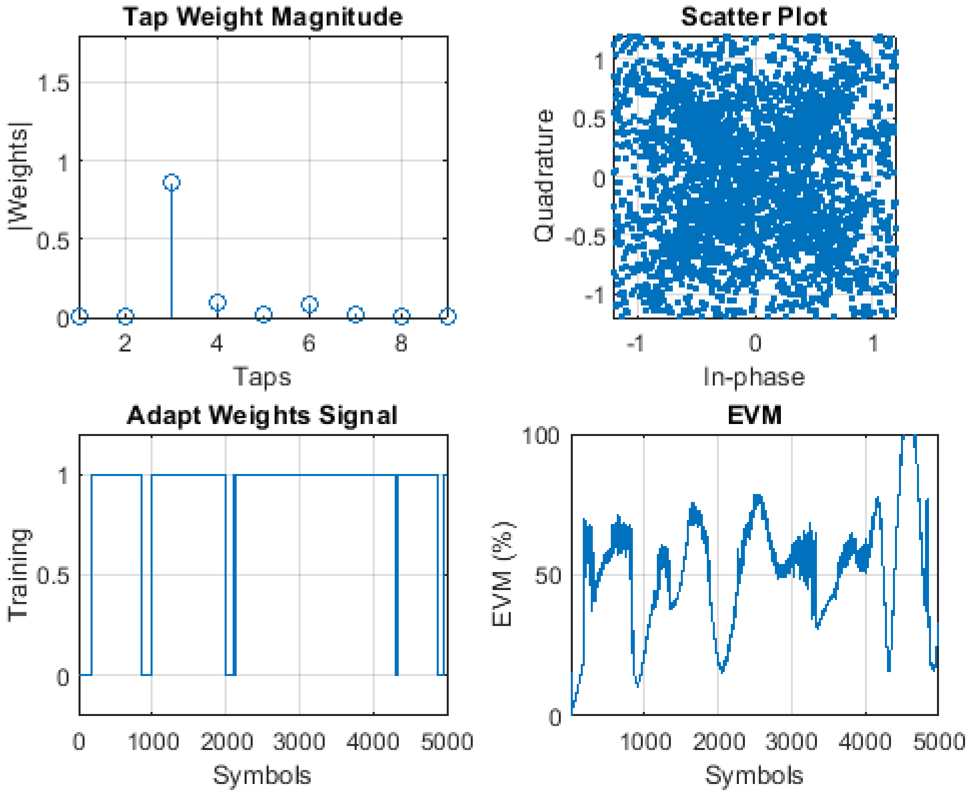
1. Generate OQPSK data packets.
2. Apply Rayleigh fading and AWGN to the transmission data.

3 Apply equalization to the received data and phase correction to the equalizer output.

4 Estimate the EVM and toggle the `adaptWeights` flag to `true` or `false` based on the EVM level.

5 Update the figure plots.

```
for p=1:numPackets
    data = randi([0 M-1],numSymbols,1);
    tx = pskmod(data,M,pi/4);
    rx = awgn(raylChan(tx),SNR);
    rxDelay = finddelay(rx,tx);
    [y,err,wts] = dfeq(rx,adaptWeights);
    y = phaseCorrection(y);
    evmEst = evm(y);
    adaptWeights = (evmEst > 20);

    updateFigures(errPlot,evmPlot,scatSym,adaptState, ...
        wts,y(end),evmEst,adaptWeights,p,numPackets)
end
```



```
rxDelay
```

```
rxDelay = 0
```

The figure plots show that, as the EVM varies, the equalizer toggles in and out of decision-directed weight adaptation mode.

Helper Functions

This helper function initializes figures that show a quad plot of simulation results.

```
function [errPlot,evmPlot,scatter,adaptState] = initFigures(numPkts,ttlTaps)
yVec = nan(numPkts,1);
evmVec = nan(numPkts,1);
wVec = zeros(ttlTaps,1);
adaptVec = nan(numPkts,1);

figure
subplot(2,2,1)
evmPlot = stem(wVec);
grid on; axis([1 ttlTaps 0 1.8])
xlabel('Taps'); ylabel('|Weights|'); title('Tap Weight Magnitude')

subplot(2,2,2)
scatter = plot(yVec, '.');
axis square; axis([-1.2 1.2 -1.2 1.2]); grid on
xlabel('In-phase'); ylabel('Quadrature'); title('Scatter Plot');
subplot(2,2,3)
adaptState = plot(adaptVec);
grid on; axis([0 numPkts -0.2 1.2])
ylabel('Training'); xlabel('Symbols'); title('Adapt Weights Signal')
subplot(2,2,4)
errPlot = plot(evmVec);
grid on; axis([1 numPkts 0 100])
xlabel('Symbols'); ylabel('EVM (%)'); title('EVM')
end
```

This helper function updates the figures.

```
function updateFigures(errPlot,evmPlot,scatSym, ...
    adaptState,w,y,evmEst,adaptWts,p,numFrames)
persistent yVec evmVec adaptVec

if p == 1
    yVec = nan(numFrames,1);
    evmVec = nan(numFrames,1);
    adaptVec = nan(numFrames,1);
end

yVec(p) = y;
evmVec(p) = evmEst;
adaptVec(p) = adaptWts;

errPlot.YData = abs(evmVec);
evmPlot.YData = abs(w);
scatSym.XData = real(yVec);
scatSym.YData = imag(yVec);
adaptState.YData = adaptVec;
drawnow limitrate
end
```

This helper function applies phase correction.

```
function y = phaseCorrection(y)
a = angle(y((real(y) > 0) & (imag(y) > 0)));
```

```
a(a < 0.1) = a(a < 0.1) + pi/2;
theta = mean(a) - pi/4;
y = y * exp(-1i*theta);
end
```

Decision Feedback Equalize Packetized Signals in Fading Environments

Recover QPSK symbols in fading environments with a decision feedback equalizer, using the least mean squares (LMS) algorithm. Use the `reset` object function to equalize independent packets. Use helper functions to generate plots. This example also shows symbol-based processing and frame-based processing.

Setup

Initialize system variables, create the equalizer System object, and initialize the plot figures.

```
M = 4; % QPSK
numSym = 1000;
numTrainingSym = 100;
numPackets = 5;
refTap = 5;
nFwdTaps = 9;
nFdbkTaps = 4;
ttlTaps = nFwdTaps + nFdbkTaps;
stepsz = 0.01;
ttlNumSym = numSym + numTrainingSym;
raylChan = comm.RayleighChannel('PathDelays',[0 1], ...
    'AveragePathGains',[0 -9], ...
    'MaximumDopplerShift',0, ...
    'PathGainsOutputPort',true);
SNR = 35;
rxVec = zeros(ttlNumSym,numPackets);
txVec = zeros(ttlNumSym,numPackets);
yVec = zeros(ttlNumSym,1);
eVec = zeros(ttlNumSym,1);

dfeq1 = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',nFwdTaps,'NumFeedbackTaps',nFdbkTaps,'ReferenceTap',refTap, ...
    'StepSize',stepsz,'TrainingFlagInputPort',true);

[errPlot,wStem,hStem,scatPlot] = initFigures(ttlNumSym,ttlTaps, ...
    raylChan.AveragePathGains);
```

Symbol-Based Processing

For symbol-based processing, provide one symbol at the input of the equalizer. Reset the equalizer state and channel after processing each packet.

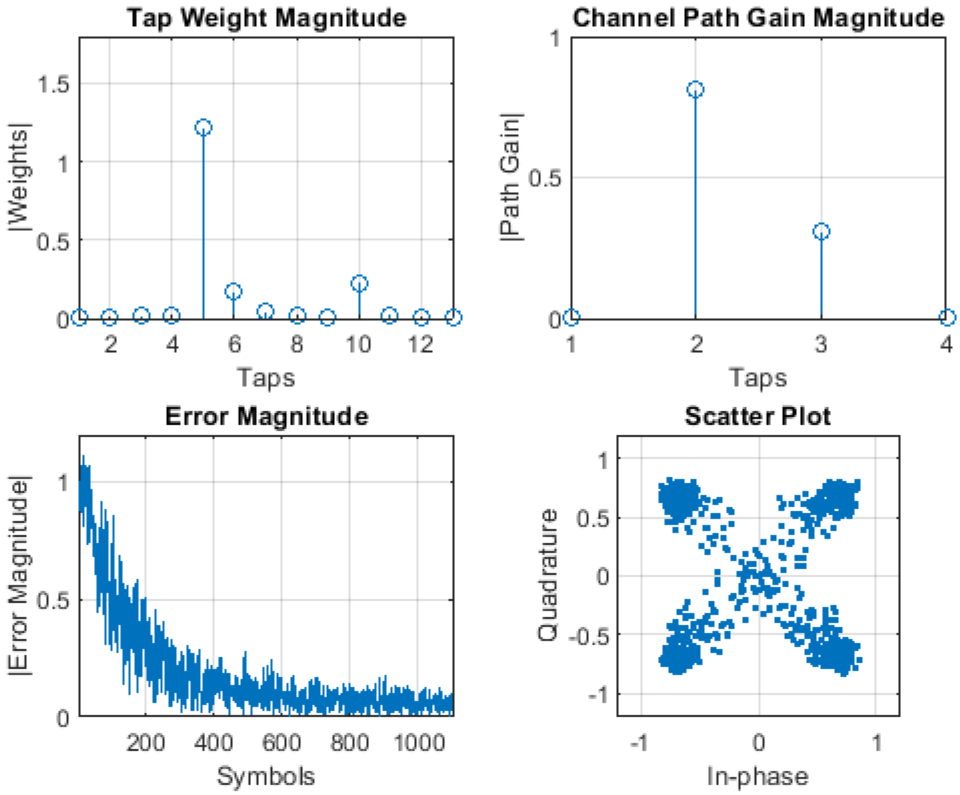
```
for p = 1:numPackets
    trainingFlag = true;
    for q=1:ttlNumSym
        data = randi([0 M-1],1,1);
        tx = pskmod(data,M,pi/4);
        [xc,pg] = raylChan(tx);
        rx = awgn(xc,25);
        [y,err,wts] = dfeq1(rx,tx,trainingFlag);
```

Disable training after processing `numTrainingSym` training symbols.

```
        if q == numTrainingSym
            trainingFlag = false;
        end
        updateFigures(errPlot,wStem,hStem,scatPlot,err,wts,y,pg,q,ttlNumSym);
        txVec(q,p) = tx;
        rxVec(q,p) = rx;
    end
```

After processing each packet, reset the channel System object to get a new realization of channel taps and the equalizer System object to restore the default taps weights.

```
    reset(raylChan)
    reset(dfeq1)
end
```



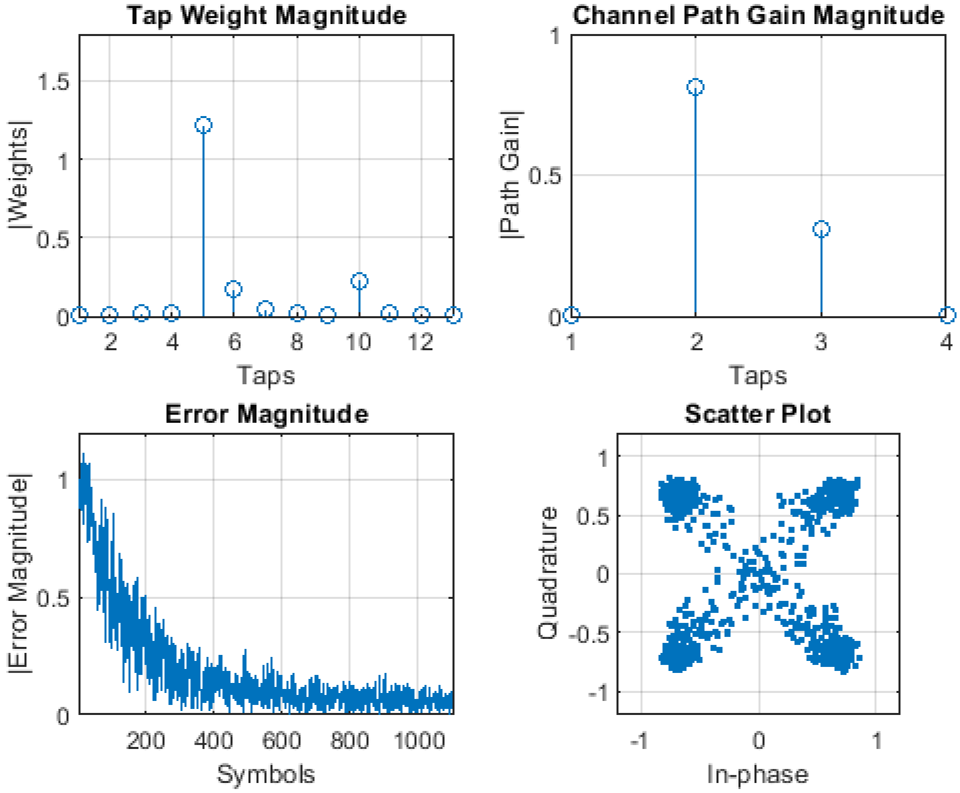
Packet-Based Processing

For packet-based processing, provide one packet at the input of the equalizer. Each packet contains `ttlNumSym` symbols. Because the training duration is less than the packet length, you do not need to specify the start-training input.

```
yVecPkt = zeros(ttlNumSym,numPackets);
errVecPkt = zeros(ttlNumSym,numPackets);
wgtVecPkt = zeros(ttlTaps,numPackets);
dfeq2 = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',nFwdTaps,'NumFeedbackTaps',nFdbkTaps,'ReferenceTap',refTap, ...
    'StepSize',stepsz);
for p = 1:numPackets
    [yVecPkt(:,p),errVecPkt(:,p),wgtVecPkt(:,p)] = ...
        dfeq2(rxVec(:,p),txVec(1:numTrainingSym,p));
    for q=1:ttlNumSym
        updateFigures(errPlot,wStem,hStem,scatPlot, ...
            errVecPkt(q,p),wgtVecPkt(:,p),yVecPkt(q,p),pg,q,ttlNumSym);
    end
```

After processing each packet, reset the channel System object to get a new realization of channel taps and the equalizer System object to restore the default taps weights.

```
    reset(raylChan)
    reset(dfeq2)
end
```



Helper Functions

This helper function initializes the figures.

```
function [errPlot,wStem,hStem,scatPlot] = initFigures(ttlNumSym,ttlTap,pg)
yVec = nan(ttlNumSym,1);
eVec = nan(ttlNumSym,1);
wVec = zeros(ttlTap,1);
figure;
subplot(2,2,1);
wStem = stem(wVec);
axis([1 ttlTap 0 1.8]); grid on
```

```
xlabel('Taps'); ylabel('|Weights|'); title('Tap Weight Magnitude')
subplot(2,2,2);
hStem = stem([0 abs(pg) 0]);
grid on;
xlabel('Taps'); ylabel('|Path Gain|'); title('Channel Path Gain Magnitude')
subplot(2,2,3);
errPlot = plot(eVec);
axis([1 ttlNumSym 0 1.2]); grid on
xlabel('Symbols'); ylabel('|Error Magnitude|'); title('Error Magnitude')
subplot(2,2,4);
scatPlot = plot(yVec,'.');
axis square; axis([-1.2 1.2 -1.2 1.2]); grid on;
xlabel('In-phase'); ylabel('Quadrature'); title(sprintf('Scatter Plot'));
end
```

This helper function updates the figures.

```
function updateFigures(errPlot,wStem,hStem,scatPlot, ...
    err,wts,y,pg,p,ttlNumSym)
persistent yVec eVec
if p == 1
    yVec = nan(ttlNumSym,1);
    eVec = nan(ttlNumSym,1);
end
yVec(p) = y;
eVec(p) = abs(err);
errPlot.YData = abs(eVec);
wStem.YData = abs(wts);
hStem.YData = [0 abs(pg) 0];
scatPlot.XData = real(yVec);
scatPlot.YData = imag(yVec);
drawnow limitrate
end
```

More About

Symbol Tap Spacing

You can configure the equalizer to operate as a symbol-spaced equalizer or as a fractional symbol-spaced equalizer.

- To operate the equalizer at a symbol-spaced rate, specify the number of samples per symbol as 1. Symbol-rate equalizers have taps spaced at the symbol duration. Symbol-rate equalizers are sensitive to timing phase.
- To operate the equalizer at a fractional symbol-spaced rate, specify the number of input samples per symbol as an integer greater than 1 and provide an input signal oversampled at that sampling rate. Fractional symbol-spaced equalizers have taps spaced at an integer fraction of the input symbol duration. Fractional symbol-spaced equalizers are not sensitive to timing phase.

Algorithms

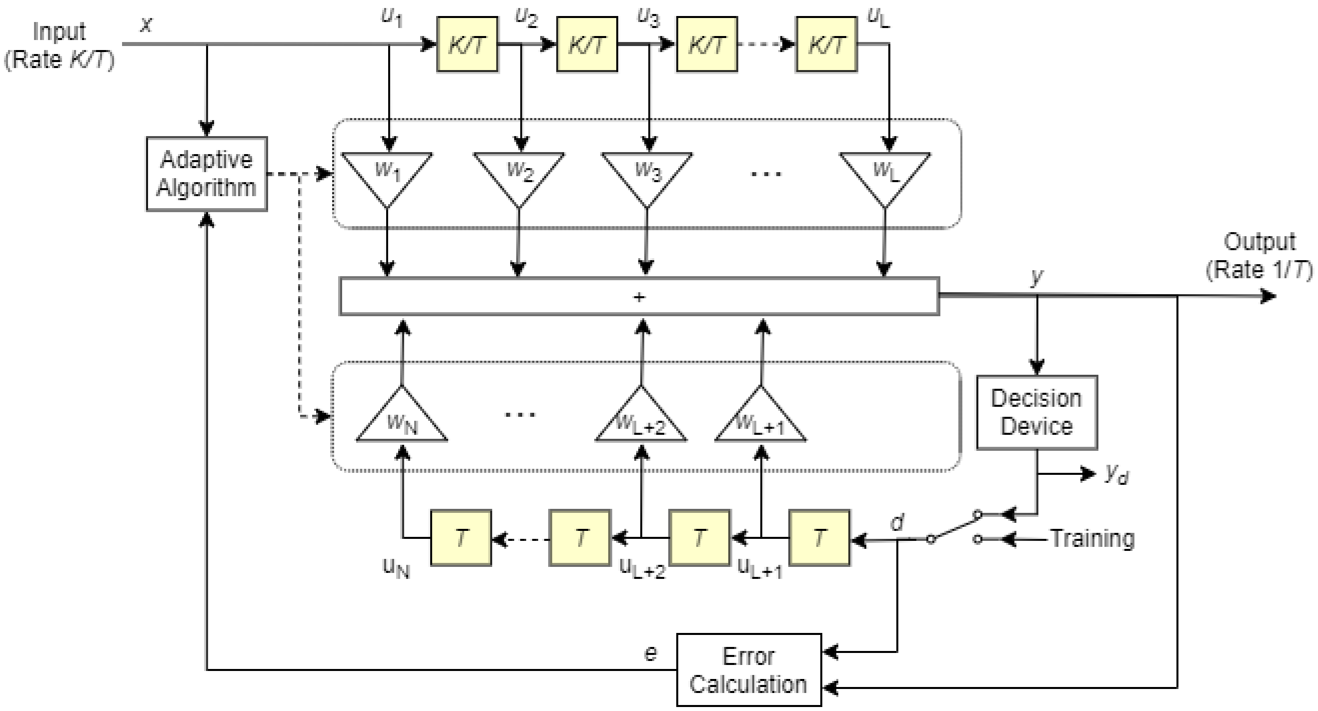
Decision Feedback Equalizers

A decision feedback equalizer (DFE) is a nonlinear equalizer that reduces intersymbol interference (ISI) in frequency-selective channels. If a null exists in the frequency response of a channel, DFEs do not enhance the noise. A DFE consists of a tapped delay line that stores samples from the input signal and contains a forward filter and a feedback filter. The forward filter is similar to a linear equalizer. The feedback filter contains a tapped delay line whose inputs are the decisions made on the equalized signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period.

DFEs can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, *K*, is 1. The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, *K*, is an integer greater than 1. Typically, *K* is 4 for fractional symbol-spaced equalizers. The output sample rate is 1/*T* and the input sample rate is *K/T*. Tap weight updating occurs at the output rate.

This schematic shows a fractional symbol-spaced DFE with a total of *N* weights, a symbol period of *T*, and *K* samples per symbol. The filter has *L* forward weights and *N-L* feedback weights. The forward filter is at the top, and the feedback filter is at the bottom. If *K* is 1, the result is a symbol-spaced DFE instead of a fractional symbol-spaced DFE.



In each symbol period, the equalizer receives *K* input samples at the forward filter and one decision or training sample at the feedback filter. The equalizer then outputs a weighted sum of the values in the forward and feedback delay lines and updates the weights to prepare for the next symbol period.

Note The algorithm for the Adaptive Algorithm block in the schematic jointly optimizes the forward and feedback weights. Joint optimization is especially important for convergence in the recursive least square (RLS) algorithm.

For more information, see “Equalization”.

Least Mean Square (LMS) Algorithm

For the LMS algorithm, in the previous schematic, *w* is a vector of all weights w_i, and *u* is a vector of all inputs u_i. Based on the current set of weights, the LMS algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + (StepSize)\ ue^*$.

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed when using the LMS adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Recursive Least Square (RLS) Algorithm

For the RLS algorithm, in the previous schematic, *w* is the vector of all weights w_i, and *u* is the vector of all inputs u_i. Based on the current set of inputs, *u*, and the inverse correlation matrix, *P*, the RLS algorithm first computes the Kalman gain vector, *K*, as

$$K = \frac{Pu}{(ForgettingFactor) + u^H Pu}.$$

The forgetting factor used by the adaptive algorithm is specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalized output signal to be less stable. *H* denotes the Hermitian transpose. Based on the current inverse correlation matrix, the new inverse correlation matrix is

$$P_{\text{new}} = \frac{(1 - Ku^H)P_{\text{current}}}{ForgettingFactor}.$$

Based on the current set of weights, the RLS algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + K^*e$.

The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Constant Modulus Algorithm (CMA)

For the CMA adaptive algorithm, in the previous schematic, *w* is the vector of all weights w_i, and *u* is the vector of all inputs u_i. Based on the current set of weights, the CMA adaptive algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + (StepSize)\ u^*e$.

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed by the CMA adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = y(R - |y|^2)$, where *R* is a constant related to the signal constellation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`comm.LinearEqualizer` | `comm.MLSEEqualizer`

Blocks
Decision Feedback Equalizer

Topics
“Equalization”
“Adaptive Equalizers”

Introduced in R2019a

comm.EVM

Package: `comm`

Measure error vector magnitude

Description

The `comm.EVM` (error vector magnitude) System object measures the modulator or demodulator performance of an impaired signal.

To measure error vector magnitude:

1. Define and set up your EVM object. See “Construction” on page 3-547.
2. Call `step` to measure the modulator or demodulator performance according to the properties of `comm.EVM`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`EVM = comm.EVM` creates an error vector magnitude object, `EVM`. This object measures the amount of impairment in a modulated signal.

`EVM = comm.EVM(Name,Value)` creates an `EVM` object with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Example: `EVM = comm.EVM('ReferenceSignalSource','Estimated from reference constellation')` creates an object, `EVM`, that measures the RMS EVM of a received signal by using a reference constellation.

Properties

`Normalization`

Normalization method

Normalization method used in EVM calculation, specified as one of the following: `'Average reference signal power'` (default), `'Average constellation power'`, or `'Peak constellation power'`.

`AverageConstellationPower`

Average constellation power

Average constellation power, specified in watts as a positive real scalar. This property is available when `Normalization` is `'Average constellation power'`. The default is `1`.

PeakConstellationPower

Peak constellation power

Peak constellation power, specified in watts as a positive real scalar. This property is available when `Normalization` is `'Peak constellation power'`. The default is `1`.

ReferenceSignalSource

Reference signal source

Reference signal source, specified as either `'Input port'` (default) or `'Estimated from reference constellation'`. To provide an explicit reference signal against which the input signal is measured, set this property to `'Input port'`. To measure the EVM of the input signal against a reference constellation, set this property to `'Estimated from reference constellation'`.

ReferenceConstellation

Reference constellation

Reference constellation, specified as a vector. This property is available when the `ReferenceSignalSource` property is `'Estimated from reference constellation'`.

The default is `[0.7071 - 0.7071i; -0.7071 - 0.7071i; -0.7071 + 0.7071i; 0.7071 + 0.7071i]`, which corresponds to a standard QPSK constellation. You can derive constellation points by using modulation functions or objects. For example, to derive the reference constellation for a 16-QAM signal, you can use `qammod(0:15,16)`.

MeasurementIntervalSource

Measurement interval source

Measurement interval source, specified as one of the following: `'Input length'` (default), `'Entire history'`, `'Custom'`, or `'Custom with periodic reset'`. This property affects the RMS and maximum EVM outputs only.

- To calculate EVM using only the current samples, set this property to `'Input length'`.
- To calculate EVM for all samples, set this property to `'Entire history'`.
- To calculate EVM over an interval you specify and to use a sliding window, set this property to `'Custom'`.
- To calculate EVM over an interval you specify and to reset the object each time the measurement interval is filled, set this property to `'Custom with periodic reset'`.

MeasurementInterval

Measurement interval

Measurement interval over which the EVM is calculated, specified in samples as a real positive integer. This property is available when `MeasurementIntervalSource` is `'Custom'` or `'Custom with periodic reset'`. The default is `100`.

AveragingDimensions

Averaging dimensions

Averaging dimensions over which to average the EVM measurements, specified as an integer or row vector of integers with element values in the range [1, 3]. For example, to average across the rows, set this property to `2`. The default is `1`.

The object supports variable-size inputs over the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must remain constant between calls of the object. For example, if the input has size `[4 3 2]` and `Averaging dimensions` is `[1 3]`, the output size is `[1 3 1]`, and the second dimension must remain fixed at `3`.

`MaximumEVMOutputPort`

Maximum EVM measurement output port

Maximum EVM measurement output port, specified as a logical scalar. To create an output port for maximum EVM measurements, set this property to `true`. The default is `false`.

`XPercentileEVMOutputPort`

X-percentile EVM measurement output port

X-percentile EVM measurement output port, specified as a logical scalar. To create an output port for *X*-percentile EVM measurements, set this property to `true`. The *X*-percentile EVM measurements persist until you reset the object. These measurements are calculated by using all of the input frames since the last reset. The default is `false`.

`XPercentileValue`

X-percentile value

X-percentile value below which *X*% of the EVM measurements fall, specified as a real scalar from `0` to `100`. This property is available when `XPercentileEVMOutputPort` is `true`. The default is `95`.

`SymbolCountOutputPort`

Symbol count output port

Symbol count output port, specified as a logical scalar. To output the number of accumulated symbols used to calculate the *X*-percentile EVM measurements, set this property to `true`. This property is available when `XPercentileEVMOutputPort` is `true`. The default is `false`.

Methods

reset	Reset states of EVM measurement object
step	Measure error vector magnitude

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Measure EVM of Noisy 16-QAM Modulated Signal

Create an EVM object. Configure it using name-value pairs to output maximum EVM, 90th percentile EVM, and the symbol count.

```
evm = comm.EVM('MaximumEVMOutputPort',true,...
    'XPercentileEVMOutputPort',true, 'XPercentileValue',90,...
    'SymbolCountOutputPort',true);
```

Generate random data symbols. Apply 16-QAM modulation. The modulated signal serves as the reference for the subsequent EVM measurements.

```
data = randi([0 15],1000,1);
refSym = qammod(data,16,'UnitAveragePower',true);
```

Pass the modulated signal through an AWGN channel.

```
rxSym = awgn(refSym,20);
```

Measure the EVM of the noisy signal.

```
[rmsEVM,maxEVM,pctEVM,numSym] = evm(refSym,rxSym)
```

```
rmsEVM = 9.8775

maxEVM = 26.8385

pctEVM = 14.9750

numSym = 1000
```

Estimate Received EVM

Generate filtered QAM data and pass it through an AWGN channel. Compute the symbol error rate, and estimate the EVM of the received signal.

Create channel and filter System objects™.

```
M = 16;
refConst = qammod(0:M-1,M);
channel = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)',...
    'SNR',15,'SignalPower',10);

txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',4);
rxfilter = comm.RaisedCosineReceiveFilter('InputSamplesPerSymbol',4, ...
    'DecimationFactor',4);
```

Create an EVM object to output RMS and maximum EVM measurements.

```
evm = comm.EVM('MaximumEVMOutputPort',true, ...
    'ReferenceSignalSource','Estimated from reference constellation', ...
    'ReferenceConstellation',refConst);
```

Create an error rate object and account for the signal delay through the transmit and receive filters. For a filter, the group delay is equal to 1/2 of the `FilterSpanInSymbols` property.

```
rxd = (txfilter.FilterSpanInSymbols + rxfilter.FilterSpanInSymbols)/2;
errorRate = comm.ErrorRate('ReceiveDelay',rxd);
```

Perform the following channel operations:

- Generate random data symbols.
- Apply 16-QAM modulation.
- Filter the modulated data through a raised cosine Tx filter.
- Pass the transmitted signal through an AWGN channel.
- Filter the received data through a raised cosine Rx filter.
- Demodulate the filtered data.

```
txData = randi([0 15],1000,1);
modData = qammod(txData,M);
txSig = txfilter(modData);
rxSig = channel(txSig);
filtSig = rxfilter(rxSig);
rxData = qamdemod(filtSig,M);
```

Calculate the error statistics and display the symbol error rate.

```
errStats = errorRate(txData,rxData);
symErrRate = errStats(1)
```

```
symErrRate = 0.0222
```

Measure and display the received RMS EVM and maximum EVM values. Take the filter delay into account by deleting the first `rxd+1` symbols. Because there are symbol errors, the EVM may not be totally accurate.

```
[rmsEVM,maxEVM] = evm(filtSig(rxd+1:end))
```

```
rmsEVM = 17.2966

maxEVM = 40.1595
```

Measure EVM Using Reference Constellation

Generate random data symbols, and apply 8-PSK modulation.

```
d = randi([0 7],2000,1);
txSig = pskmod(d,8,pi/8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,30);
```

Create an EVM object. Measure the RMS EVM using the transmitted signal as the reference.

```
evm = comm.EVM;
rmsEVM1 = evm(txSig,rxSig);
```

Release the EVM object. Configure the object to estimate the EVM of the received signal against a reference constellation.

```
release(evm)
evm.ReferenceSignalSource = 'Estimated from reference constellation';
evm.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Measure the RMS EVM using only the received signal as an input. Verify that it matches the result obtained when using a reference signal.

```
rmsEVM2 = evm(rxSig);
[rmsEVM1 rmsEVM2]
```

```
ans = 1×2

    3.1524    3.1524
```

Measure EVM Using Custom Measurement Interval

Measure the EVM of a noisy 8-PSK signal using two types of custom measurement intervals. Display the results.

Set the number of frames, M, and the number of subframes per frame, K.

```
M = 2;
K = 5;
```

Set the number of symbols in a subframe. Calculate the corresponding frame length.

```
sfLen = 100;
frmLen = K*sfLen
```

```
frmLen = 500
```

Create an EVM object. Configure the object to use a custom measurement interval equal to the frame length.

```
evm1 = comm.EVM('MeasurementIntervalSource','Custom', ...
    'MeasurementInterval',frmLen);
```

Configure the object to measure EVM using an 8-PSK reference constellation.

```
evm1.ReferenceSignalSource = 'Estimated from reference constellation';
evm1.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Create an EVM object, and configure it use a 500-symbol measurement interval with a periodic reset. Configure the object to measure EVM using an 8-PSK reference constellation.

```
evm2 = comm.EVM('MeasurementIntervalSource','Custom with periodic reset', ...
    'MeasurementInterval',frmLen);
evm2.ReferenceSignalSource = 'Estimated from reference constellation';
evm2.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Initialize the EVM and signal-to-noise arrays.

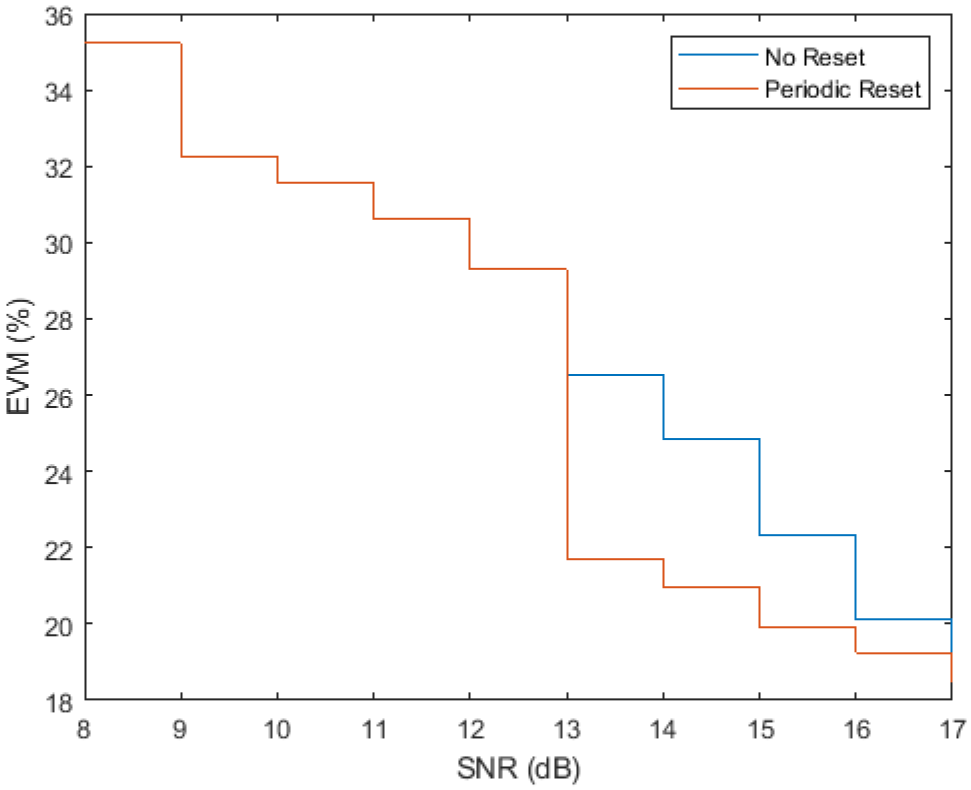
```
rmsEVM1 = zeros(K,M);
rmsEVM2 = zeros(K,M);
snrdB = zeros(K,M);
```

Measure the EVM for a noisy 8-PSK signal using both objects. The SNR increases by 1 dB from subframe to subframe. For `evm1`, the 500 most recent symbols are used to compute the estimate. In this case, a sliding window is used so that an entire frame of data is always processed. For `evm2`, the symbols are cleared each time a new frame is encountered.

```
for m = 1:M
    for k = 1:K
        data = randi([0 7],sfLen,1);
        txSig = pskmod(data,8,pi/8);
        snrdB(k,m) = k+(m-1)*K+7;
        rxSig = awgn(txSig,snrdB(k,m));
        rmsEVM1(k,m) = evm1(rxSig);
        rmsEVM2(k,m) = evm2(rxSig);
    end
end
```

Display the EVM measured using the two approaches. The windowing used in the first case provides an averaging across the subframes. In the second case, the EVM object resets after the first frame so that the calculated EVM values more accurately reflect the current SNR.

```
stairs(snrdB(:),[rmsEVM1(:) rmsEVM2(:)])
xlabel('SNR (dB)')
ylabel('EVM (%)')
legend('No Reset','Periodic Reset')
```



Measure EVM Across Different Dimensions

Create OFDM modulator and demodulator objects.

```
ofdmmod = comm.OFDMModulator('FFTLength',32,'NumSymbols',4);
ofdmdemod = comm.OFDMDemodulator('FFTLength',32,'NumSymbols',4);
```

Determine the number of subcarriers and symbols in the OFDM signal.

```
ofdmDims = info(ofdmmod);
numSC = ofdmDims.DataInputSize(1)
```

```
numSC = 21
```

```
numSym = ofdmDims.DataInputSize(2)
```

```
numSym = 4
```

Generate random symbols and apply QPSK modulation.

```
msg = randi([0 3],numSC,numSym);
modSig = pskmod(msg,4,pi/4);
```

OFDM modulate the QPSK signal. Pass the signal through an AWGN channel. Demodulate the noisy signal.

```
txSig = ofdmmod(modSig);
rxSig = awgn(txSig,10,'measured');
demodSig = ofdmdemod(rxSig);
```

Create an EVM object, where the result is averaged over the subcarriers. Measure the EVM. There are four entries corresponding to each of the 4 OFDM symbols.

```
evm = comm.EVM('AveragingDimensions',1);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 1×4

   27.4354   23.6279   22.6772   23.1699
```

Overwrite the EVM object, where the result is averaged over the OFDM symbols. Measure the EVM. There are 21 entries corresponding to each of the 21 subcarriers.

```
evm = comm.EVM('AveragingDimensions',2);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 21×1

   28.8225
   17.8536
   18.6809
   20.8872
   22.3532
   24.7197
   30.1954
   33.4899
   36.2847
   21.4230
```

```
      ⋮
```

Measure the EVM and average over both the subcarriers and the OFDM symbols.

```
evm = comm.EVM('AveragingDimensions',[1 2]);
rmsEVM = evm(demodSig,modSig)
```

```
rmsEVM = 24.2986
```

Plot Time-Varying EVM for OFDM Signal

Calculate and plot the EVM of an OFDM signal. The signal consists of two packets separated by an interval.

Create System objects to:

- OFDM modulate a signal
- Introduce phase noise
- Plot time-varying signals

```
ofdmmod = comm.OFDMModulator('FFTLength',256,'NumSymbols',2);

pnoise = comm.PhaseNoise('Level',-60,'FrequencyOffset',20,'SampleRate',1000);

tscope = timescope('YLabel','EVM (%)','YLimits',[0 40], ...
    'SampleRate',1000,'TimeSpanSource','Property','TimeSpan',1.2, ...
    'ShowGrid',true);
```

Create an EVM object. To generate a time-varying estimate of the EVM, set the `AveragingDimensions` property to `2`.

```
evm = comm.EVM('MaximumEVMOutputPort',false, ...
    'ReferenceSignalSource','Input port', ...
    'AveragingDimensions',2);
```

Determine the input data dimensions of the OFDM modulator.

```
modDims = info(ofdmmod)
```

```
modDims = 

  struct with fields:

    DataInputSize: [245 2]
       OutputSize: [544 1]
```

Create QPSK-modulated random data for the first packet. Apply OFDM modulation.

```
data = randi([0 3],modDims.DataInputSize);
qpskSig = pskmod(data,4,pi/4);
txSig1 = ofdmmod(qpskSig);
```

Create a second data packet.

```
data = randi([0 3],modDims.DataInputSize);
qpskSig = pskmod(data,4,pi/4);
txSig2 = ofdmmod(qpskSig);
```

Concatenate the two packets and include an interval in which nothing is transmitted.

```
txSig = [txSig1; zeros(112,1); txSig2];
```

Apply I/Q amplitude and phase imbalance to the transmitted signal.

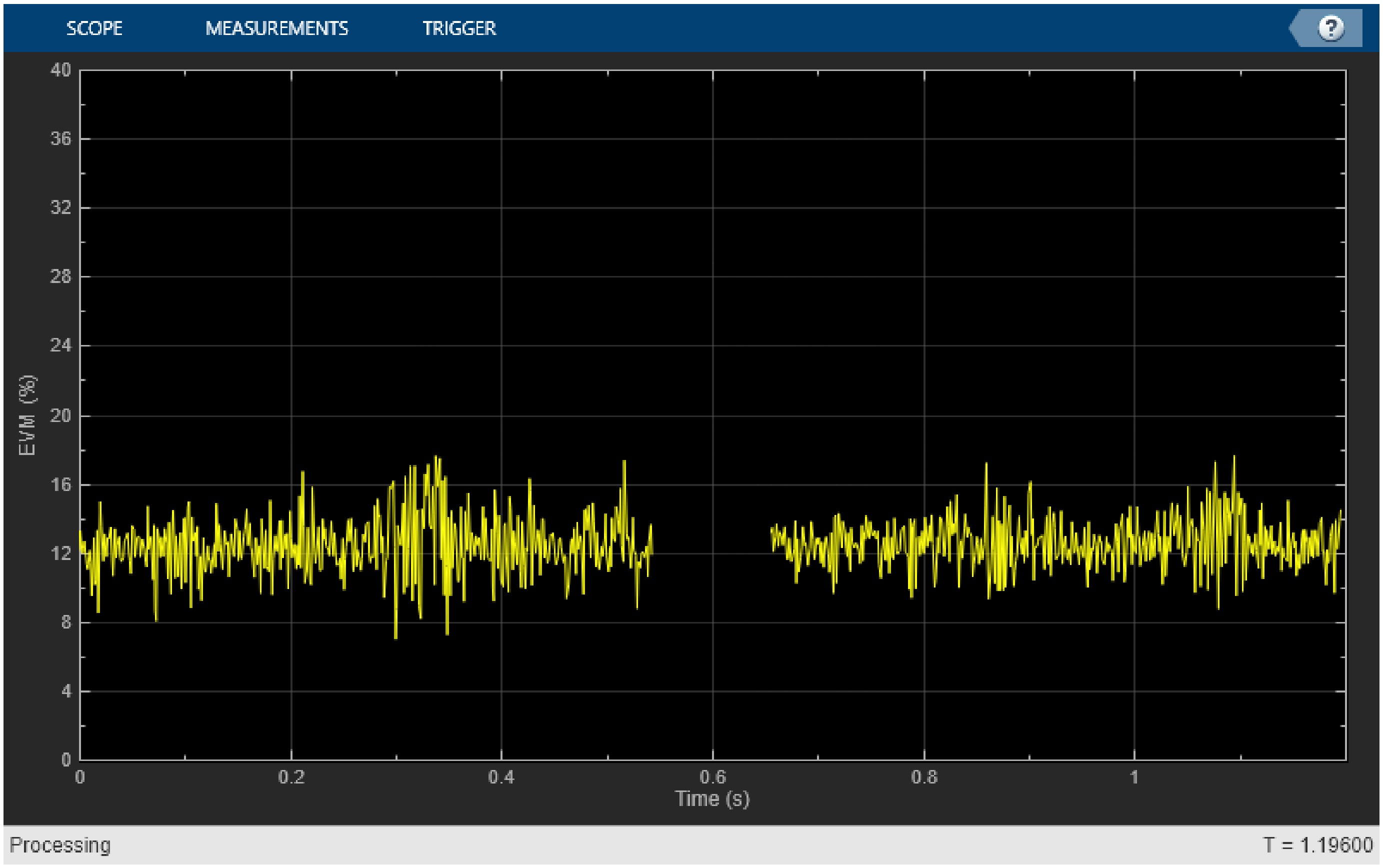
```
rxSigIQimb = iqimbal(txSig,2,5);
```

Apply phase noise.

```
rxSig = pnoise(rxSigIQimb);
```

Measure the EVM of the received signal, and plot its time-varying EVM.

```
e = evm(txSig,rxSig);
tscope(e)
```



Algorithms

Both the EVM block and the EVM object provide three normalization methods. You can normalize measurements according to the average power of the reference signal, average constellation power, or peak constellation power. Different industry standards follow one of these normalization methods.

The block or object calculates the RMS EVM value differently for each normalization method.

EVM Normalization Method	Algorithm
Reference signal	$EVM_{RMS} = \sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{avg}}}}$
Peak power	$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{max}}}}$

Where:

- $e_k = e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k = In-phase measurement of the kth symbol in the burst
- Q_k = Quadrature phase measurement of the kth symbol in the burst
- N = Input vector length
- P_{avg} = The value for **Average constellation power**
- P_{max} = The value for **Peak constellation power**
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The max EVM is the maximum EVM value in a frame or $\mathrm{EVM}_{\mathrm{max}} = \max_{k \in [1, \ldots, N]} \{\mathrm{EVM}_k\}$, where k is the kth symbol in a burst of length N.

The definition for EVM_k varies depending upon which normalization method you select for computing measurements. The block or object supports these algorithms.

EVM Normalization	Algorithm
Reference signal	$EVM_k = \sqrt{\frac{e_k}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$\mathrm{EVM}_k = 100\sqrt{\frac{e_k}{P_{\mathrm{avg}}}}$
Peak power	$\mathrm{EVM}_k = 100\sqrt{\frac{e_k}{P_{\mathrm{max}}}}$

The block or object computes the X-percentile EVM by creating a histogram of all the incoming EVM_k values. The output provides the EVM value below which X% of the EVM values fall.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.CCDF` | `comm.MER`

Introduced in R2012a

reset

System object: `comm.EVM`
Package: `comm`

Reset states of EVM measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `EVM` object, `H`.

step

System object: comm.EVM
Package: comm

Measure error vector magnitude

Syntax

```
RMSEVM = step(EVM,REFSYM,RXSYM)
RMSEVM = step(EVM,RXSYM)
[___,MAXEVM] = step(___)
[___,XEVM] = step(___)
[___,NUMSYM] = step(___)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

RMSEVM = step(EVM,REFSYM,RXSYM) returns the measured root-mean-square EVM, RMSEVM, of the received signal RXSYM, based on reference signal REFSYM. EVM values are measured as a percentage.

REFSYM and RXSYM inputs are complex column vectors of equal dimensions and data type. The data type can be double, single, signed integer, or signed fixed point with power-of-two slope and zero bias. All outputs of the object are of data type double. To set the interval over which the EVM is measured, use the MeasurementIntervalSource and MeasurementInterval properties.

RMSEVM = step(EVM,RXSYM) returns the measured EVM of received signal RXSYM based on a reference signal specified in the ReceivedConstellation property.

[___,MAXEVM] = step(___) returns the maximum EVM, MAXEVM, given either of the two previous syntaxes.

To return the maximum EVM value, set the MaximumEVMOutputPort property to true. To set the interval over which the maximum EVM is measured, use the MeasurementIntervalSource and MeasurementInterval properties.

[___,XEVM] = step(___) returns the *X*-percentile EVM, XEVM.

To return the X-percentile EVM, set the XPercentileEVMOutputPort property to true. XEVM is the EVM below which *X*% of the measurements fall, where *X* is set by the XPercentileValue property. XEVM is measured using all the input frames since the last reset.

[___,NUMSYM] = step(___) returns the number of symbols, NUMSYM, used to calculate the *X*-percentile EVM.

To return `NUMSYM`, set the `SymbolCountOutputPort` property to `true`. `NUMSYM` is measured using all the input frames since the last reset.

Note `EVM` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.EyeDiagram

Package: `comm`

(To be removed) Display eye diagram of time-domain signals

Note will be removed in a future release. To display the eye diagram of a signal, use the `eyediagram` function instead. For more details on the recommended workflow, see “Compatibility Considerations”.

Description

The `comm.EyeDiagram` System object displays multiple traces of a modulated signal to produce an eye diagram. You can use the object to reveal the modulation characteristics of the signal, such as the effects of pulse shaping or channel distortions. The eye diagram can measure signal characteristics and plot horizontal and vertical bathtub curves when the jitter and noise comply with the dual-Dirac model [1].

To display the eye diagram of an input signal:

1. Create the `comm.EyeDiagram` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
ed = comm.EyeDiagram
ed = comm.EyeDiagram(Name,Value)
```

Description

`ed = comm.EyeDiagram` creates an eye diagram System object with default property values.

`ed = comm.EyeDiagram(Name,Value)` sets properties using one or more name-value pair argument. Enclose each property name in single quotes. Unspecified properties have default values.

Example: `comm.EyeDiagram('SampleRate',2,'DisplayMode','2D color histogram')`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Name — Title of eye diagram window
'Eye Diagram' (default) | character vector

Title of eye diagram window, specified as a character vector.

Tunable: Yes

Data Types: char

SampleRate — Sample rate of input signal
1 (default) | positive real-valued scalar

Sample rate of the input signal in hertz, specified as a positive real-valued scalar.

Data Types: double

SamplesPerSymbol — Number of samples per symbol
8 (default) | positive integer

Number of samples per symbol, specified as a positive integer.

Tunable: Yes

Data Types: double

SampleOffset — Number of samples to skip before plotting the first point
0 (default) | nonnegative integer

Number of samples to skip before plotting the first point, specified as a nonnegative integer. To avoid irregular behavior, specify the offset to be less than the product of the SamplesPerSymbol and SymbolsPerTrace properties.

Tunable: Yes

Data Types: double

SymbolsPerTrace — Number of symbols per trace
2 (default) | positive integer

Number of symbols per trace, specified as a positive integer. To obtain eye measurements and visualize bathtub curves, use the default value of 2.

Tunable: Yes

Data Types: double

TracesToDisplay — Number of traces to display
40 (default) | positive integer

Number of traces to display, specified as a positive integer.

Tunable: Yes

Dependencies

To enable this property, set DisplayMode property to 'Line plot'.

Data Types: `double`

`DisplayMode` — Eye diagram display mode

`'Line plot'` (default) | `'2D color histogram'`

Eye diagram display mode, specified as one of these values.

- `'Line plot'` — Overlay traces by plotting one line for each of the last TracesToDisplay traces.
- `'2D color histogram'` — Display a color gradient that shows how often the input matches different time and amplitude values.

Tunable: Yes

Data Types: `char`

`EnableMeasurements` — Option to enable eye diagram measurements

`false` (default) | `true`

Option to enable eye diagram measurements, specified as `true` or `false`. Set this property to `true` to display the measurement pane and calculations in the eye diagram.

Tunable: Yes

Data Types: `logical`

`ShowBathtub` — Option to enable visualization of bathtub curves

`'None'` (default) | `'Horizontal'` | `'Vertical'` | `'Both'`

Option to enable visualization of bathtub curves, specified as `'None'`, `'Horizontal'`, `'Vertical'`, or `'Both'`.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `char`

`OverlayHistogram` — Histogram overlay

`'None'` (default) | `'Jitter'` | `'Noise'`

Histogram overlay, specified as `'None'`, `'Jitter'`, or `'Noise'`.

- To overlay a horizontal histogram on the eye diagram, set this property to `'Jitter'`.
- To overlay a vertical histogram on the eye diagram, set this property to `'Noise'`.
- To display no histogram overlay, set this property to `'None'`.

Tunable: Yes

Dependencies

To enable this property, set the DisplayMode property to `'2D color histogram'` and EnableMeasurements property to `true`.

Data Types: `char`

`DecisionBoundary` — Amplitude level threshold

`0` (default) | real-valued scalar

Amplitude level threshold in volts, specified as a real-valued scalar. This property separates the different signaling regions for horizontal (jitter) histograms. Jitter histograms reset when this property changes.

For non-return-to-zero (NRZ) signals, set `DecisionBoundary` to 0. For return-to-zero (RZ) signals, set `DecisionBoundary` to half the maximum amplitude.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`EyeLevelBoundaries` — Time range for calculating eye levels

`[40 60]` (default) | two-element row vector

Time range for calculating eye levels, specified as a two-element row vector. Specify the vector values as percentages of the symbol duration.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`RiseFallThresholds` — Amplitude levels of the rise and fall transitions

`[10 90]` (default) | two-element row vector

Amplitude levels of the rise and fall transitions, specified as a two-element row vector. Specify the vector values as percentages of the eye amplitude. The crossing histograms of the rise and fall thresholds reset when this property changes.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`Hysteresis` — Amplitude tolerance of the horizontal crossings

`0` (default) | real-valued scalar

Amplitude tolerance of the horizontal crossings in volts, specified as a real-valued scalar. Increase this value to provide more tolerance to spurious crossings due to noise. Jitter and the rise and fall histograms reset when this property changes.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`BERThreshold` — BER used for eye measurements

`1e-12` (default) | scalar in the range [0, 0.5]

Bit error rate (BER) used for eye measurements, specified as a scalar in the range [0, 0.5]. The System object uses this value to measure the random jitter, the total jitter, horizontal eye openings, and vertical eye openings.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`BathtubBER` — BER values used to calculate openings of bathtub curves

`[0.5 10.^-(1:12)]` (default) | vector

BER values used to calculate the openings of bathtub curves, specified as a vector of elements in the range [0, 0.5]. Horizontal and vertical eye openings are calculated for each of the values specified by this property.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `true` and ShowBathtub property to `'Both'`, `'Horizontal'`, or `'Vertical'`.

Data Types: `double`

`MeasurementDelay` — Duration of initial data discarded from measurements

`0` (default) | nonnegative scalar

Duration of initial data discarded from measurements in seconds, specified as a nonnegative scalar.

Dependencies

To enable this property, set the EnableMeasurements property to `true`.

Data Types: `double`

`OversamplingMethod` — Oversampling method

`'None'` (default) | `'Input interpolation'` | `'Histogram interpolation'`

Oversampling method, specified as `'None'`, `'Input interpolation'`, or `'Histogram interpolation'`.

To plot eye diagrams as quickly as possible, set `OversamplingMethod` to `'None'`. The drawback to not oversampling is that the plots look pixelated when the number of symbols per trace is small.

To create smoother, less-pixelated plots using a small number of symbols per trace, set `OversamplingMethod` to`'Input interpolation'` or `'Histogram interpolation'`. In this case, `'Input interpolation'` is the faster interpolation method and produces good results when the signal-to-noise ratio (SNR) is high. With a low SNR, this oversampling method is not recommended because it introduces a bias to the centers of the histogram ranges. `'Histogram` `interpolation'` is not as fast as the other techniques, but it provides good results even when the SNR is low.

Tunable: Yes

Dependencies

To enable this property, set the DisplayMode property to `'2D color histogram'`.

Data Types: `char`

`ColorScale` — Color scale of the histogram

`'Linear'` (default) | `'Logarithmic'`

Color scale of the histogram, specified as `'Linear'` or `'Logarithmic'`. Change this property if certain areas of the histogram include a disproportionate number of points. Use the `'Logarithmic'` option for eye diagrams with sharp peaks, where the signal repetitively matches specific time and amplitude values.

Tunable: Yes

Dependencies

To enable this property, set the DisplayMode property to `'2D color histogram'`.

Data Types: `char`

`ColorFading` — Color fading

`false` (default) | `true`

Color fading, specified as `true` or `false`. To fade the points in the display as the interval of time after they are first plotted increases, set this property to `true`. This animation resembles an oscilloscope.

Tunable: Yes

Dependencies

To enable this property, set the DisplayMode property to `'Line plot'`.

Data Types: `logical`

`ShowImaginaryEye` — Show imaginary signal component

`false` (default) | `true`

Show imaginary signal component, specified as `true` or `false`. To view the imaginary or quadrature component of the input signal, set this property to `true`.

Tunable: Yes

Dependencies

To enable this property, set the EnableMeasurements property to `false`.

Data Types: `logical`

`YLimits` — *Y*-axis limits

`[-1.1 1.1]` (default) | two-element row vector

Y-axis limits of the eye diagram in volts, specified as a two-element vector. The first element corresponds to *ymin* and the second to *ymax*. The second element must be greater than the first.

Tunable: Yes

Data Types: `double`

`ShowGrid` — Option to enable grid display

`false` (default) | `true`

Option to enable grid display on the eye diagram, specified as `true` or `false`. To display a grid on the eye diagram, set this property to `true`.

Tunable: Yes

Data Types: `logical`

`Position` — Scope window position

four-element row vector

Scope window position in pixels, specified as a four-element row vector of the form [*left bottom width height*].

Tunable: Yes

Data Types: `double`

Usage

Syntax

```
ed(x)
```

Description

`ed(x)` displays and analyzes input signal `x` in an eye diagram.

Input Arguments

`x` — Input signal

vector | matrix

Input signal to be analyzed and displayed in the eye diagram, specified as a vector or matrix. `x` can be either a real or complex vector, or a real two-column matrix.

Data Types: `double`
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.EyeDiagram`

show	Show scope window
hide	Hide scope window
horizontalBathtub	Horizontal bathtub curve
verticalBathtub	Vertical bathtub curve
jitterHistogram	Jitter histogram
noiseHistogram	Noise histogram
measurements	Measure eye diagram parameters

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Eye Diagram of Filtered QPSK Signal

Specify the sample rate and the number of output samples per symbol parameters.

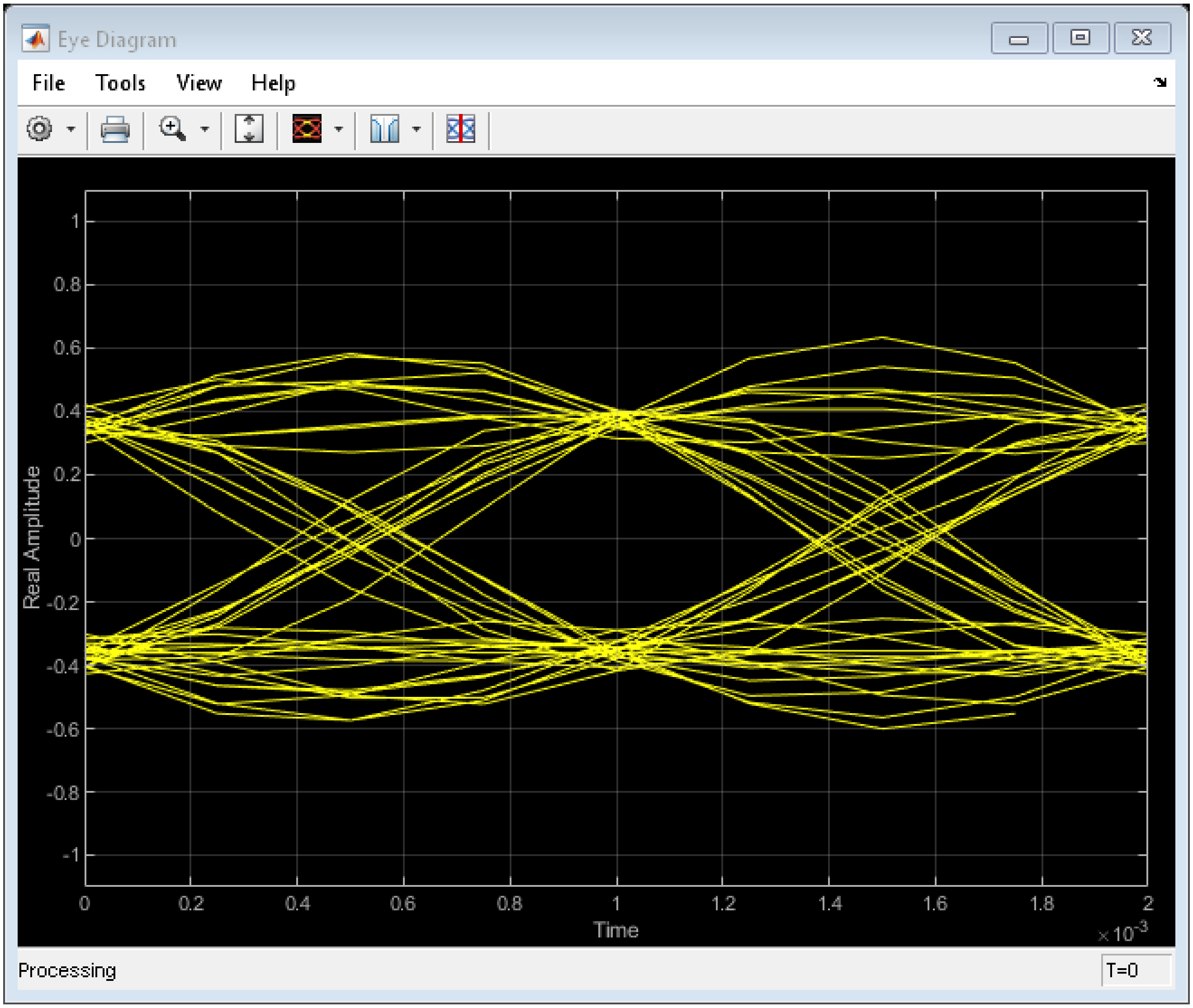
```
fs = 1000;
sps = 4;
```

Create transmit filter and eye diagram objects.

```
txfilter = comm.RaisedCosineTransmitFilter(...
    'OutputSamplesPerSymbol',sps);
ed = comm.EyeDiagram('SampleRate',fs*sps,'SamplesPerSymbol',sps);
```

Generate random symbols and apply QPSK modulation. Then filter the modulated signal and display the eye diagram.

```
data = randi([0 3],1000,1);
modSig = pskmod(data,4,pi/4);

txSig = txfilter(modSig);
ed(txSig)
```



More About

Measurements

Measurements assume that the eye diagram object has valid data. A valid eye diagram has two distinct eye crossing points and two distinct eye levels.

To open the measurements pane, click on the **Eye Measurements** button or select **Tools > Measurements > Eye Measurements** from the toolbar menu.

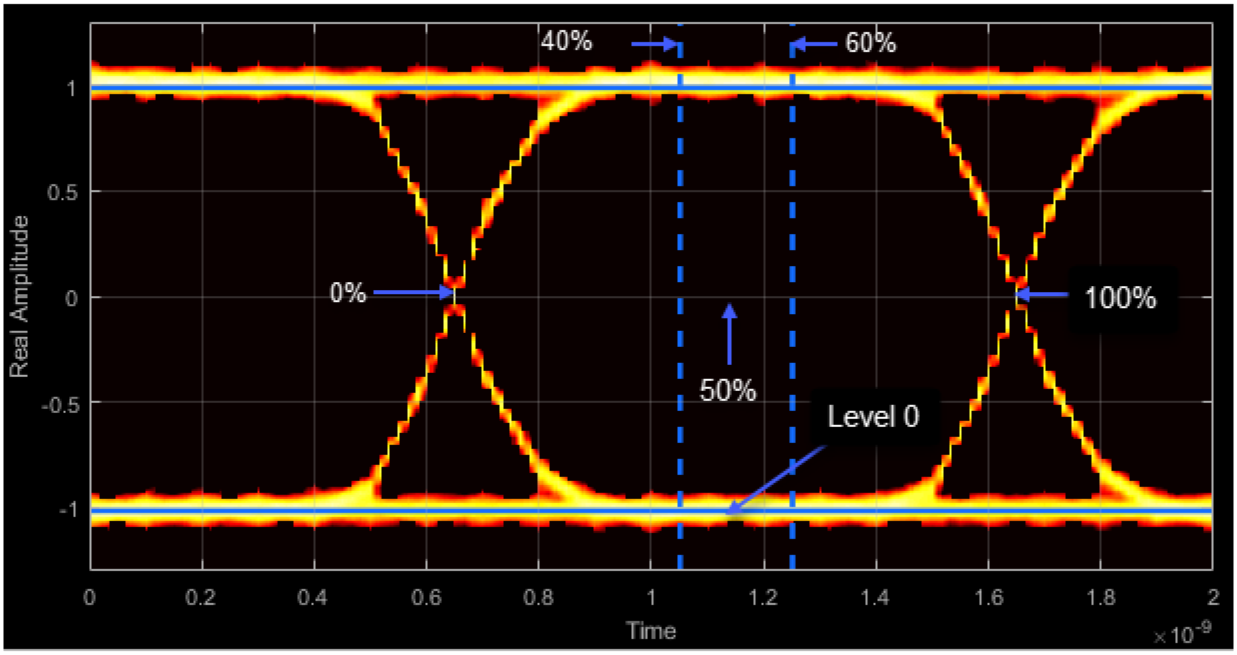
Note

- For amplitude measurements, at least one bin per vertical histogram must reach 10 hits before the measurement is taken, ensuring higher accuracy.
- For time measurements, at least one bin per horizontal histogram must reach 10 hits before the measurement is taken.

- When an eye crossing time measurement falls within the [-0.5/Fs, 0) seconds interval, the time measurement wraps to the end of the eye diagram, i.e., the measurement wraps by $2\times T_s$ seconds (where T_s is the symbol time). For a complex signal case, the analyze method issues a warning if the crossing time measurement of the in-phase branch wraps while that of the quadrature branch does not (or vice versa). To avoid time-wrapping or a warning, add a half-symbol duration delay to the current value in the `MeasurementDelay` property of the eye diagram object. This additional delay repositions the eye in the approximate center of the scope.

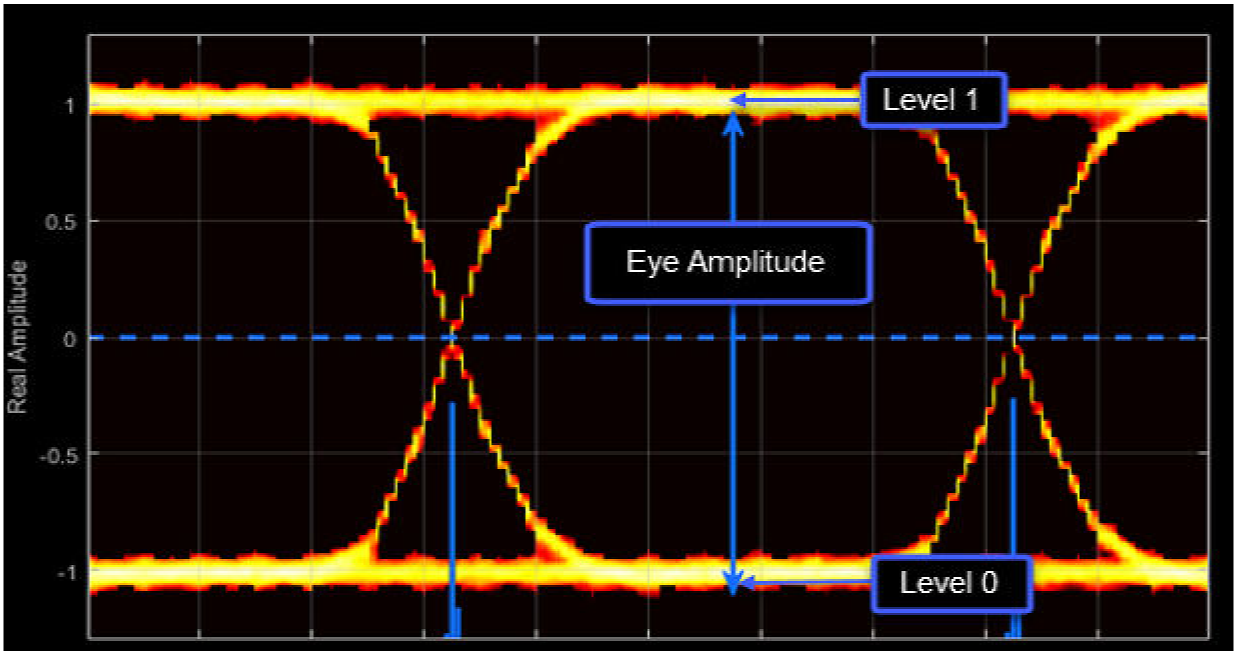
Eye Levels - Amplitude level used to represent data bits

Eye level is the amplitude level used to represent data bits. For the displayed NRZ signal, the levels are –1 V and +1 V. The eye levels are calculated by averaging the 2-D histogram within the eye level boundaries. For example, when the EyeLevelBoundaries property is set to `[40 60]`, that is, 40% and 60% of the symbol duration, the eye levels are calculated by estimating the mean value of the vertical histogram in this window marked by the eye level boundaries.



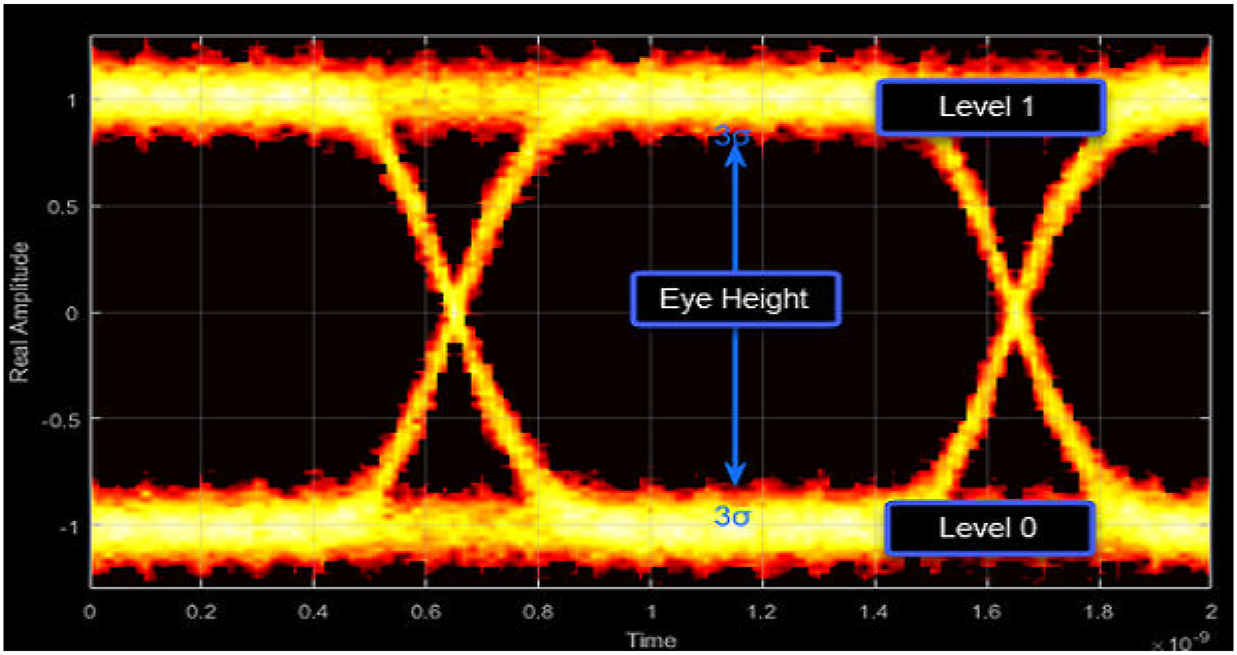
Eye Amplitude - Distance between eye levels

Eye amplitude is the distance in V between the mean value of two eye levels.



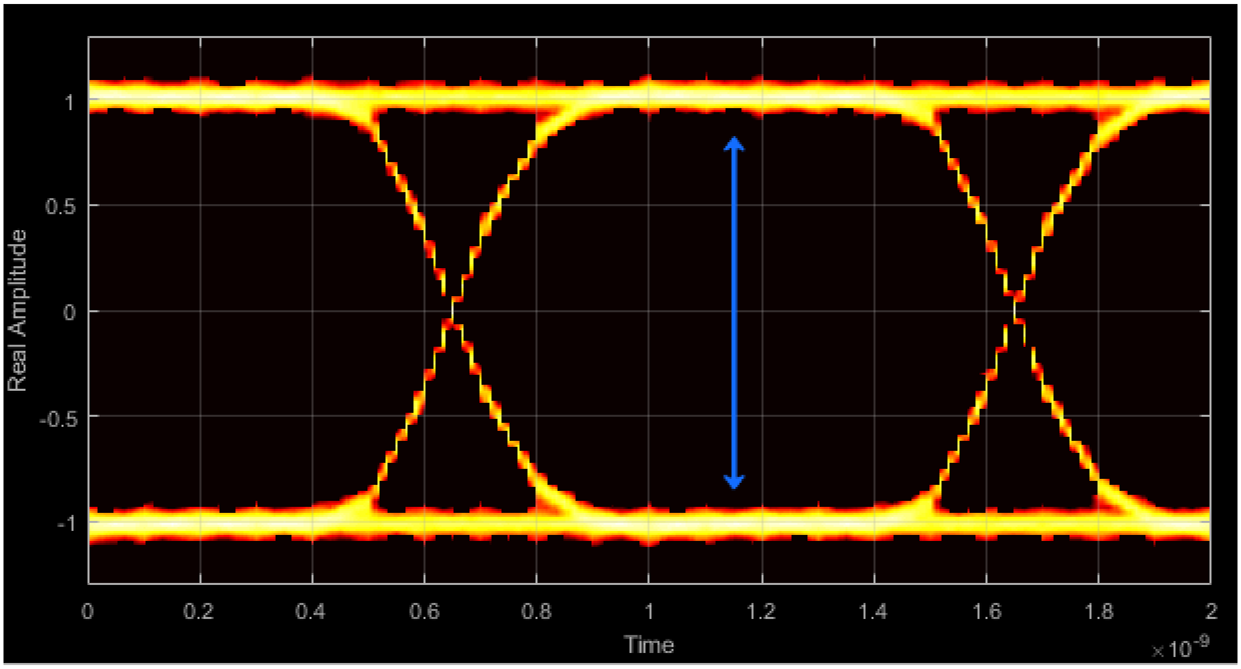
Eye Height - Statistical minimum distance between eye levels

Eye height is the distance between μ – 3σ of the upper eye level and μ + 3σ of the lower eye level. μ is the mean of the eye level, and σ is the standard deviation.



Vertical Opening - Distance between BER threshold points

The vertical opening is the distance between the two points that correspond to the BERThreshold property. For example, for a BER threshold of 10^{-12}, these points correspond to the 7σ distance from each eye level.



Eye SNR - Signal-to-noise ratio

The eye SNR is the ratio of the eye level difference to the difference of the vertical standard deviations corresponding to each eye level:

$$\mathrm{SNR} = \frac{L_1 - L_0}{\sigma_1 - \sigma_0},$$

where L_1 and L_0 represent the means of the upper and lower eye levels and σ_1 and σ_0 represent their standard deviations.

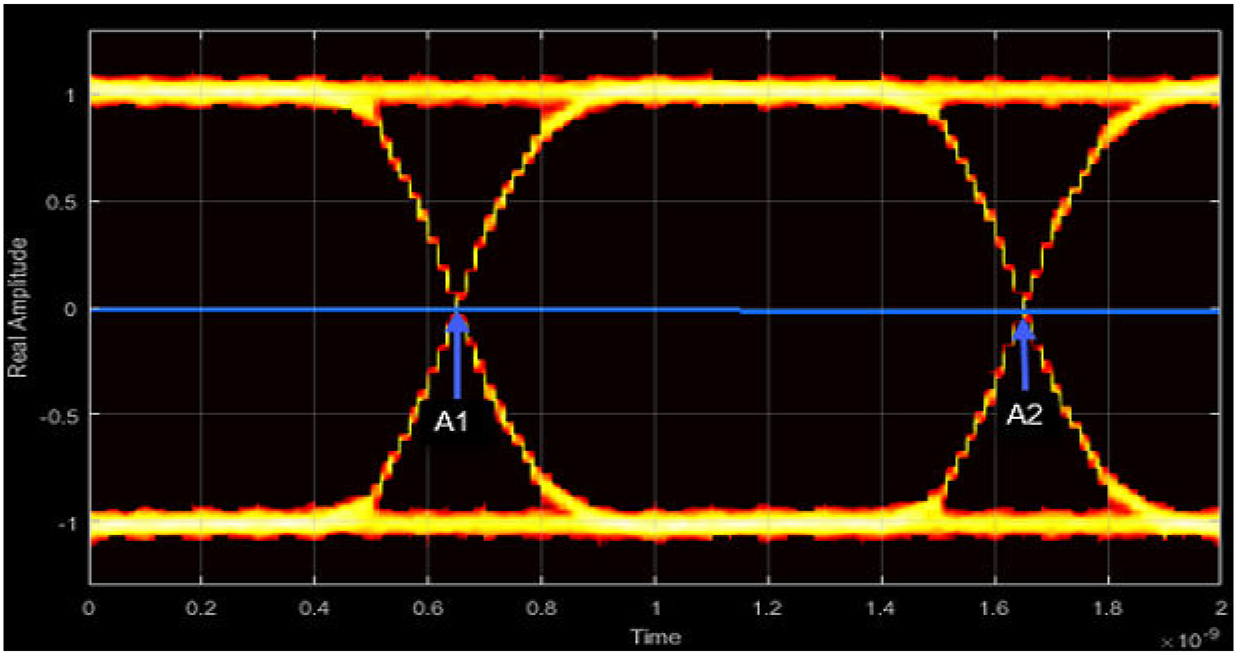
Q Factor - Quality factor

The Q factor is the quality factor and is calculated using the same formula as the eye SNR. However, the standard deviations of the vertical histograms are replaced with those computed with the dual-Dirac analysis.

Crossing Levels - Amplitude levels for eye crossings

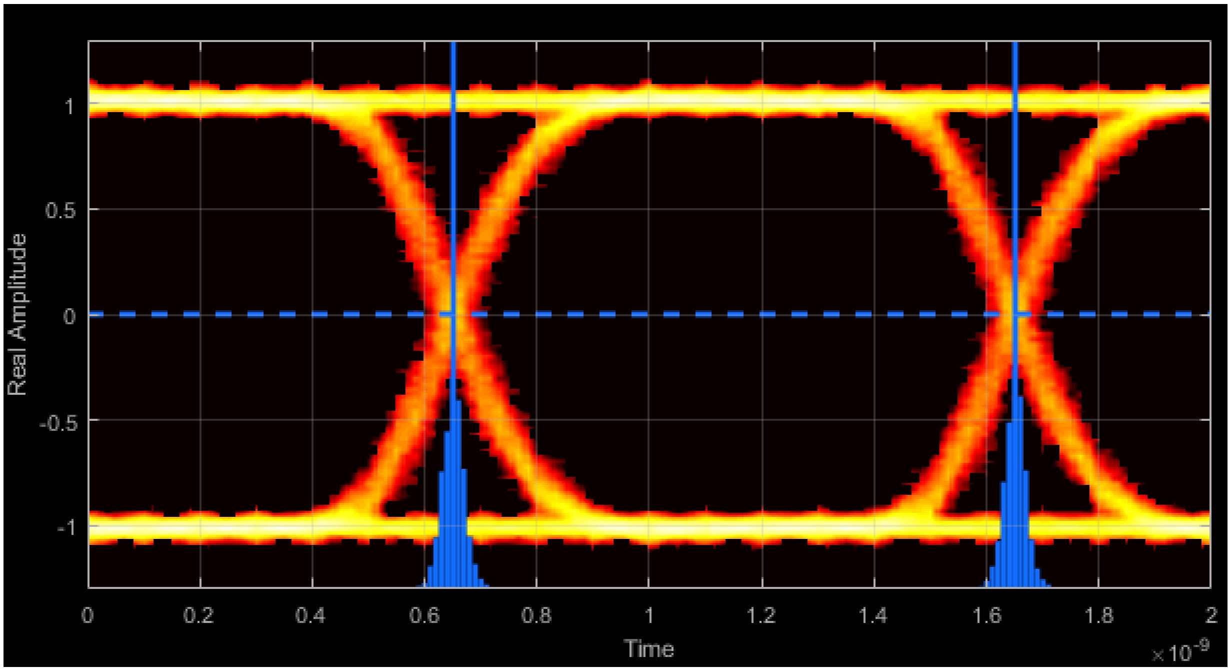
The crossing levels are the amplitude levels at which the eye crossings occur.

The level at which the input signal crosses the amplitude value is specified by the DecisionBoundary property.



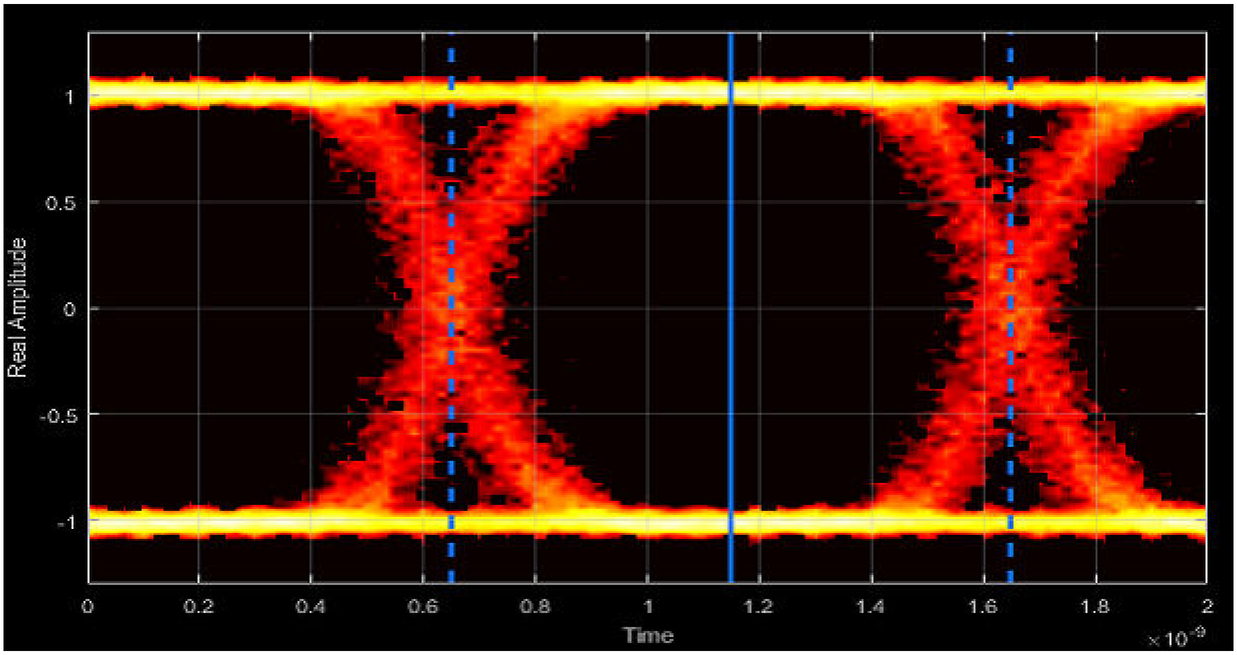
Crossing Times - Times for which crossings occur

The crossing times are the times at which the crossings occur. The times are computed as the mean values of the horizontal (jitter) histograms.



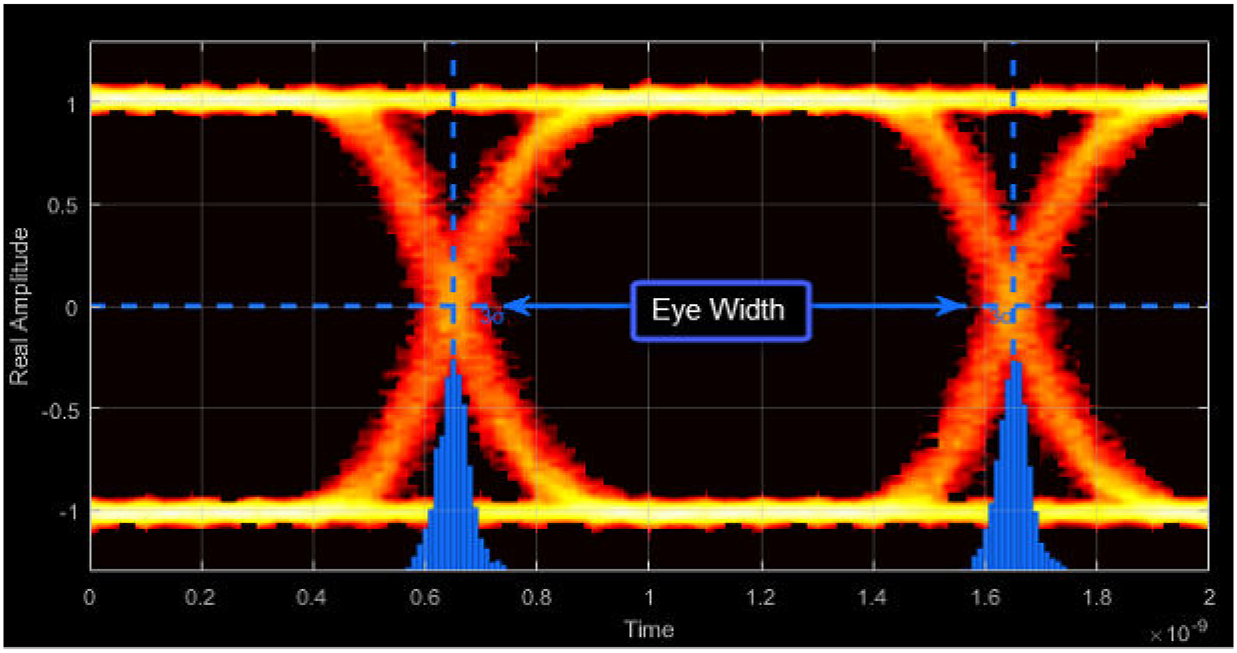
Eye Delay - Mean time between eye crossings

Eye delay is the midpoint between the two crossing times.



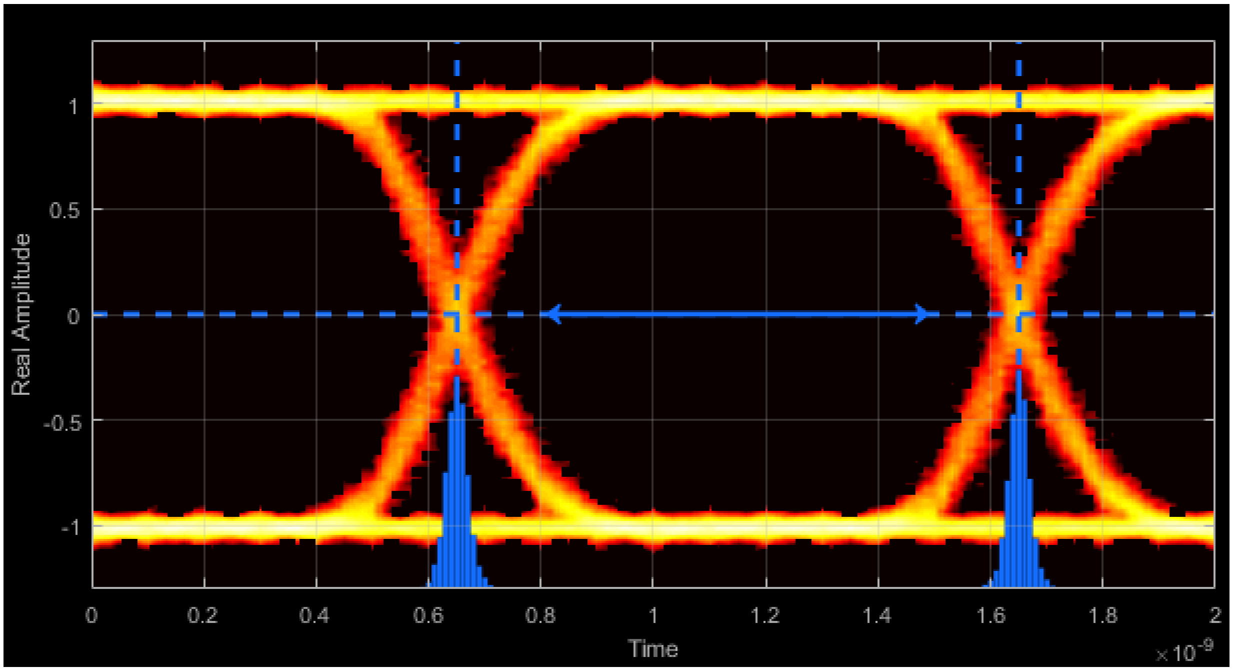
Eye Width - Statistical minimum time between eye crossings

Eye width is the horizontal distance between μ + 3σ of the left crossing time and μ – 3σ of the right crossing time. μ is the mean of the jitter histogram, and σ is the standard deviation.



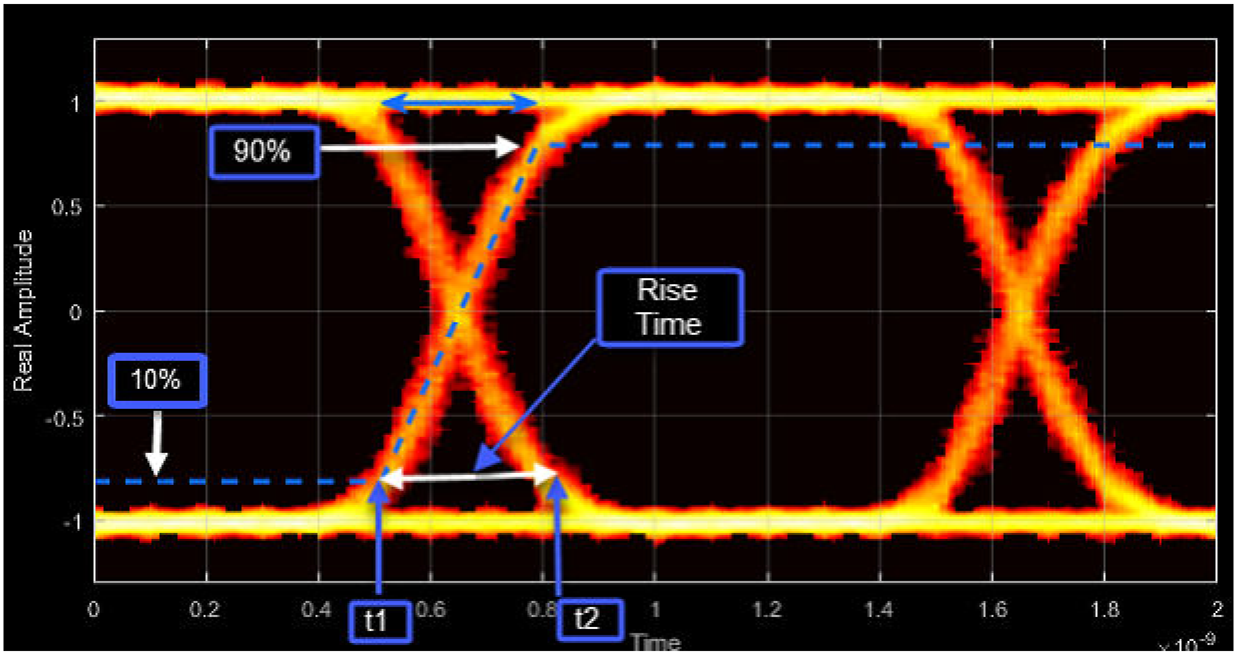
Horizontal Opening - Time between BER threshold points

The horizontal opening is the distance between the two points that correspond to the BERThreshold property. For example, for a 10^{-12} BER, these two points correspond to the 7σ distance from each crossing time.



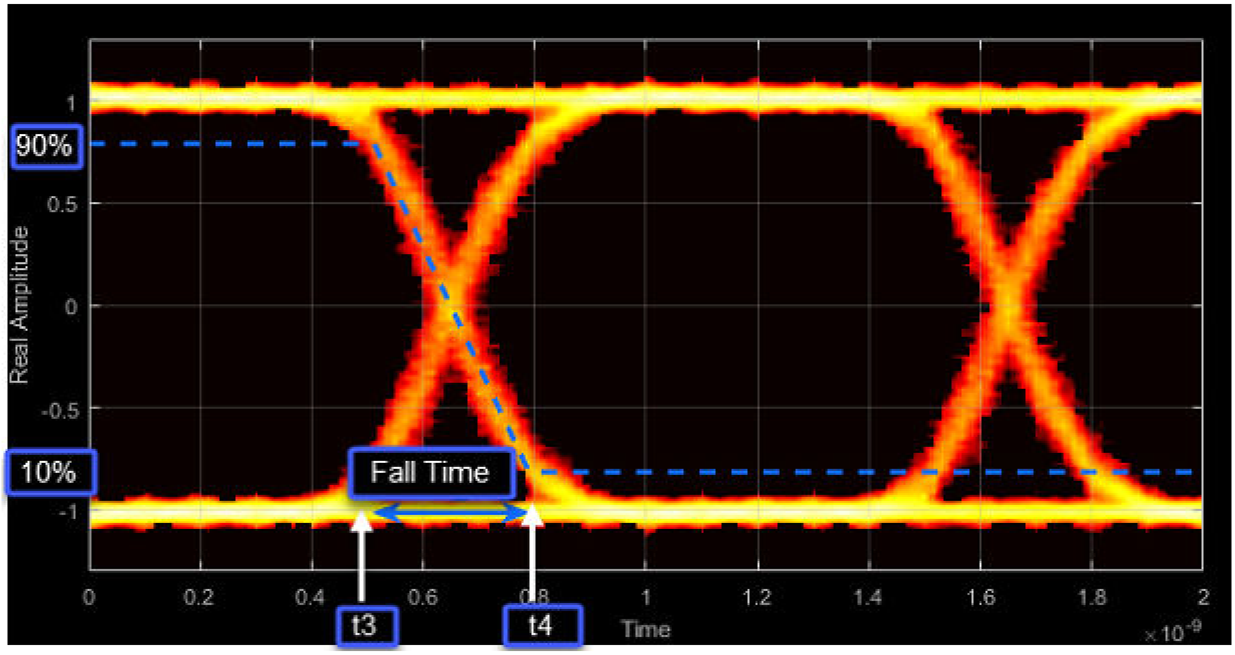
Rise Time - Time to transition from low to high

Rise time is the mean time between the low and high rise/fall thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



Fall Time - Time to transition from high to low

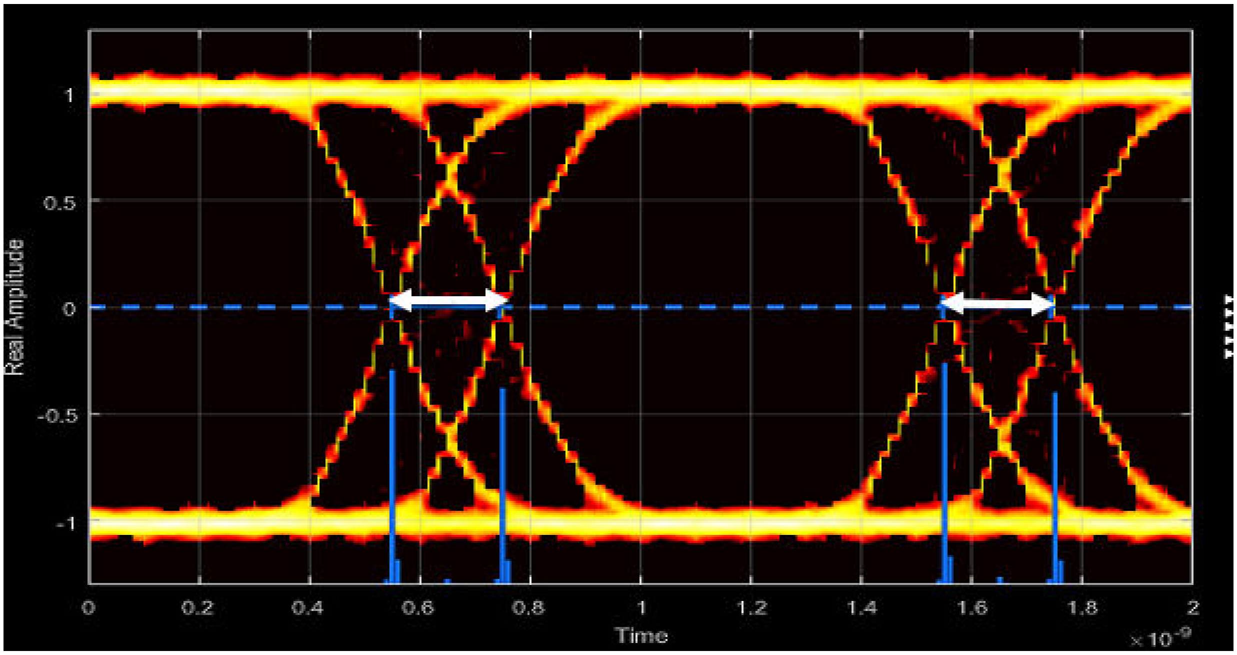
Fall time is the mean time between the high and low rise/fall thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



Deterministic Jitter - Deterministic deviation from ideal signal timing

Jitter is the deviation of a signal's timing event from its intended (ideal) occurrence in time [2]. Jitter can be represented with a dual-Dirac model. A dual-Dirac model assumes that the jitter has two components: deterministic jitter (DJ) and random jitter (RJ).

DJ is the distance between the two peaks of the dual-Dirac histograms. The probability density function (PDF) of DJ is composed of two delta functions.



Random Jitter - Random deviation from ideal signal timing

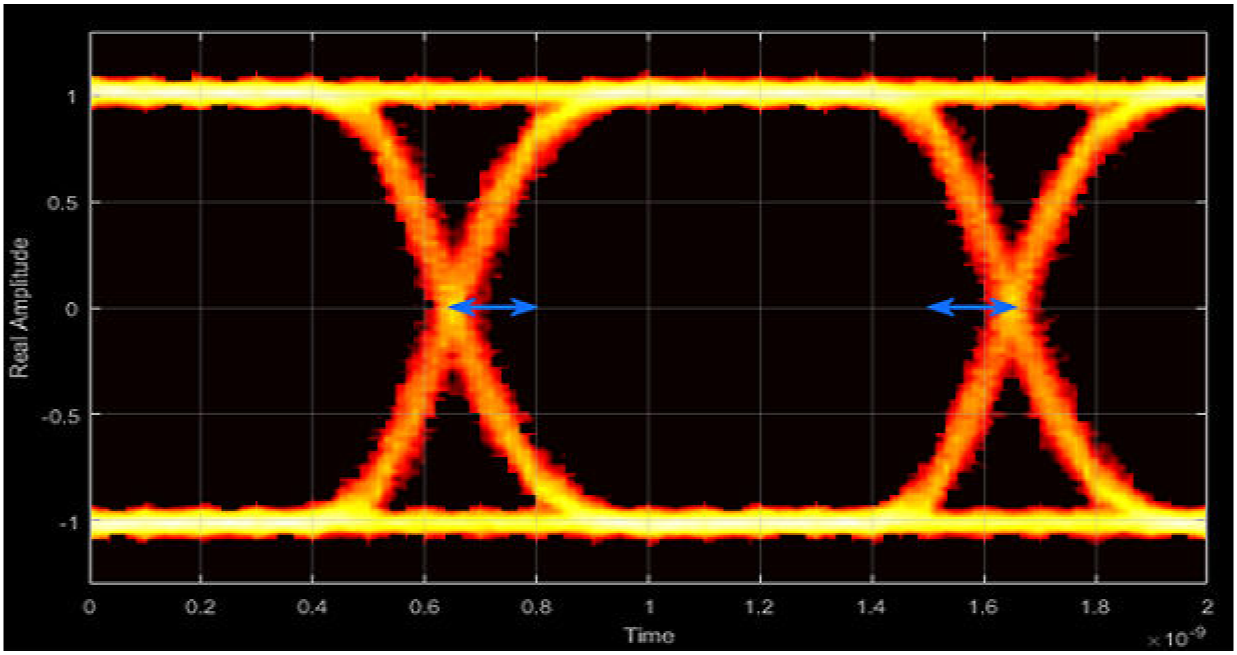
RJ is the Gaussian unbounded jitter component. The random component of jitter is modeled as a zero-mean Gaussian random variable with a specified standard-deviation of σ. The RJ is computed as:

$$\mathrm{RJ} = (Q_L + Q_R)\sigma\,,$$

where

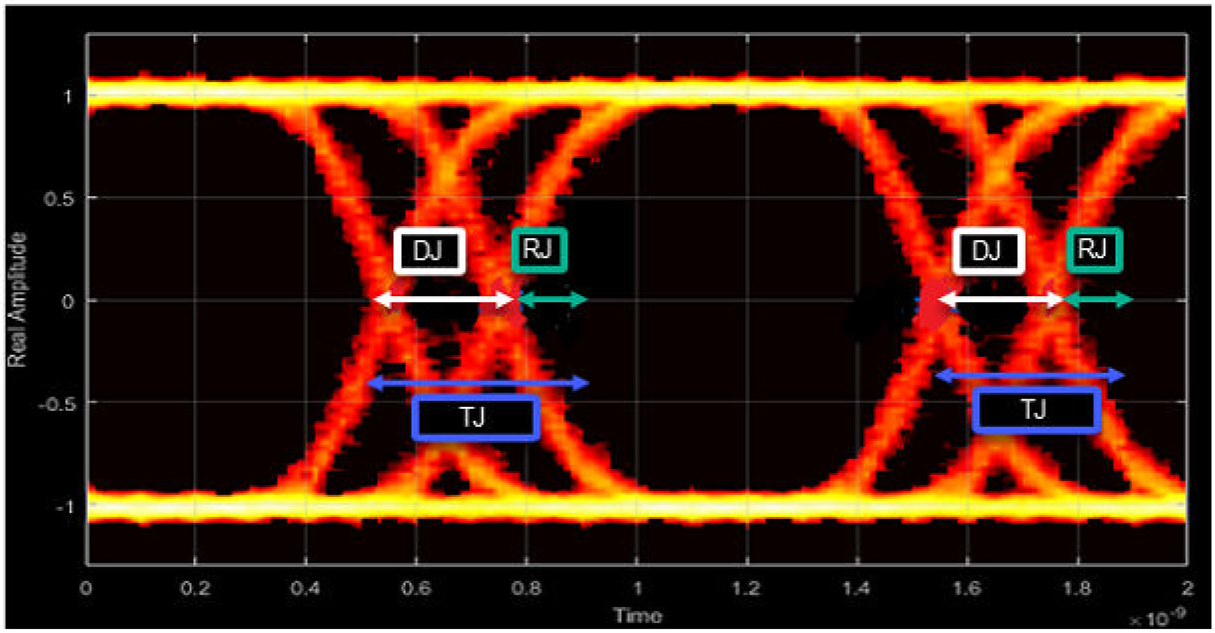
$$Q = \sqrt{2}\mathrm{erfc}^{-1}\left(2\frac{BER}{\rho}\right)\,.$$

BER is the specified BER threshold. ρ is the amplitude of the left and right Dirac function, which is determined from the bin counts of the jitter histograms.

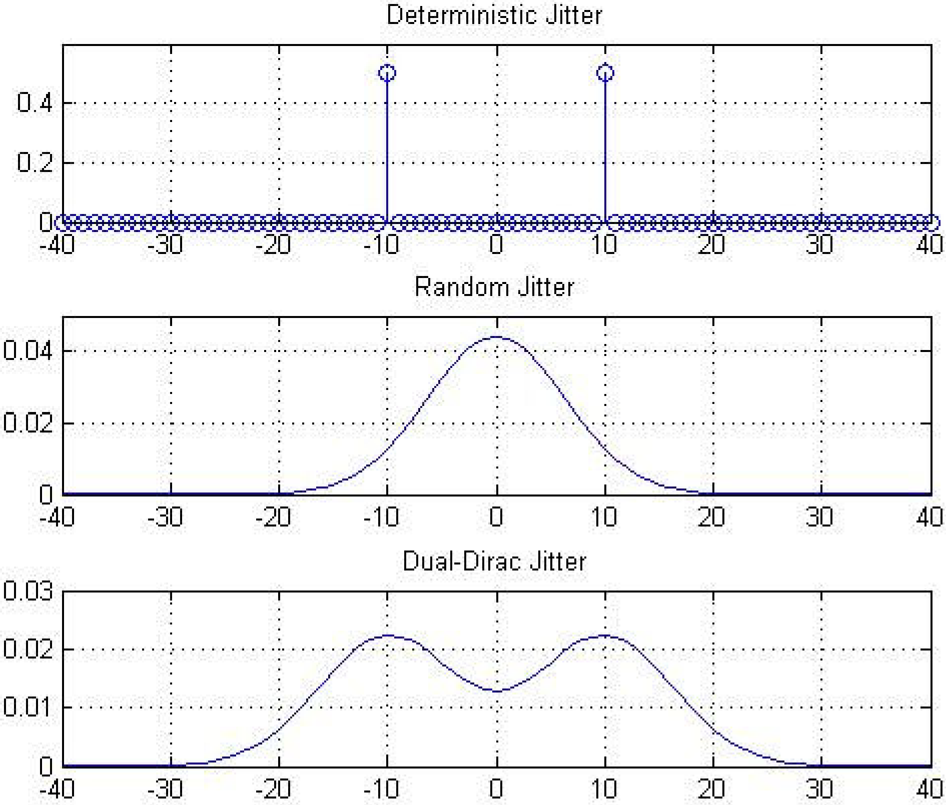


Total Jitter - Deviation from ideal signal timing

Total jitter (TJ) is the sum of the deterministic and random jitter, such that TJ = DJ + RJ.

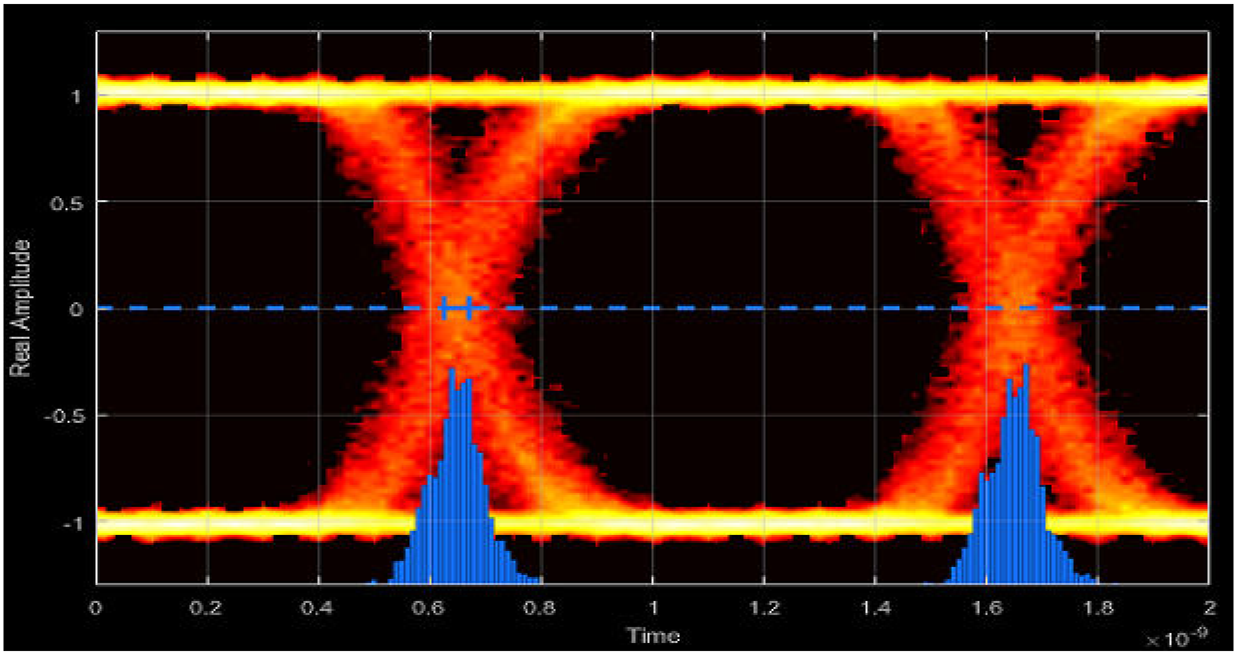


The total jitter PDF is the convolution of the DJ PDF and the RJ PDF.



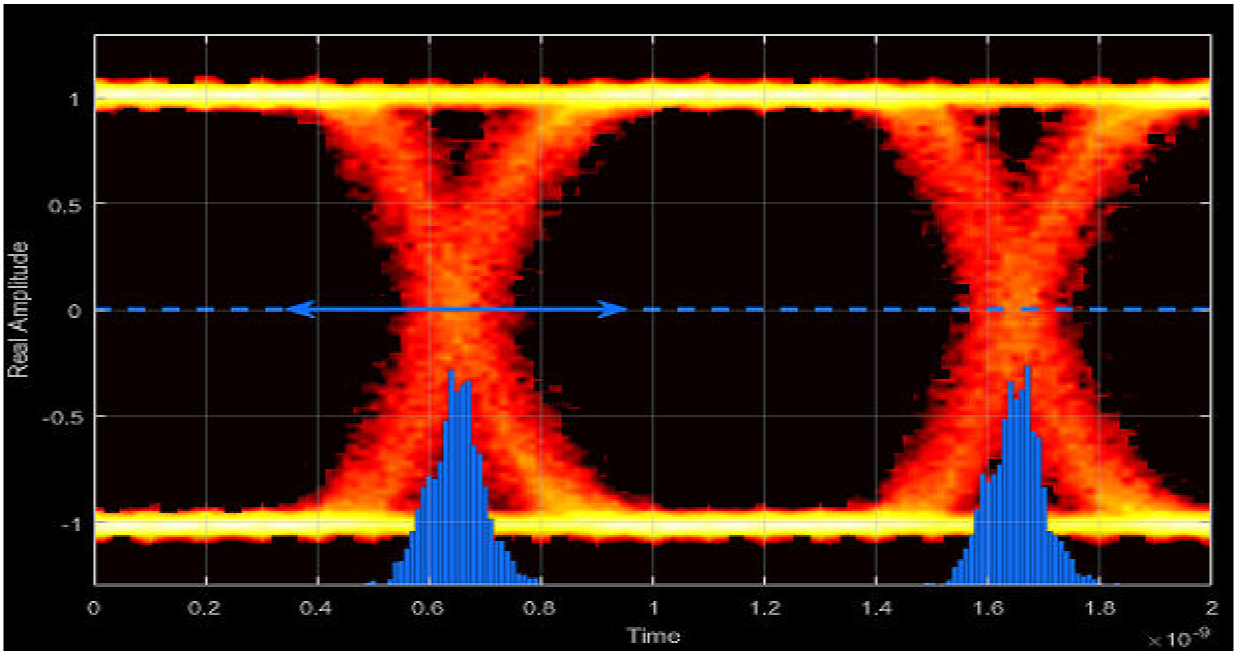
RMS Jitter - Standard deviation of jitter

RMS jitter is the standard deviation of the jitter calculated in the horizontal (jitter) histogram at the decision boundary.



Peak-to-Peak Jitter - Distance between extreme data points of histogram

Peak-to-peak jitter is the maximum horizontal distance between the left and right nonzero values in the horizontal histogram of each crossing time.



Compatibility Considerations

`comm.EyeDiagram` will be removed in a future release.
Not recommended starting in R2020b

comm.EyeDiagram will be removed in a future release. To display the eye diagram of a signal, use the `eyediagram` function instead.

References

[1] Stephens, Ransom. "Jitter analysis: The dual-Dirac model, RJ/DJ, and Q-scale." *Agilent Technical Note* (2004).

[2] Ou, N., T. Farahmand, A. Kuo, S. Tabatabaei, and A. Ivanov. “Jitter Models for the Design and Test of Gbps-Speed Serial Interconnects.” *IEEE Design and Test of Computers* 21, no. 4 (July 2004): 302–13. https://doi.org/10.1109/MDT.2004.34.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supports MEX code generation by treating the calls to the object as extrinsic. Does not support code generation for standalone applications.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.ConstellationDiagram`

Blocks
Eye Diagram Scope

Functions
`eyediagram`

Introduced in R2016b

hide

System object: `comm.EyeDiagram`
Package: `comm`

Hide scope window

Syntax

```
hide(ed)
```

Description

`hide(ed)` hides the eye diagram window associated with System object `ed`.

See Also

`show`

Introduced in R2016b

horizontalBathtub

Package: comm

Horizontal bathtub curve

Syntax

```
s = horizontalBathtub(ed)
```

Description

`s = horizontalBathtub(ed)` returns a structure containing information of horizontalBathtub curve for the System object.

Note This method is available when both of these conditions apply:

- `EnableMeasurements` is `true`
- `ShowBathtub` is `'Horizontal'` or `'Both'`

Examples

Horizontal and Vertical Bathtub Curve Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Generate and plot the horizontal and vertical bathtub curves.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps, ...
    'SampleOffset',sps/2,'DisplayMode','2D color histogram', ...
    'ColorScale','Logarithmic','EnableMeasurements',true, ...
    'ShowBathtub','Both','YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps, ...
    'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on', ...
    'DiracJitter','on','DiracDelta',[-5e-04 5e-04],'RandomStd',2e-4);
```

Generate two symbols for each trace.

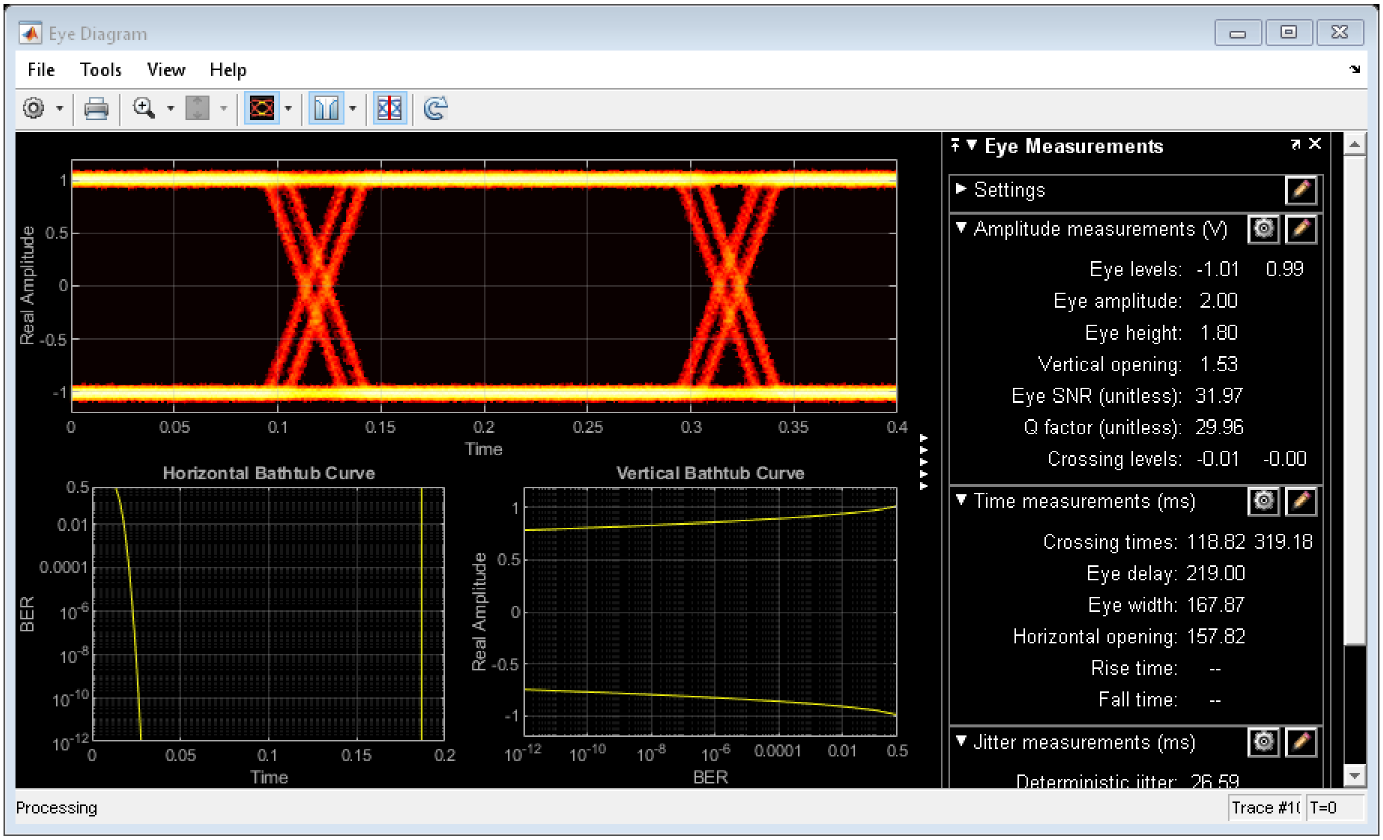
```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Display the eye diagram.

```
ed(y)
```



Eye Measurements

▶ Settings

▼ Amplitude measurements (V)

```
         Eye levels:  -1.01    0.99
      Eye amplitude:   2.00
         Eye height:   1.80
   Vertical opening:   1.53
  Eye SNR (unitless): 31.97
 Q factor (unitless): 29.96
    Crossing levels:  -0.01   -0.00
```

▼ Time measurements (ms)

```
     Crossing times: 118.82 319.18
          Eye delay: 219.00
          Eye width: 167.87
 Horizontal opening: 157.82
          Rise time:   --
          Fall time:   --
```

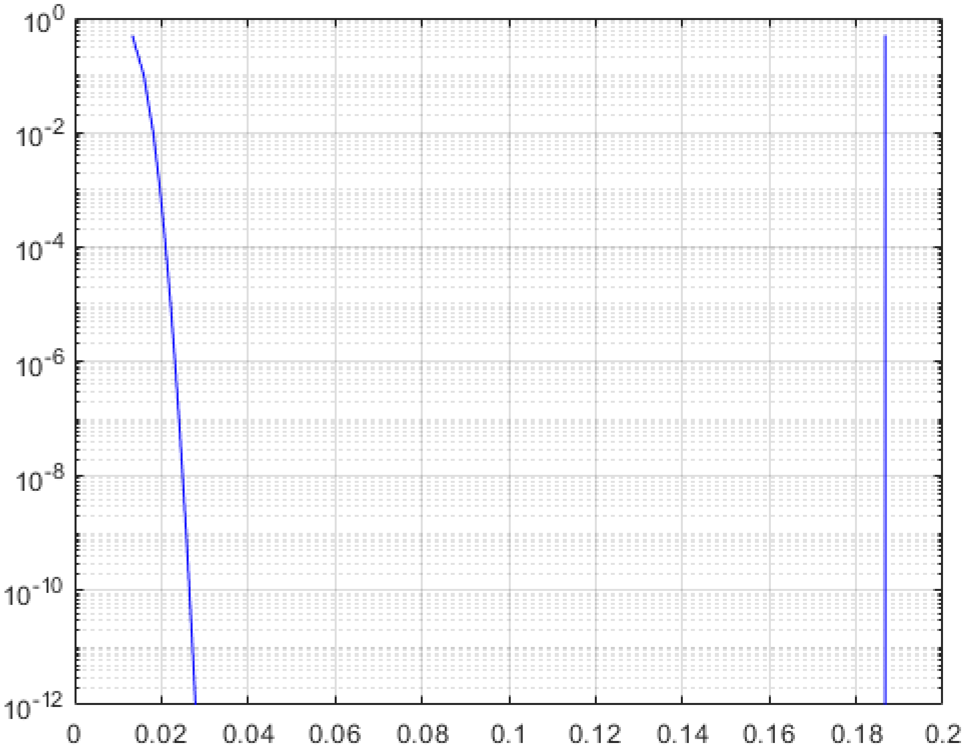
▼ Jitter measurements (ms)

```
 Deterministic jitter: 26.59
```

Generate the horizontal bathtub data for the eye diagram. Plot the curve.

```
hb = horizontalBathtub(ed)
semilogy([hb.LeftThreshold],[hb.BER],'b', ...
    [hb.RightThreshold],[hb.BER],'b')
grid
```

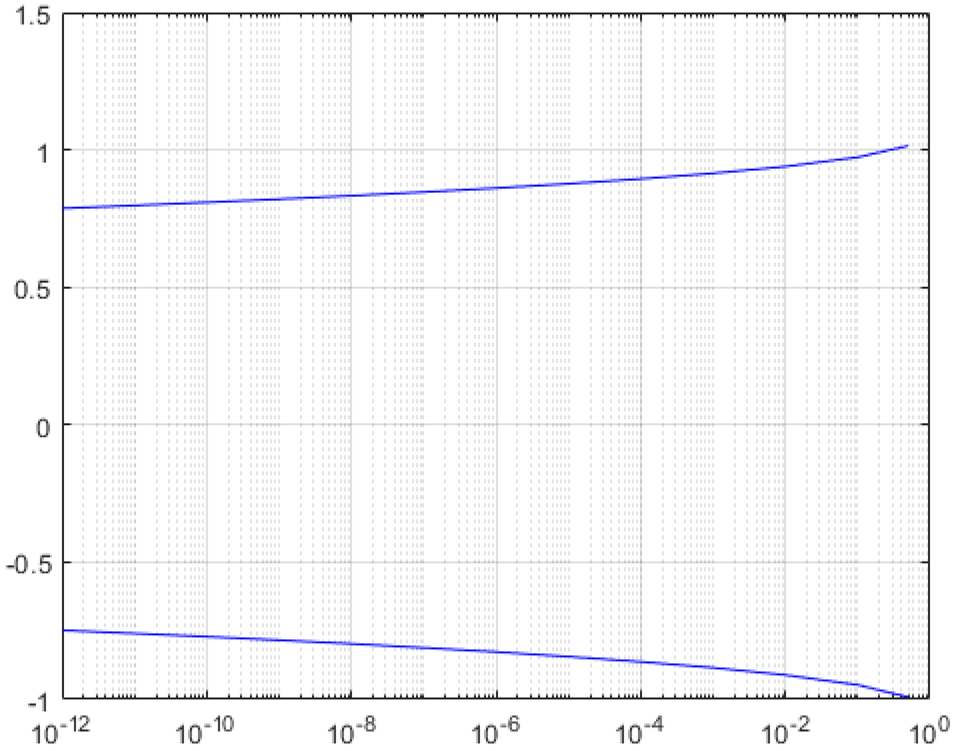
```
hb = 

  1x13 struct array with fields:

    BER
    LeftThreshold
    RightThreshold
```



Generate the vertical bathtub data for the eye diagram. Plot the curve.

```
vb = verticalBathtub(ed)
semilogx([vb.BER],[vb.LowerThreshold],'b', ...
    [vb.BER],[vb.UpperThreshold],'b')
grid
```

```
vb = 

  1x13 struct array with fields:

    BER
    UpperThreshold
    LowerThreshold
```



Input Arguments

ed — Eye Diagram System object
System object

Eye Diagram System object, where you get the bathtub curve information from.

Output Arguments

s — Structure containing information
struct

Structure containing information about the horizontal bathtub curve.

BER — Bit error rate values
scalar

Bit error rate values, mapped on the Y-axis of the horizontalBathtub plot against the corresponding `LeftThreshold` and `RightThreshold` values on the x-axis, specified as a scalar.

Data Types: `double`

`LeftThreshold` — Left threshold values
scalar

Left threshold values, mapped on the x-axis in the plot against corresponding BER values on the x-axis.

Data Types: `double`

`RightThreshold` — Right threshold values
scalar

Right threshold values, mapped on the x-axis in the plot against corresponding BER values on the x-axis.

Data Types: `double`

See Also

`verticalBathtub`

Introduced in R2016b

jitterHistogram

Package: comm

Jitter histogram

Syntax

```
jh = jitterHistogram(ed)
```

Description

`jh = jitterHistogram(ed)` returns the bin counts for decision boundary crossings set in eye diagram System object.

Note This method is available when `EnableMeasurements` is `true`.

Examples

Jitter and Noise Histogram Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Plot the jitter and noise histograms.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps, ...
    'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

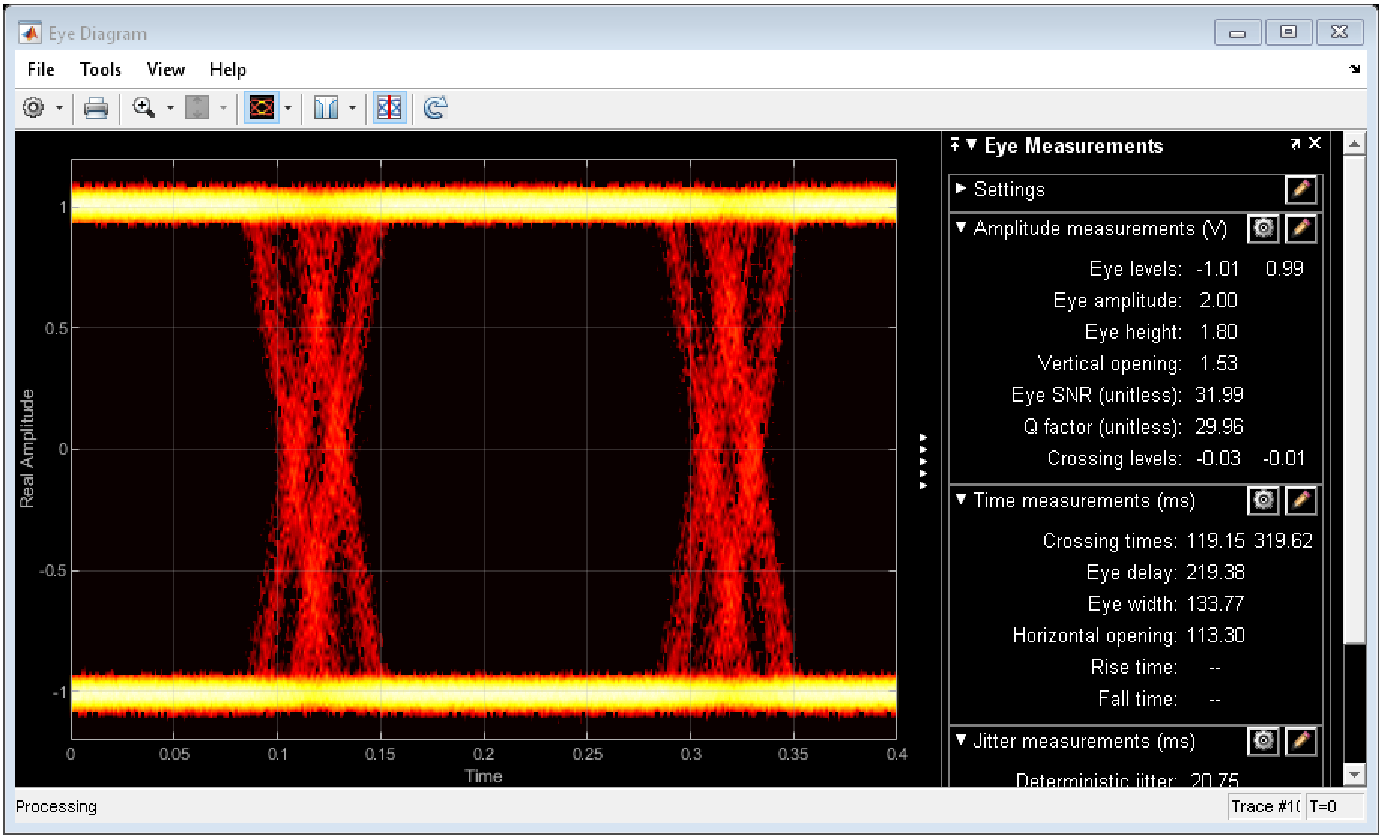
```
src = commsrc.pattern('SamplesPerSymbol',sps, ...
    'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on', ...
    'DiracJitter','on','DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

Generate two symbols for each trace.

```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
ed(y)
```



Amplitude measurements (V)

Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.99
Q factor (unitless): 29.96
Crossing levels: -0.03 -0.01

Time measurements (ms)

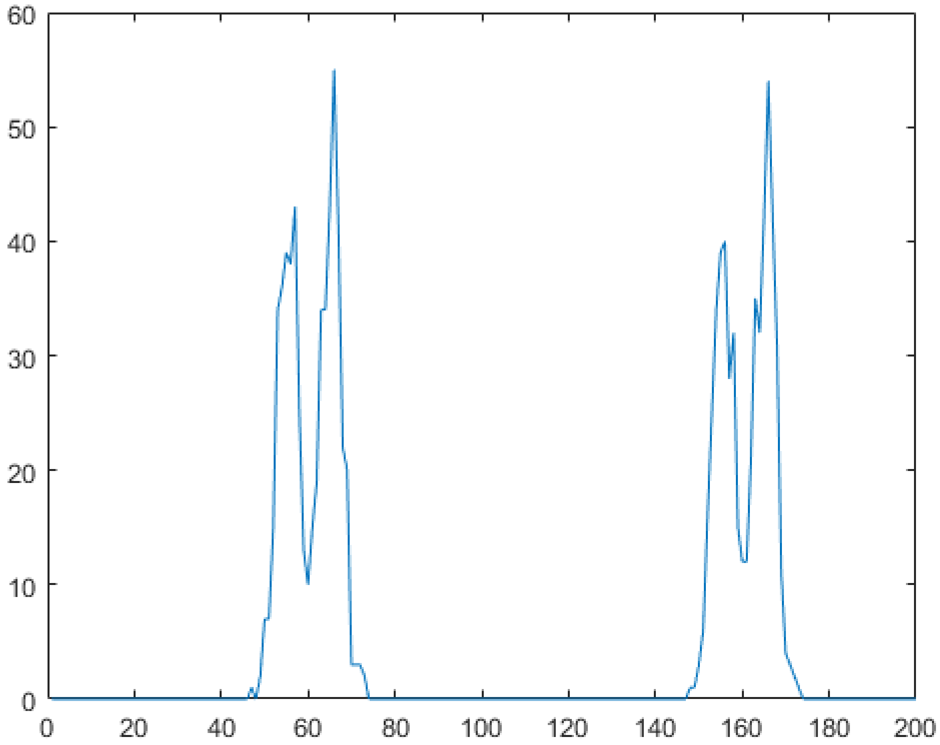
Crossing times: 119.15 319.62
Eye delay: 219.38
Eye width: 133.77
Horizontal opening: 113.30
Rise time: --
Fall time: --

Jitter measurements (ms)

Deterministic jitter: 20.75

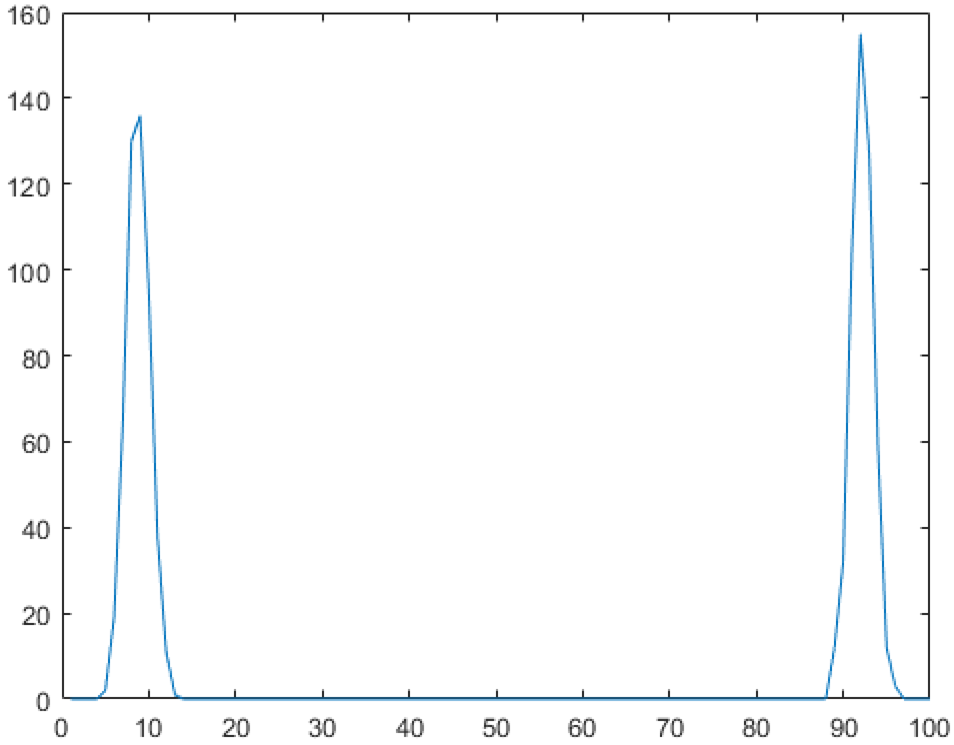
Calculate the jitter histogram count for each bin by using the `jitterHistogram` method. Plot the histogram.

```
jbins = jitterHistogram(ed);
plot(jbins)
```



Calculate the noise histogram count for each bin by using the `noiseHistogram` method. Plot the histogram.

```
nbins = noiseHistogram(ed);
plot(nbins)
```



Input Arguments

ed — Eye Diagram System object
System object

Eye Diagram System object, where the count for decision boundary crossings is set.

Output Arguments

`jh` — Jitter histogram
nonnegative integer

Jitter histogram, which represent the counts for decision boundary crossings, specified as a nonnegative integer.

Data Types: `double`

See Also

`noiseHistogram`

Introduced in R2016b

measurements

Package: comm

Measure eye diagram parameters

Syntax

```
m = measurements(ed)
```

Description

`m = measurements(ed)` returns the parameter measurements calculated by eye diagram System object.

Note This method is available when `EnableMeasurements` is `true`.

Examples

Rise and Fall Time of NRZ Signal

Create a combined jitter object having random jitter with a 2e-4 standard deviation.

```
jtr = commsrc.combinedjitter('RandomJitter','on','RandomStd',2e-4);
```

Generate an NRZ signal having random jitter and 3 ms rise and fall times.

```
genNRZ = commsrc.pattern('Jitter',jtr,'RiseTime',3e-3,'FallTime',3e-3);
x = generate(genNRZ,2000);
```

Pass the signal through an AWGN channel with fixed seed for repeatable results.

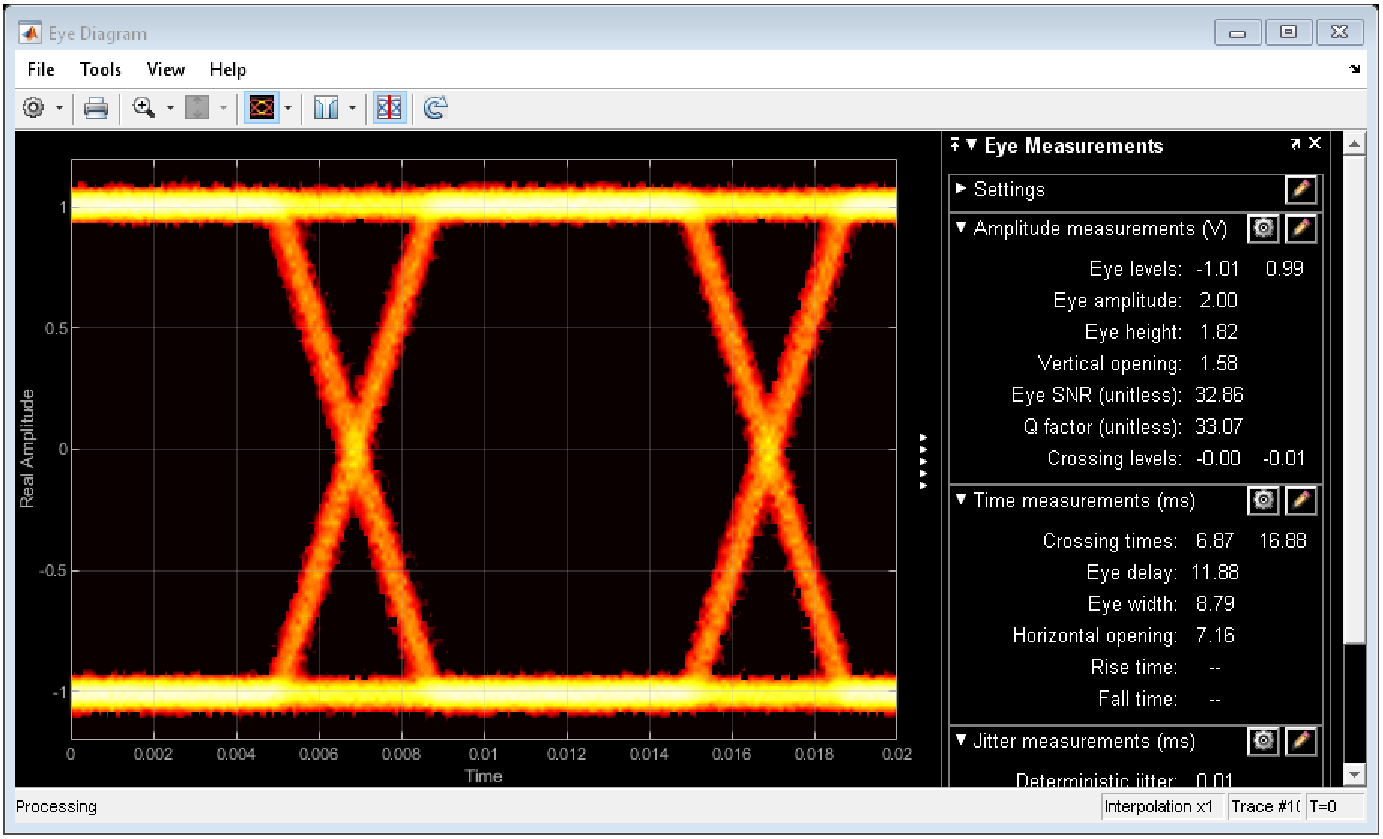
```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Create an eye diagram object. Enable the measurements.

```
ed = comm.EyeDiagram('SamplesPerSymbol',genNRZ.SamplesPerSymbol, ...
    'SampleRate',genNRZ.SamplingFrequency, ...
    'SampleOffset',genNRZ.SamplesPerSymbol/2, ...
    'EnableMeasurements',true,'DisplayMode','2D color histogram', ...
    'OversamplingMethod','Input interpolation', ...
    'ColorScale','Logarithmic','YLimits',[-1.2 1.2]);
```

To compute the rise and fall times, determine the rise and fall thresholds from the eye level and eye amplitude measurements. Plot the eye diagram to calculate these parameters.

```
ed(y)
```



Eye Measurements

Settings

Amplitude measurements (V)

```
Eye levels:  -1.01    0.99
Eye amplitude:  2.00
Eye height:  1.82
Vertical opening:  1.58
Eye SNR (unitless):  32.86
Q factor (unitless):  33.07
Crossing levels:  -0.00    -0.01
```

Time measurements (ms)

```
Crossing times:  6.87    16.88
Eye delay:  11.88
Eye width:  8.79
Horizontal opening:  7.16
Rise time:  --
Fall time:  --
```

Jitter measurements (ms)

```
Deterministic jitter:  0.01
```

Processing | Interpolation x1 | Trace #1(| T=0

Pass the signal through the eye diagram object again to measure the rise and fall times.

```
ed(y)
hide(ed)
```

Display the data by using the measurements method.

```
eyestats = measurements(ed);
riseTime = eyestats.RiseTime
fallTime = eyestats.FallTime
```

```
riseTime =

    0.0030


fallTime =

    0.0030
```

The measured values match the 3 ms specification.

Input Arguments

ed — Eye Diagram System object

System object

Eye Diagram System object, where the parameter measurements are calculated.

Output Arguments

`m` — Eye Diagram parameters measurement

struct

Eye Diagram parameters measurement, returned as a structure containing all 18 parameters mentioned in “Measurements” on page 3-570, along with their values.

Data Types: `double`

See Also

Introduced in R2016b

noiseHistogram

Package: comm

Noise histogram

Syntax

```
nh = noiseHistogram(ed)
```

Description

nh = noiseHistogram(ed) returns the bin counts for the signal values at the vertical opening (eye delay) as set in eye diagram System object.

Note This method is available when both of these conditions apply:

- EnableMeasurements is true
- DisplayMode is '2D color histogram'

Examples

Jitter and Noise Histogram Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Plot the jitter and noise histograms.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps, ...
    'SampleOffset',sps/2, ...
    'DisplayMode','2D color histogram','ColorScale','Logarithmic', ...
    'EnableMeasurements',true,'YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

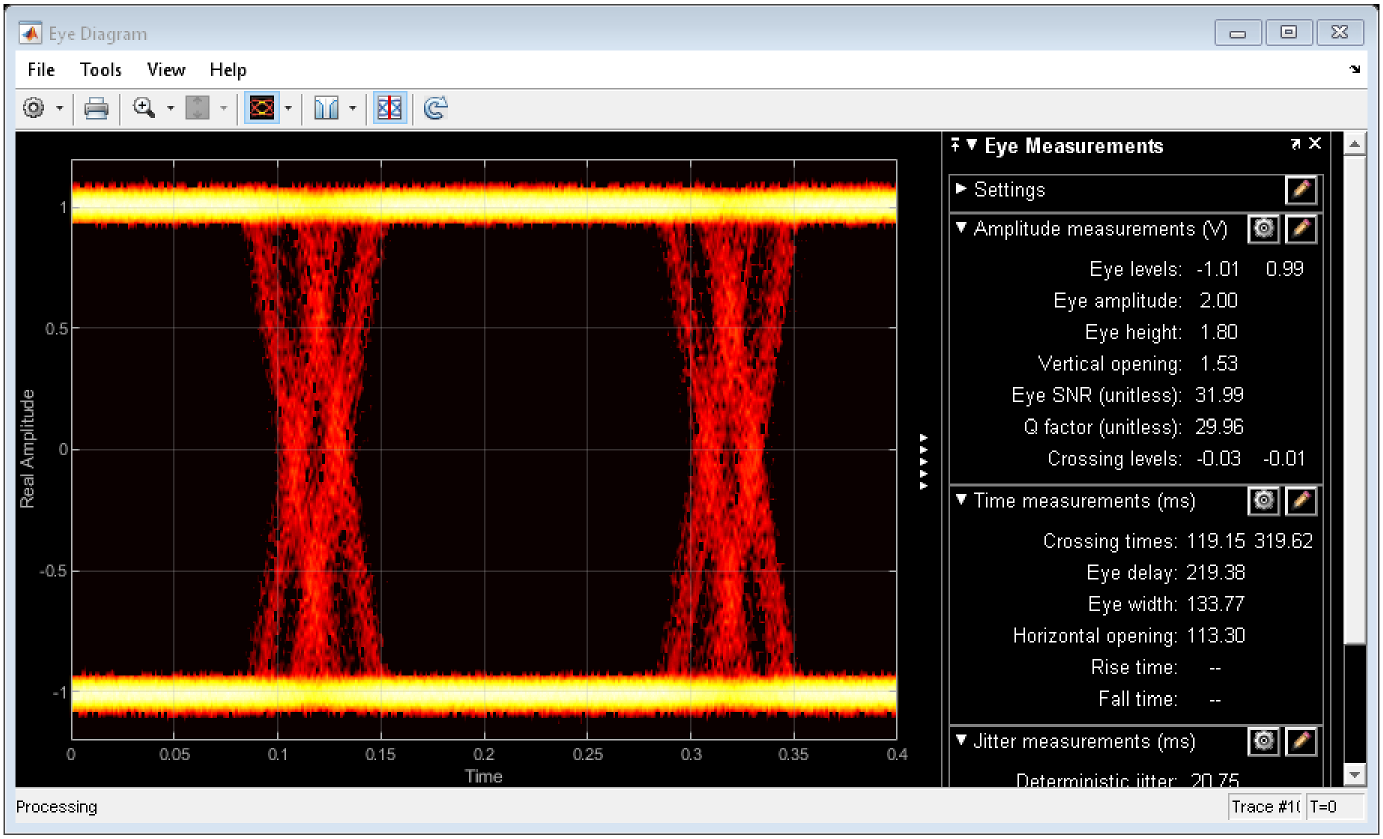
```
src = commsrc.pattern('SamplesPerSymbol',sps, ...
    'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on', ...
    'DiracJitter','on','DiracDelta',[-10e-04 10e-04],'RandomStd',5e-4);
```

Generate two symbols for each trace.

```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
ed(y)
```



Amplitude measurements (V)

```
Eye levels:  -1.01   0.99
Eye amplitude:  2.00
Eye height:  1.80
Vertical opening:  1.53
Eye SNR (unitless):  31.99
Q factor (unitless):  29.96
Crossing levels:  -0.03   -0.01
```

Time measurements (ms)

```
Crossing times: 119.15 319.62
Eye delay: 219.38
Eye width: 133.77
Horizontal opening: 113.30
Rise time:  --
Fall time:  --
```

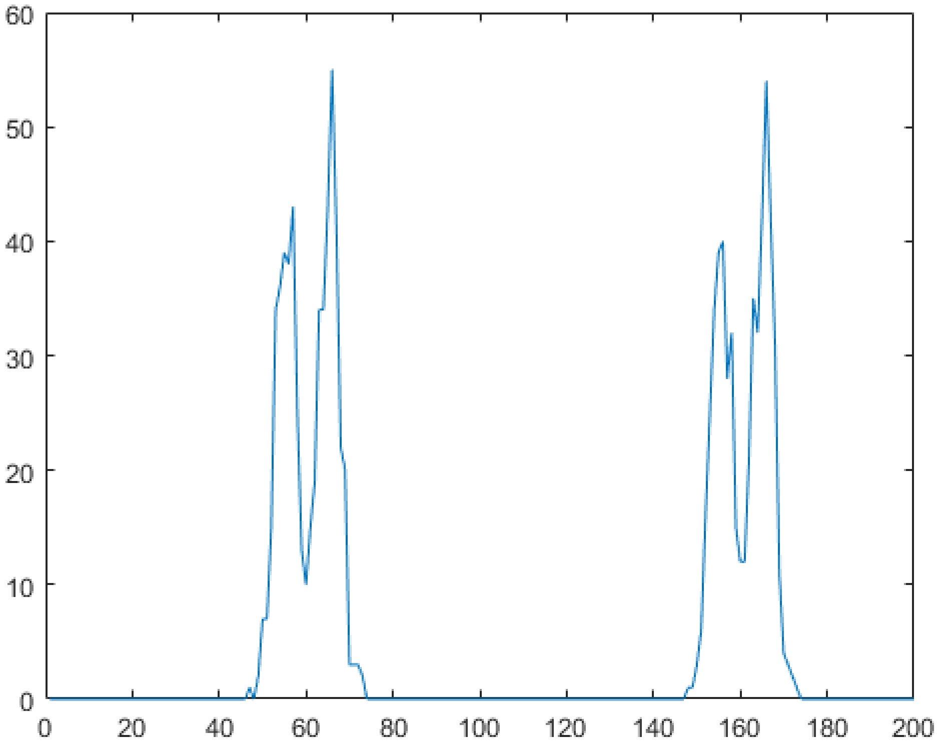
Jitter measurements (ms)

```
Deterministic jitter: 20.75
```

Processing   Trace #1(T=0

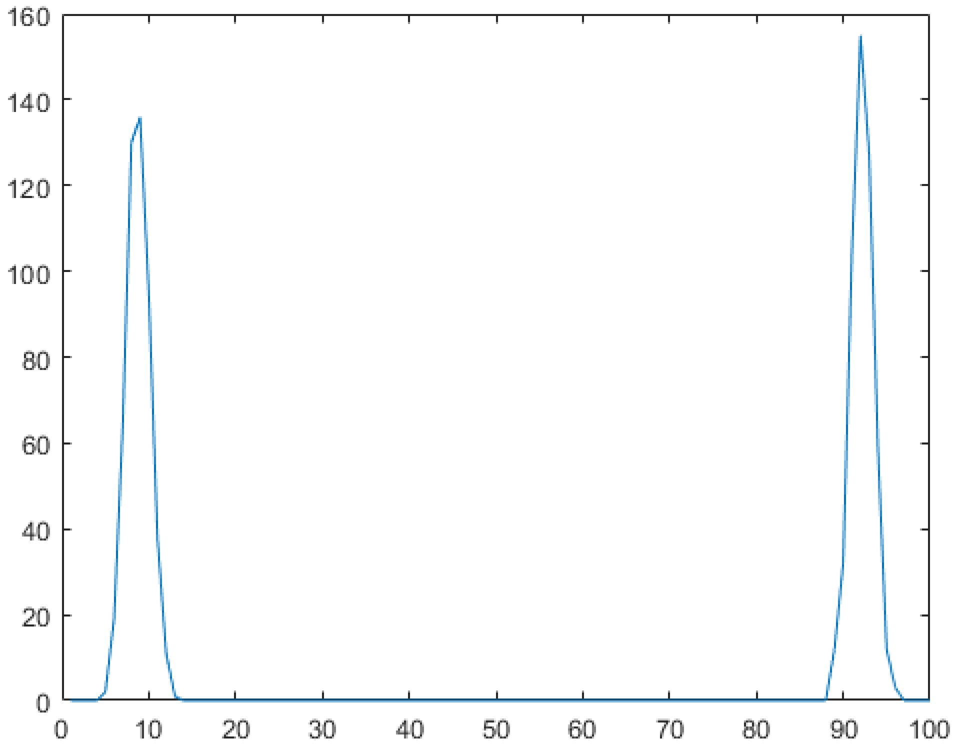
Calculate the jitter histogram count for each bin by using the `jitterHistogram` method. Plot the histogram.

```
jbins = jitterHistogram(ed);
plot(jbins)
```



Calculate the noise histogram count for each bin by using the `noiseHistogram` method. Plot the histogram.

```
nbins = noiseHistogram(ed);
plot(nbins)
```



Input Arguments

ed — Eye Diagram System object

System object

Eye Diagram System object, where the counts for signal values at the eye delay is set.

Output Arguments

nh — Noise histogram

nonnegative integer

Noise histogram, which represent the counts for the signal values at the vertical opening (eye delay), specified as a nonnegative integer.

Data Types: `double`

See Also

`jitterHistogram`

Introduced in R2016b

show

System object: `comm.EyeDiagram`
Package: `comm`

Make scope window visible

Syntax

```
show(ed)
```

Description

`show(ed)` makes the eye diagram window associated with System object `ed` visible.

See Also

`hide`

Introduced in R2016b

verticalBathtub

Package: comm

Vertical bathtub curve

Syntax

```
s = verticalBathtub(ed)
```

Description

`s = verticalBathtub(ed)` returns a structure containing information of verticalBathtub curve for the System object.

Note This method is available when both of these conditions apply:

- `EnableMeasurements` is `true`
- `ShowBathtub` is `'Vertical'` or `'Both'`

Examples

Horizontal and Vertical Bathtub Curve Methods

Display the eye diagram for a waveform having dual-dirac and random jitter. Generate and plot the horizontal and vertical bathtub curves.

Specify the sample rate, the samples per symbol, and the number of traces parameters.

```
fs = 1000;
sps = 200;
numTraces = 1000;
```

Create an eye diagram object.

```
ed = comm.EyeDiagram('SampleRate',fs,'SamplesPerSymbol',sps, ...
    'SampleOffset',sps/2,'DisplayMode','2D color histogram', ...
    'ColorScale','Logarithmic','EnableMeasurements',true, ...
    'ShowBathtub','Both','YLimits',[-1.2 1.2]);
```

Generate a waveform having dual-dirac and random jitter. Specify 3 ms rise and fall times.

```
src = commsrc.pattern('SamplesPerSymbol',sps, ...
    'RiseTime',3e-3,'FallTime', 3e-3);
src.Jitter = commsrc.combinedjitter('RandomJitter','on', ...
    'DiracJitter','on','DiracDelta',[-5e-04 5e-04],'RandomStd',2e-4);
```

Generate two symbols for each trace.

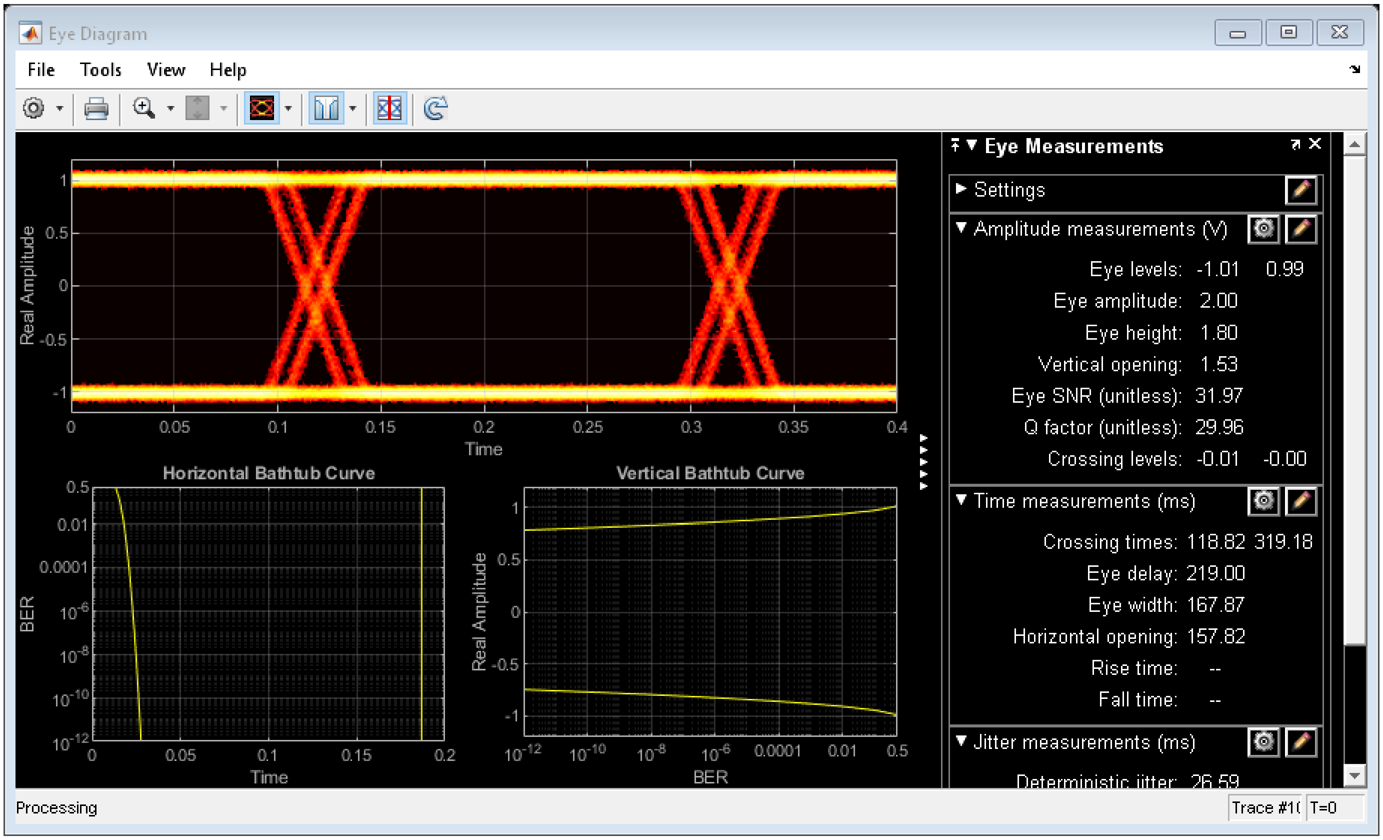
```
x = src.generate(numTraces*2);
```

Pass the signal through an AWGN channel with a fixed seed for repeatable results.

```
randStream = RandStream('mt19937ar','Seed',5489);
y = awgn(x,30,'measured',randStream);
```

Display the eye diagram.

```
ed(y)
```



Eye levels: -1.01 0.99
Eye amplitude: 2.00
Eye height: 1.80
Vertical opening: 1.53
Eye SNR (unitless): 31.97
Q factor (unitless): 29.96
Crossing levels: -0.01 -0.00

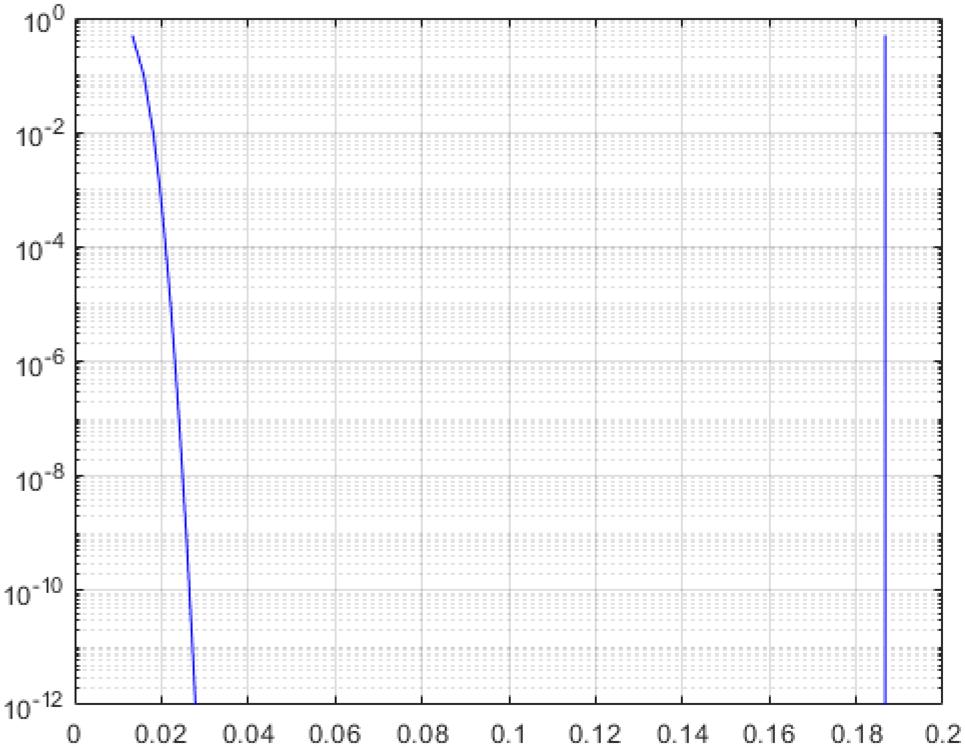
Crossing times: 118.82 319.18
Eye delay: 219.00
Eye width: 167.87
Horizontal opening: 157.82
Rise time: --
Fall time: --

Deterministic jitter: 26.59

Generate the horizontal bathtub data for the eye diagram. Plot the curve.

```
hb = horizontalBathtub(ed)
semilogy([hb.LeftThreshold],[hb.BER],'b', ...
    [hb.RightThreshold],[hb.BER],'b')
grid
```

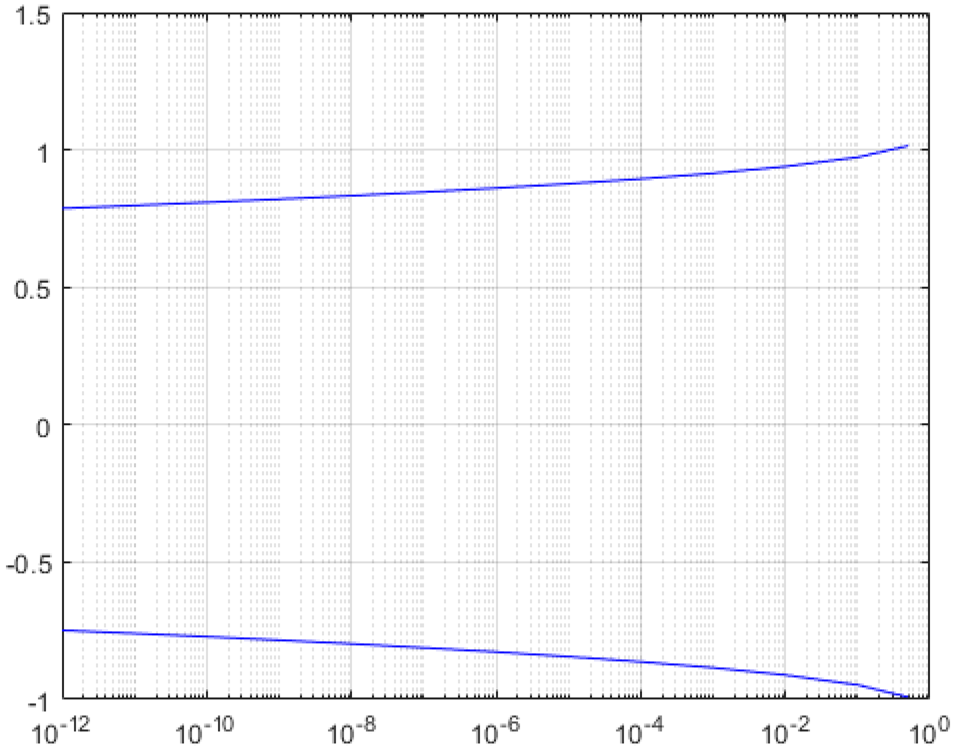
```
hb = 

  1x13 struct array with fields:

    BER
    LeftThreshold
    RightThreshold
```



Generate the vertical bathtub data for the eye diagram. Plot the curve.

```
vb = verticalBathtub(ed)
semilogx([vb.BER],[vb.LowerThreshold],'b', ...
    [vb.BER],[vb.UpperThreshold],'b')
grid
```

```
vb = 

  1x13 struct array with fields:

    BER
    UpperThreshold
    LowerThreshold
```



Input Arguments

ed — Eye Diagram System object
System object

Eye Diagram System object, where you get the bathtub curve information from.

Output Arguments

s — Structure containing information
struct

Structure containing information about the vertical bathtub curve.

BER — Bit error rate values
scalar

Bit error rate values, mapped on the Y-axis of the verticalBathtub plot against the corresponding `UpperThreshold` and `LowerThreshold` values on the x-axis, specified as a scalar.

Data Types: `double`

`UpperThreshold` — Upper threshold value
scalar

Upper threshold value, mapped on the x-axis in the plot against its corresponding BER value on the x-axis.

Data Types: `double`

`LowerThreshold` — Lower threshold value

scalar

Lower threshold value, mapped on the x-axis in the plot against its corresponding BER value on the x-axis.

Data Types: `double`

See Also

`horizontalBathtub`

Introduced in R2016b

comm.FMBroadcastDemodulator

Package: comm

Demodulate broadcast FM signal

Description

The comm.FMBroadcastDemodulator System object demodulates a complex baseband FM signal and filters the signal with a de-emphasis filter to produce an audio signal. If the Stereo property is set to true, the object performs stereo decoding. If the RBDS property is set to true, the object also demodulates the RDS/RBDS waveform. For more details, see "Algorithms" on page 3-609.

To demodulate a complex baseband FM signal:

1. Define and set up the comm.FMBroadcastDemodulator object. See "Construction" on page 3-605.
2. Call step to demodulate the complex baseband FM signal according to the properties of comm.FMBroadcastDemodulator.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

fmbDemod = comm.FMBroadcastDemodulator creates a demodulator System object, fmbDemod, that frequency demodulates an input signal.

fmbDemod = comm.FMBroadcastDemodulator(Name,Value) creates an FM demodulator object, fmbDemod, with each specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

fmbDemod = comm.FMBroadcastDemodulator(MOD) creates an FM demodulator object, fmbDemod, whose properties are determined by the corresponding FM modulator object, MOD.

Properties

SampleRate

Input signal sample rate (Hz)

Specify the sample rate of the input signal in Hz as a positive real scalar. The default value is 240e3. This property is nontunable.

FrequencyDeviation

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM demodulator in Hz as a positive real scalar. The default value is 75e3. System bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe. This property is nontunable.

FilterTimeConstant

Time constant of the de-emphasis filter (s)

Specify the de-emphasis lowpass filter time constant as a positive real scalar. The default value is 7.5e-05. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe. This property is nontunable.

AudioSampleRate

Audio sample rate of the output signal (Hz)

Specify the output audio sample rate as a positive real scalar. The default value is 48000. This property is nontunable.

PlaySound

Flag to enable or disable audio playback

To playback the output signal on the default audio device, set this property to true. The default is false. This property is nontunable.

BufferSize

Buffer size of the audio device

Specify the size of the buffer (in samples) that the object uses to communicate with an audio device as a positive scalar integer. The default is 4096. This property is available only when PlaySound is true. This property is nontunable.

Stereo

Flag to enable or disable stereo audio

Set this property to true to demodulate a stereophonic audio signal. Set to false if the input signal is monophonic. The default is false. This property is nontunable.

RBDS

Flag to demodulate RDS/RBDS waveform

If RBDS is set to true, the second output of the step method is the baseband RDS/RBDS waveform. The default value is false. This property is nontunable.

RBDSSamplesPerSymbol

Oversampling factor of RDS/RBDS output

Specify the number of samples of the RDS/RBDS output as a positive integer. The RDS/RBDS sample rate is given by RBDSSamplesPerSymbol × 1187.5 Hz. According to the RDS/RBDS standard, the sample rate of each bit is 1187.5 Hz.

This property applies only when you set RBDS to true.

The default is 10.

RBDSCostasLoop

Option to recover phase of RDS/RBDS signal

Specify whether a Costas loop is used to recover the phase of the RDS/RBDS signal. Set this option to `true` for radio stations that do not lock the `57` kHz RDS/RBDS signal in phase with the third harmonic of the `19` kHz pilot tone.

This property applies only when you set `RBDS` to `true`.

The default value is `false`.

Methods

info	Filter information about FM broadcast demodulator
reset	Reset states of the FM broadcast demodulator object
step	Apply FM broadcast demodulation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

FM Broadcast a Streaming Audio Signal

Modulate and demodulate a streaming audio signal with the FM broadcast modulator and demodulator objects. Play the audio signal using a default audio device.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an audio file reader System object™ and read the file `guitartune.wav`.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',4410);
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator. Set the `PlaySound` property of the demodulator to `true` to enable audio playback.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3,'PlaySound',true);
```

Read the audio data in frames of length 4410, apply FM broadcast modulation, demodulate the FM signal and playback the audio input.

```
while ~isDone(audio)
    audioData = audio();
    modData = fmbMod(audioData);
    demodData = fmbDemod(modData);
end
```

FM Modulate and Demodulate an RBDS Waveform

Generate a basic RBDS waveform, FM modulate it with an audio signal, and then demodulate it.

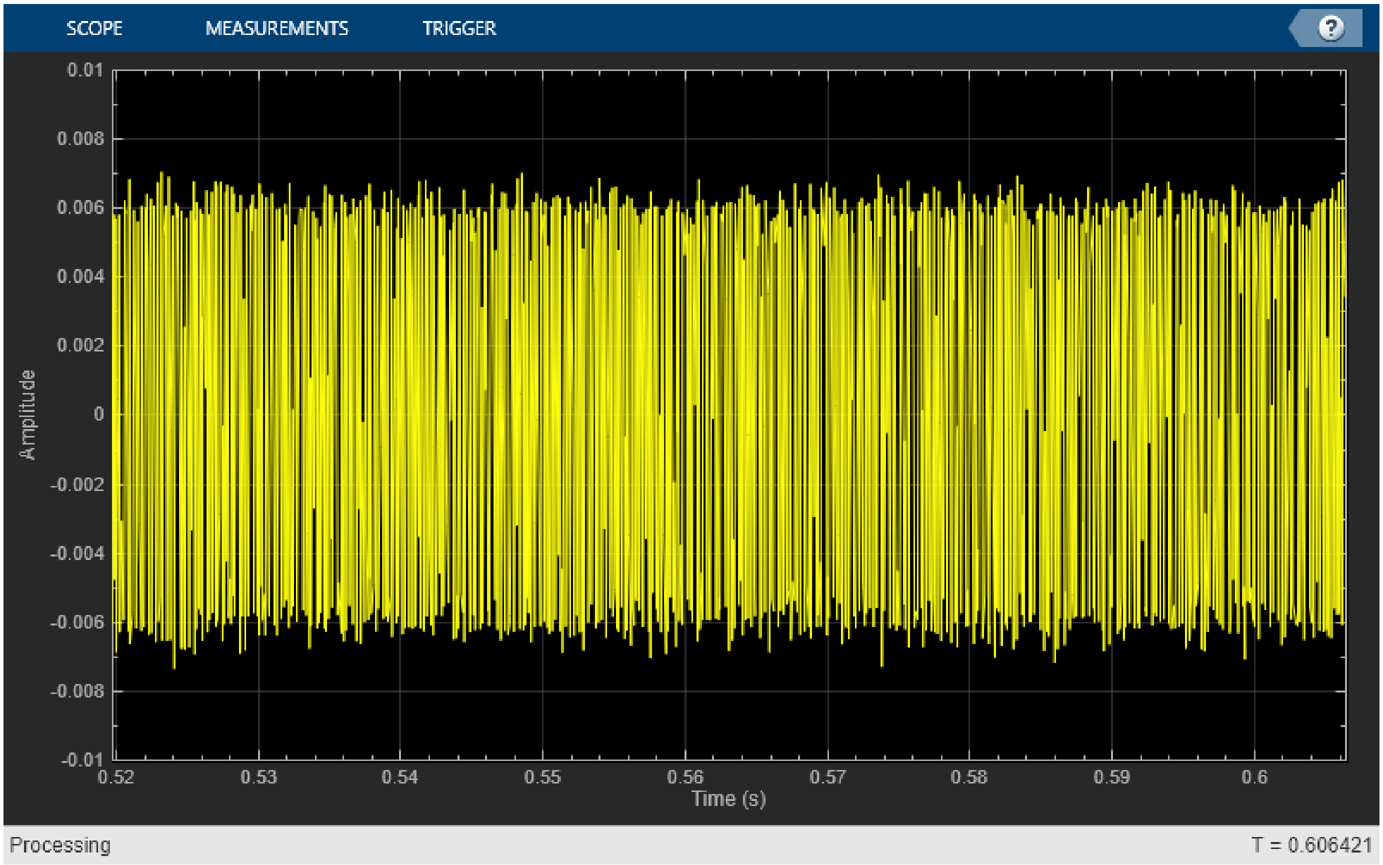
Note: This example runs only in R2017a or later.

Create a RBDS waveform with 19 groups per frame and 10 samples per symbol. The sample rate of the RBDS waveform is given by 1187.5 x 10. Set the audio sample rate to 1187.5 x 40.

```
groupLen = 104;
sps = 10;
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRate/rbdsRat
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sps);

fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = timescope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the current audio input. Generate RBDS information at the same configured rate as audio. FM modulate the stereo audio with RBDS information. Add additive white Gaussian noise. FM demodulate the audio and RBDS waveforms. View the waveforms in a time scope.

```
for idx = 1:7
    input = afr();
    rbdsWave = rbds();
    yFM = fmMod([input input], rbdsWave);
    rcv = awgn(yFM, 40);
    [audioRcv, rbdsRcv] = fmDemod(rcv);
    scope(rbdsRcv);
end
```

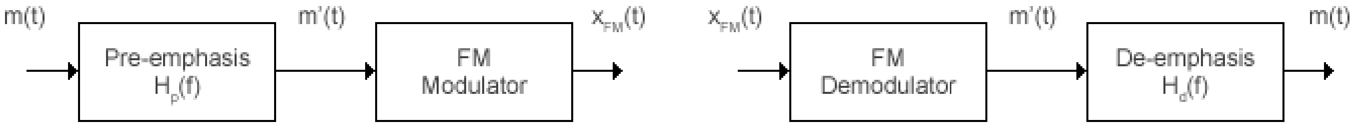


Algorithms

The FM Broadcast demodulator includes the functionality of the baseband FM demodulator, de-emphasis filtering, and the ability to receive stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMDemodulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



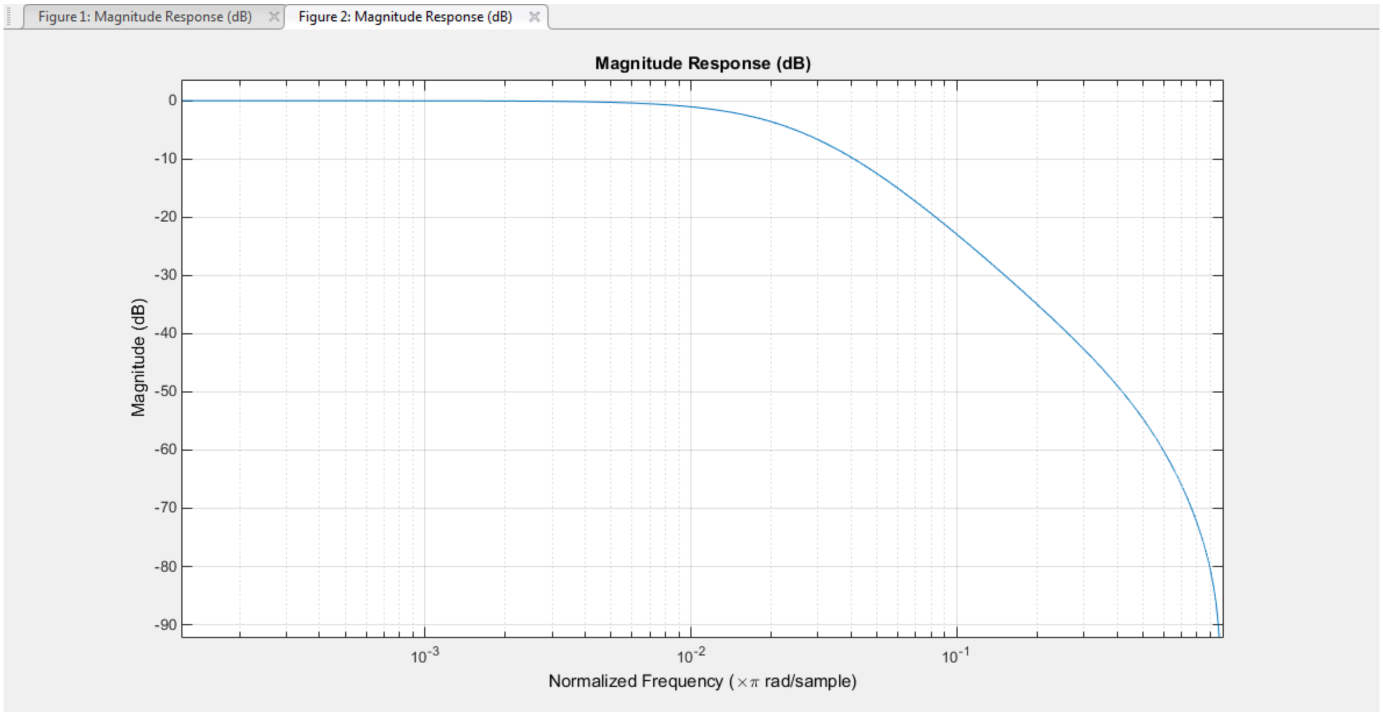
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s\,,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s} .$$

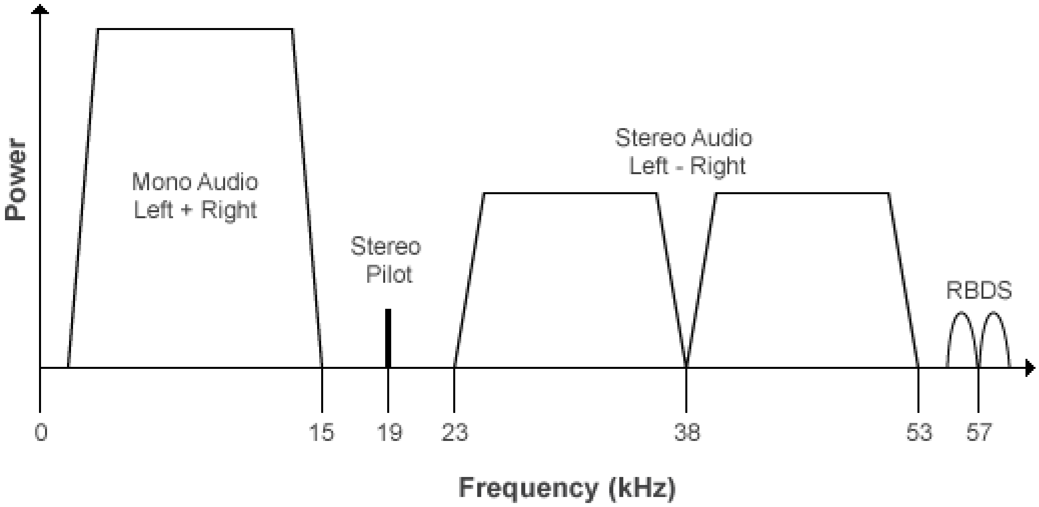
For an audio sample rate of 44.1 kHz, the de-emphasis filter has the following response.



Stereo and RDS/RBDS FM — Multiplex Signal

The FM broadcast demodulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals.

Here is the spectrum of the multiplex baseband signal, *m(t)*.



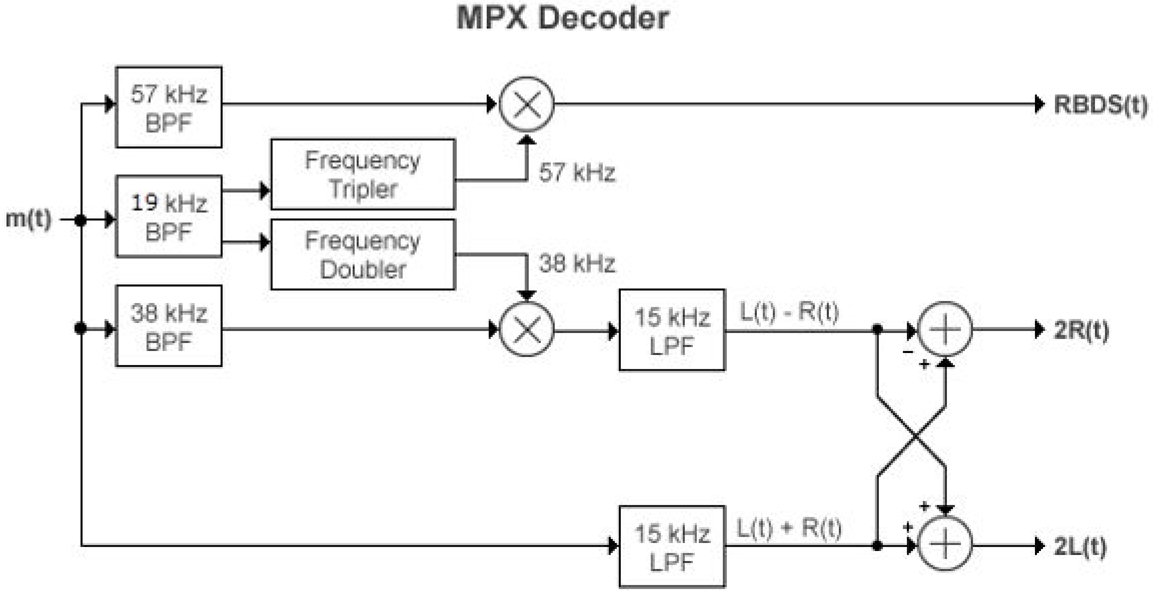
m(t) is given by

$$m(t) = C_0[L(t) + R(t)] + C_1\cos(2\pi \times 19kHz \times t) + C_0[L(t) - R(t)]\cos(2\pi \times 38kHz \times t) + C_2 RBDS(t)\cos(2\pi \times 57kHz \times t)\,,$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

The demodulator applies *m(t)* to three bandpass filters with center frequencies at 19, 38, and 57 kHz, and to a lowpass filter with a 3-dB cutoff frequency of 15 kHz. The 19 kHz bandpass filter extracts the pilot tone from the modulated signal. The recovered pilot tone is doubled and tripled in frequency to produce the 38 kHz and 57 kHz signals, which demodulate the (*L* – *R*) and RDS/RBDS signals, respectively. To generate a scaled version of the left and right channels that produce the stereo sound, the (*L* + *R*) and (*L* – *R*) signals are added and subtracted. The RDS/RBDS signal is recovered by mixing with the 57 kHz signal.

Here is the block diagram of the FM broadcast demodulator.



Limitations

The input length must be an integer multiple of the `AudioDecimationFactor` property. If `RBDS` is set to `true`, the input length in addition must be an integer multiple of `RBDSDecimationFactor`. For more information on these two properties, see the `info` method.

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.FMBroadcastModulator` | `comm.FMDemodulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Blocks

FM Broadcast Demodulator Baseband | FM Broadcast Modulator Baseband

Introduced in R2015a

info

System object: comm.FMBroadcastDemodulator
Package: comm

Filter information about FM broadcast demodulator

Syntax

```
S = info(fmbDemod)
```

Description

S = info(fmbDemod) returns a structure, S, containing this information for the comm.FMBroadcastDemodulator System object, fmbDemod:

Field	Description
AudioDecimationFactor	Decimation factor of the audio demodulator filter.
AudioInterpolationFactor	Interpolation factor of the audio demodulator filter.
RBDSDecimationFactor	Decimation factor of the RDS/RBDS demodulator filter.
RBDSInterpolationFactor	Interpolation factor of the RDS/RBDS demodulator filter.

Note When RBDS is true, the demodulator input sequence length must be a multiple of AudioDecimationFactor and RBDSDecimationFactor.

When RBDS is false, the demodulator input sequence length must be a multiple of AudioDecimationFactor.

Introduced in R2015a

reset

System object: `comm.FMBroadcastDemodulator`
Package: `comm`

Reset states of the FM broadcast demodulator object

Syntax

```
reset(fmbDemod)
```

Description

`reset(fmbDemod)` resets the states of the `comm.FMBroadcastDemodulator` object, `fmbDemod`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

Introduced in R2015a

step

System object: `comm.FMBroadcastDemodulator`
Package: `comm`

Apply FM broadcast demodulation

Syntax

```
audioSig = step(fmbDemod,X)
[audioSig,rbdsSig] = step(fmbDemod,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`audioSig = step(fmbDemod,X)` demodulates the complex baseband FM signal, `X`, and filters this signal with a de-emphasis filter to produce an audio signal, `audioSig`. If the `Stereo` property is set to `true`, stereo decoding is also performed. The output, `audioSig`, is a real vector with length equal to `(AudioSampleRate/SampleRate) × length(X)`.

`[audioSig,rbdsSig] = step(fmbDemod,X)` also demodulates the baseband RBDS signal, `rbdsSig`. The `step` method outputs the RBDS signal only if the `RBDS` property is set to `true`. The output, `rbdsSig`, is a real vector with length equal to `(RBDSSamplesPerSymbol × 1187.5/SampleRate) × length(X)`.

Note `fmbDemod` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2015a

comm.FMBroadcastModulator

Package: comm

Modulate broadcast FM signal

Description

The comm.FMBroadcastModulator System object pre-emphasizes an audio signal and modulates it onto a baseband FM signal. If the Stereo property is set to true, the object modulates the audio input (*L*–*R*) in the 38 kHz band, in addition to modulating it in the baseband (*L*+*R*). If the RBDS property is set to true, the object modulates a baseband RDS/RBDS signal at 57 kHz. For more details, see “Algorithms” on page 3-621.

To FM modulate an audio signal:

1. Define and set up the comm.FMBroadcastModulator object. See “Construction” on page 3-617.
2. Call step to apply broadcast FM modulation to an audio signal according to the properties of comm.FMBroadcastModulator.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

fmbMod = comm.FMBroadcastModulator creates a modulator System object, fmbMod, that frequency modulates an input signal.

fmbMod = comm.FMBroadcastModulator(demod) creates a broadcast FM modulator object whose properties are determined by the corresponding broadcast FM demodulator object, demod.

fmbMod = comm.FMBroadcastModulator(Name,Value) creates a broadcast FM modulator object with each specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SampleRate

Output signal sample rate (Hz)

Specify the sample rate of the output signal in Hz as a positive real scalar. The default value is 240e3. This property is nontunable.

FrequencyDeviation

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM modulator in Hz as a positive real scalar. The default value is 75e3. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe. This property is nontunable.

FilterTimeConstant

Filter time constant (s)

Specify the pre-emphasis highpass filter time constant as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe. The default value is 7.5e-05. The property is nontunable.

AudioSampleRate

Sample rate of the input audio signal (Hz)

Specify the audio sample rate as a positive real scalar. The default value is 48000. This property is nontunable.

Stereo

Flag to set stereo operations

Set this property to true if the input is a stereophonic audio signal. Set to false if the input signal is monophonic. The default is false. This property is nontunable.

RBDS

Flag to modulate RDS/RBDS waveform

If RBDS is set to true, the step method accepts the baseband RDS/RBDS waveform as its second input and the object modulates the signal at 57 kHz. The default value is false. This property is nontunable.

RBDSSamplesPerSymbol

Oversampling factor of RDS/RBDS input

Specify the number of samples per RDS/RBDS symbol as a positive integer. The RDS/RBDS sample rate is given by RBDSSamplesPerSymbol × 1187.5 Hz. According to the RDS/RBDS standard, the sample rate of each bit is 1187.5 Hz.

This property applies only when you set RBDS to true.

The default is 10.

Methods

info	Filter information about FM broadcast modulator
reset	Reset states of the FM broadcast modulator object
step	Apply FM broadcast modulation

Common to All System Objects	
release	Allow System object property value changes

Examples

FM Broadcast a Streaming Audio Signal

Modulate and demodulate a streaming audio signal with the FM broadcast modulator and demodulator objects. Play the audio signal using a default audio device.

Note: This example runs only in R2016b or later. If you are using an earlier release, replace each call to the function with the equivalent `step` syntax. For example, myObject(x) becomes step(myObject,x).

Create an audio file reader System object™ and read the file `guitartune.wav`.

```
audio = dsp.AudioFileReader('guitartune.wav','SamplesPerFrame',4410);
```

Create FM broadcast modulator and demodulator objects. Set the `AudioSampleRate` property to match the sample rate of the input signal. Set the `SampleRate` property of the demodulator to match the specified sample rate of the modulator. Set the `PlaySound` property of the demodulator to `true` to enable audio playback.

```
fmbMod = comm.FMBroadcastModulator('AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3);
fmbDemod = comm.FMBroadcastDemodulator( ...
    'AudioSampleRate',audio.SampleRate, ...
    'SampleRate',240e3,'PlaySound',true);
```

Read the audio data in frames of length 4410, apply FM broadcast modulation, demodulate the FM signal and playback the audio input.

```
while ~isDone(audio)
    audioData = audio();
    modData = fmbMod(audioData);
    demodData = fmbDemod(modData);
end
```

FM Modulate and Demodulate an RBDS Waveform

Generate a basic RBDS waveform, FM modulate it with an audio signal, and then demodulate it.

Note: This example runs only in R2017a or later.

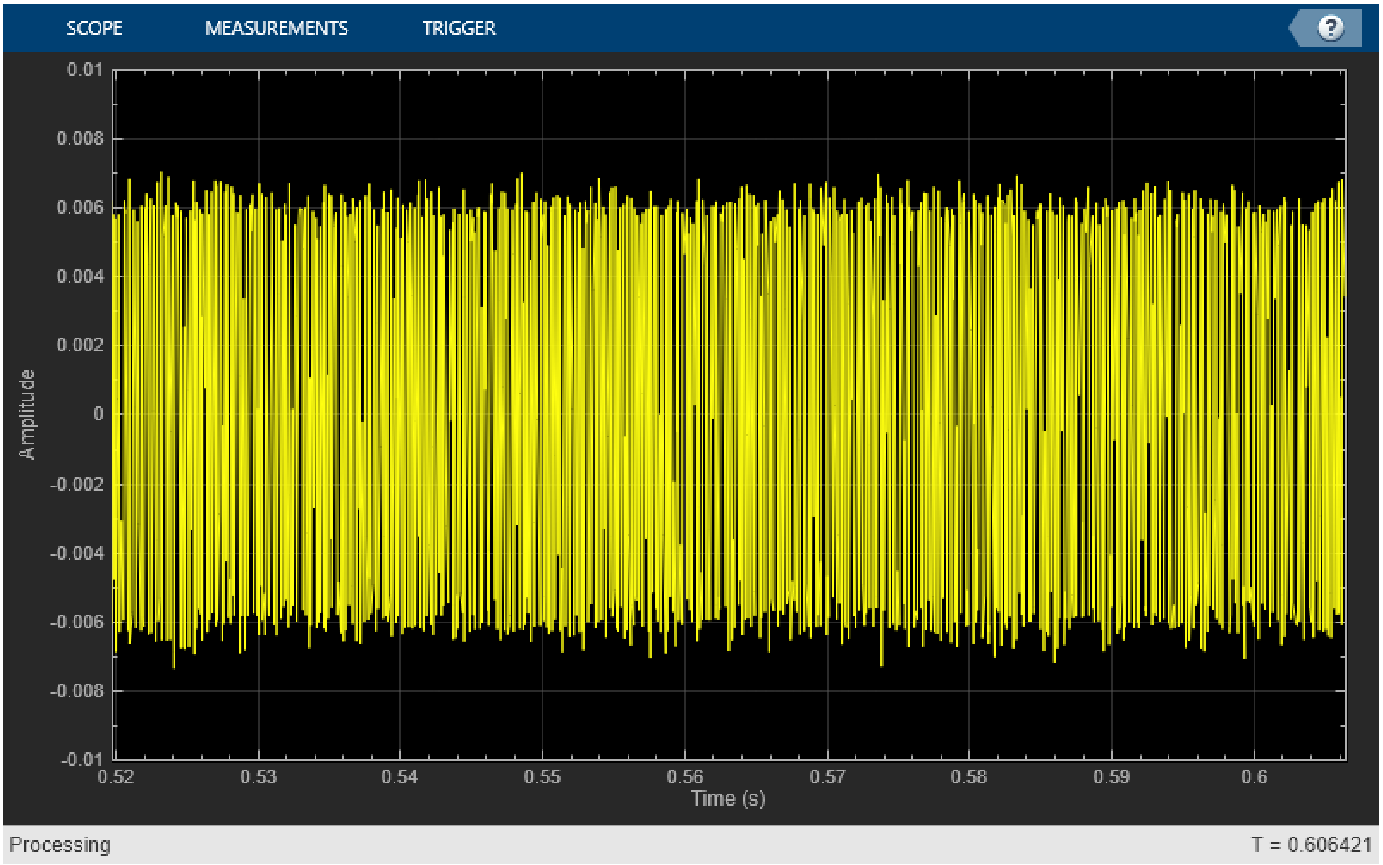
Create a RBDS waveform with 19 groups per frame and 10 samples per symbol. The sample rate of the RBDS waveform is given by 1187.5 x 10. Set the audio sample rate to 1187.5 x 40.

```
groupLen = 104;
sps = 10;
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
```

```
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRate/rbdsRa
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sps);

fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = timescope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the current audio input. Generate RBDS information at the same configured rate as audio. FM modulate the stereo audio with RBDS information. Add additive white Gaussian noise. FM demodulate the audio and RBDS waveforms. View the waveforms in a time scope.

```
for idx = 1:7
    input = afr();
    rbdsWave = rbds();
    yFM = fmMod([input input], rbdsWave);
    rcv = awgn(yFM, 40);
    [audioRcv, rbdsRcv] = fmDemod(rcv);
    scope(rbdsRcv);
end
```

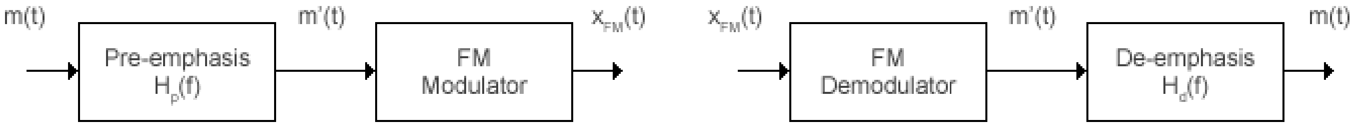


Algorithms

The FM Broadcast modulator includes the functionality of the baseband FM modulator, pre-emphasis filtering, and the ability to transmit stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMModulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



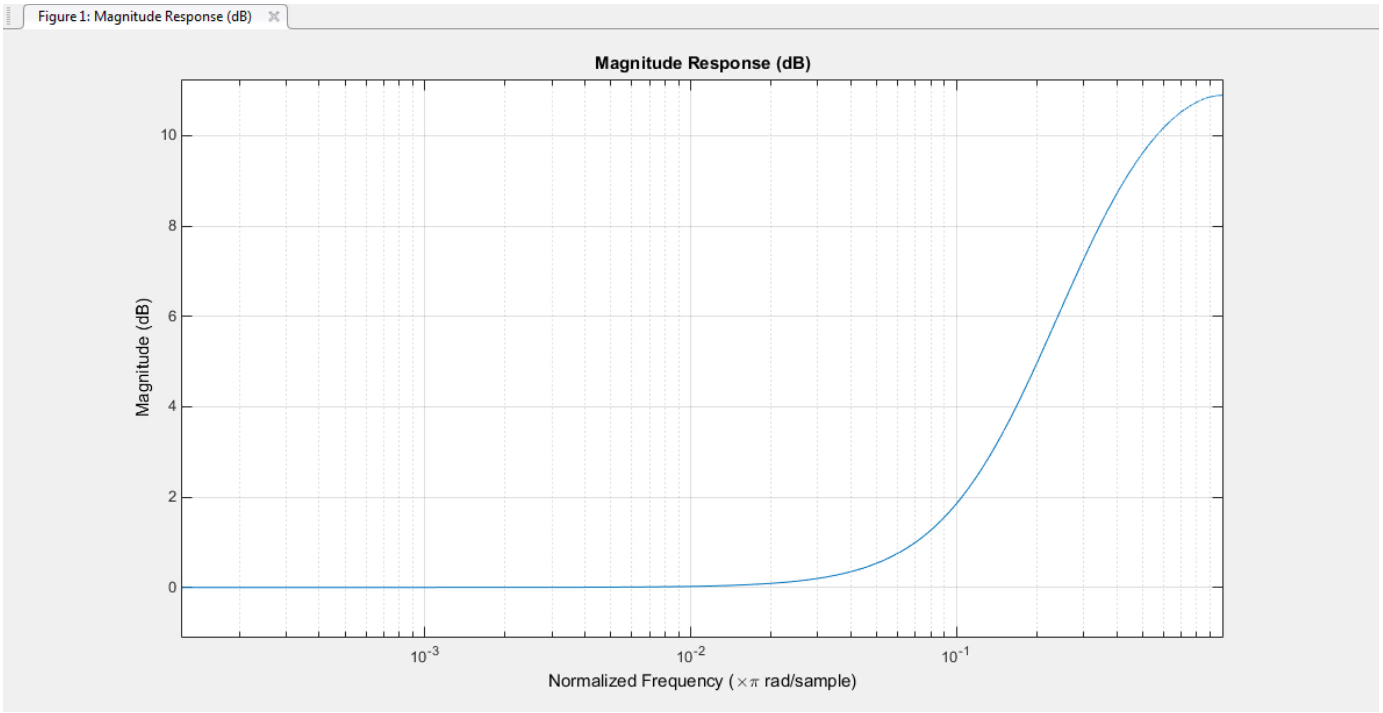
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s\,,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

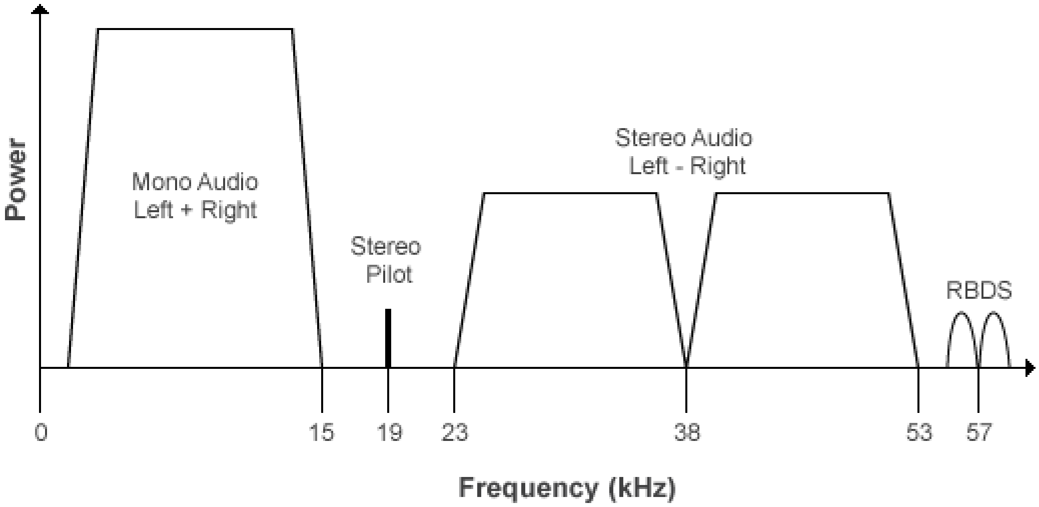
$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s} .$$

Irrespective of the audio sampling rate, the signal is converted to a 152 kHz output sampling rate. For an audio sample rate of 44.1 kHz, the pre-emphasis filter has the following response.

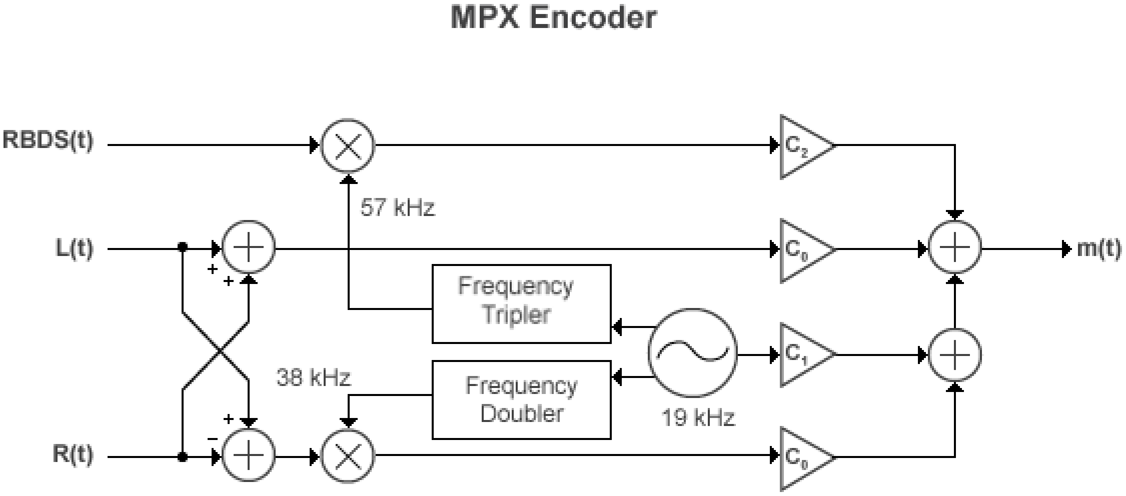


Stereo and RDS/RBDS FM – Multiplex Signal

The FM broadcast modulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals. Here is the spectrum of the multiplex baseband signal.



Here is the block diagram of the FM broadcast modulator, which is used to generate the multiplex baseband signal. *L(t)* and *R(t)* denote the time-domain waveforms from the left and right channels. *RBDS(t)* denotes the time-domain waveform of the RDS/RBDS signal.



The multiplex message signal, *m(t)* is given by

$$m(t) = C_0[L(t) + R(t)] + C_1\cos(2\pi \times 19kHz \times t) + C_0[L(t) - R(t)]\cos(2\pi \times 38kHz \times t) + C_2RBDS(t)\cos(2\pi \times 57kHz \times t) ,$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

Limitations

- If `RBDS` is `true`, both the audio and RDS/RBDS inputs must satisfy the following equation:

$$\frac{audioLength}{audioSampleRate} = \frac{RBDSLength}{RBDSSampleRate}$$

- The input length of the audio signal must be an integer multiple of the `AudioDecimationFactor` property. The input length of the RDS/RBDS signal must be an integer multiple of the `RBDSDecimationFactor` property. For more information on these two properties, see the `info` method.

References

[1] Chakrabarti, I. H., and Hatai, I. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems*. New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. "Frequency Modulation (FM) Tutorial". FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Objects

`comm.FMBroadcastDemodulator` | `comm.FMDemodulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Blocks

FM Broadcast Demodulator Baseband | FM Broadcast Modulator Baseband

Introduced in R2015a

info

System object: `comm.FMBroadcastModulator`
Package: `comm`

Filter information about FM broadcast modulator

Syntax

```
S = info(fmbMod)
```

Description

`S = info(fmbMod)` returns a structure, `S`, containing this information for the `comm.FMBroadcastModulator` System object, `fmbMod`:

Field	Description
`AudioDecimationFactor`	Decimation factor of the audio modulator filter.
`AudioInterpolationFactor`	Interpolation factor of the audio modulator filter.
`RBDSDecimationFactor`	Decimation factor of the RDS/RBDS modulator filter.
`RBDSInterpolationFactor`	Interpolation factor of the RDS/RBDS modulator filter.

Note The modulator input sequence length for the audio input must be a multiple of `AudioDecimationFactor`.

The modulator input sequence length for the RDS/RBDS input must be a multiple of `RBDSDecimationFactor`.

Introduced in R2015a

reset

System object: `comm.FMBroadcastModulator`
Package: `comm`

Reset states of the FM broadcast modulator object

Syntax

```
reset(fmbMod)
```

Description

`reset(fmbMod)` resets the states of the `comm.FMBroadcastModulator` object, `fmbMod`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

Introduced in R2015a

step

System object: `comm.FMBroadcastModulator`
Package: `comm`

Apply FM broadcast modulation

Syntax

```
modSig = step(fmbMod,audioSig)
modSig = step(fmbMod,audioSig,rbdsSig)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`modSig = step(fmbMod,audioSig)` pre-emphasizes the audio signal, `audioSig`, and modulates it onto a baseband FM signal. The audio signal can be real or complex with a single-precision or a double-precision data type. If the `Stereo` property of `fmbMod` is set to `true`, stereo encoding is performed after pre-emphasis and the audio signal must have at least two channels. If `Stereo` is `false`, the audio signal must be a column vector. The length of the modulated signal, `modSig`, is `(SampleRate/AudioSampleRate) × length(audioSig)`.

`modSig = step(fmbMod,audioSig,rbdsSig)` also modulates a baseband RBDS signal at 57 kHz. You can pass `rbdsSig` as an input only if you set the `RBDS` property to true. The length of output vector `modSig` is `(SampleRate/AudioSampleRate) × length(audioSig)`.

Note `fmbMod` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2015a

comm.FMDemodulator

Package: comm

Demodulate using FM method

Description

The FMDemodulator System object demodulates an FM modulated signal.

To FM demodulate a signal:

1. Define and set up the FMDemodulator object. See "Construction" on page 3-628.
2. Call step to FM demodulate a signal according to the properties of comm.FMDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.FMDemodulator creates a demodulator System object, H, that frequency demodulates an input signal.

H = comm.FMDemodulator(mod) creates an FM demodulator object whose properties are determined by the corresponding FM modulator object, mod.

H = comm.FMDemodulator(Name,Value) creates an FM demodulator object with each specified property Name set to the specified Value. Name must appear inside single quotes. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

FrequencyDeviation

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM demodulator in Hz as a positive real scalar. The default value is 75e3. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. This property is nontunable.

SampleRate

Sample rate of input signal (Hz)

Specify the sample rate in Hz as a positive real scalar. The default value is 240e3. The output sample rate is equal to the input sample rate. This property is nontunable.

Methods

reset	Reset states of the FM demodulator object
step	Applies FM baseband demodulation

Common to All System Objects	
release	Allow System object property value changes

Examples

FM Modulate and Demodulate a Sinusoidal Signal

Modulate and demodulate a sinusoidal signal. Plot the demodulated signal and compare it to the original signal.

Set the example parameters.

```
fs = 100; % Sample rate (Hz)
ts = 1/fs; % Sample period (s)
fd = 25; % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5 s and frequency 4 Hz.

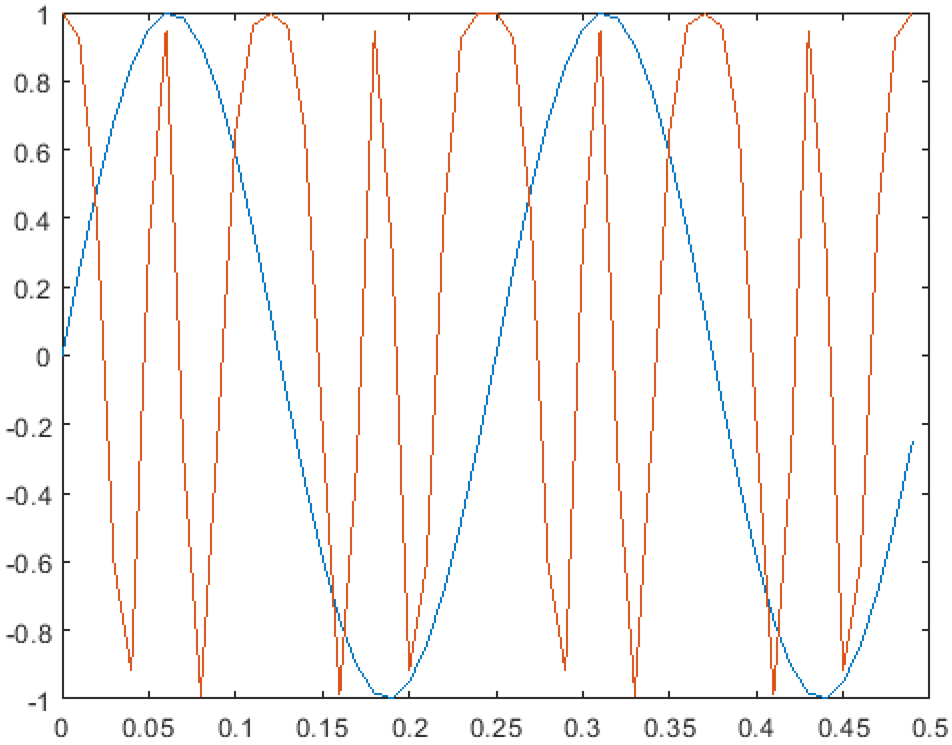
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create FM modulator and demodulator System objects.

```
fmmod = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
fmdemod = comm.FMDemodulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = fmmod(x);
plot(t,[x real(y)])
```

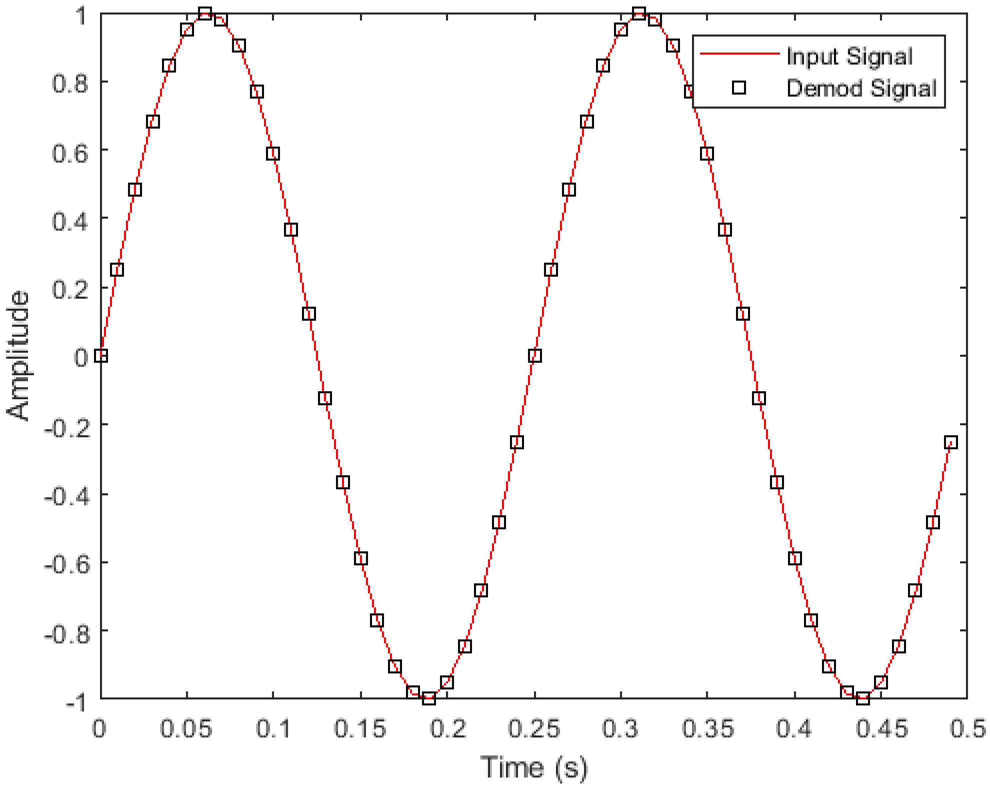


Demodulate the FM modulated signal.

```
z = fmdemod(y);
```

Plot the input and demodulated signals. The demodulator output signal exactly aligns with the input signal.

```
plot(t,x,'r',t,z,'ks')
legend('Input Signal','Demod Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



Create an FM Demodulator from an FM Modulator

Create an FM demodulator System object? from an FM modulator object. Modulate and demodulate audio data loaded from a file and compare its spectrum with that of the input data.

Set the example parameters.

```
fd = 50e3;                          % Frequency deviation (Hz)
fs = 300e3;                         % Sample rate (Hz)
```

Create an FM modulator System object.

```
MOD = comm.FMModulator('FrequencyDeviation',fd,'SampleRate',fs);
```

Create a companion demodulator object based on the modulator.

```
DEMOD = comm.FMDemodulator(MOD);
```

Verify that the properties are identical in the two System objects.

```
MOD
DEMOD
```

```
MOD =
  comm.FMModulator with properties:

             SampleRate: 300000
     FrequencyDeviation: 50000


DEMOD = 

  comm.FMDemodulator with properties:

             SampleRate: 300000
     FrequencyDeviation: 50000
```

Load audio data into structure variable, S.

```
S = load('handel.mat');
data = S.y;
fsamp = S.Fs;
```

Create a spectrum analyzer System object.

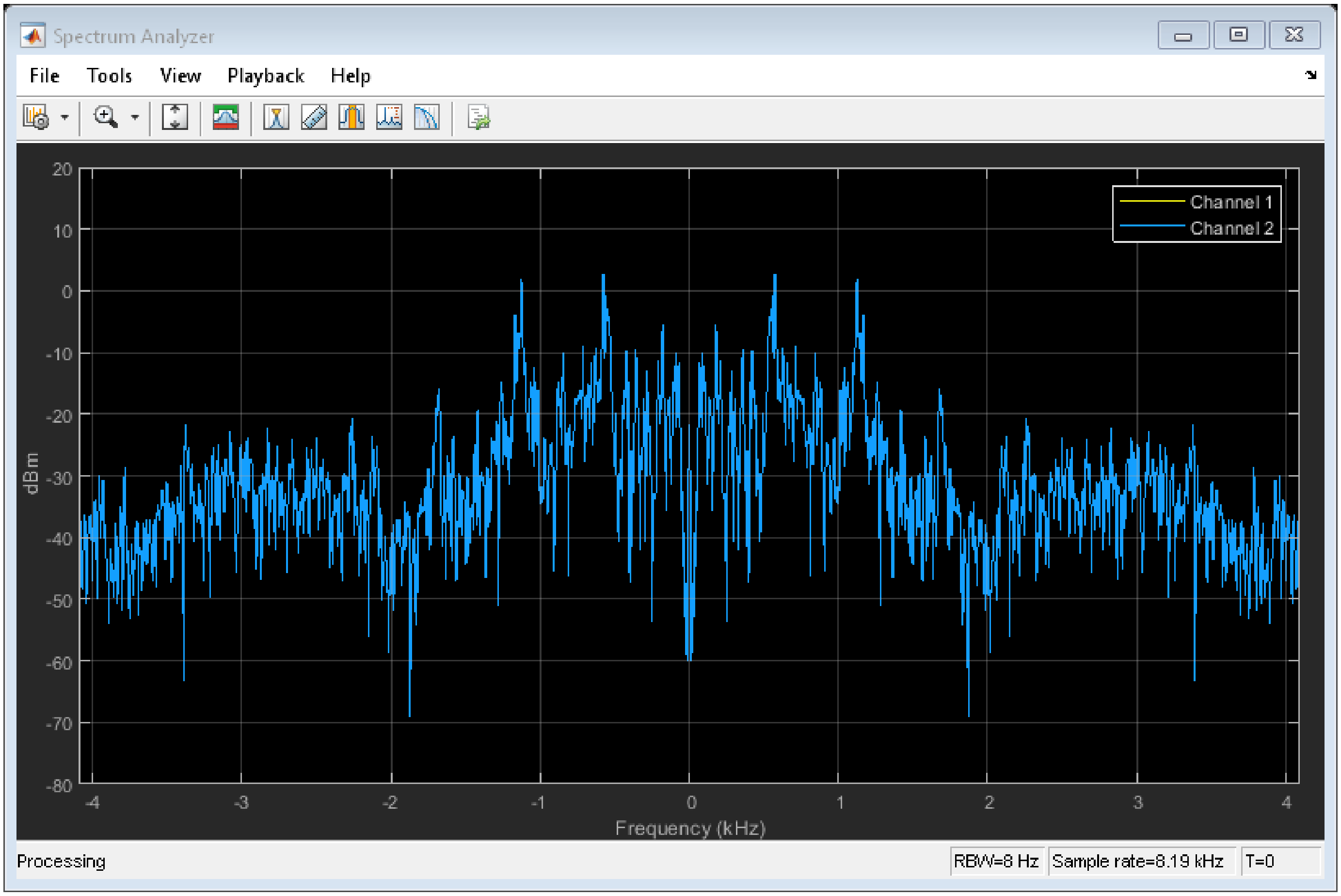
```
SA = dsp.SpectrumAnalyzer('SampleRate',fsamp,'ShowLegend',true);
```

FM modulate and demodulate the audio data.

```
modData = step(MOD,data);
demodData = step(DEMOD,modData);
```

Verify that the spectrum plot of the input data (Channel 1) is aligned with that of the demodulated data (Channel 2).

```
step(SA,[data demodData])
```



FM Modulate and Demodulate an Audio File

Playback an audio file after applying FM modulation and demodulation. The example takes advantage of the characteristics of System objects™ to process the data in streaming mode.

Load the audio file, `guitartune.wav`, using an audio file reader object.

```
AUDIO = dsp.AudioFileReader...
    ('guitartune.wav','SamplesPerFrame',4410);
```

Create an audio device writer object for audio playback.

```
AUDIOPLAYER = audioDeviceWriter;
```

Create modulator and demodulator objects having default properties.

```
MOD = comm.FMModulator;
DEMOD = comm.FMDemodulator;
```

Read audio data, FM modulate, FM demodulate, and playback the demodulated signal, `z`.

```
while ~isDone(AUDIO)
    x = step(AUDIO);                  % Read audio data
    y = step(MOD,x);                    % FM modulate
    z = step(DEMOD,y);                  % FM demodulate
    step(AUDIOPLAYER,z);                % Playback the demodulated signal
end
```

FM Modulate and Demodulate a Sinusoidal Signal

Modulate and demodulate a sinusoidal signal. Plot the demodulated signal and compare it to the original signal.

Set the example parameters.

```
fs = 100; % Sample rate (Hz)
ts = 1/fs; % Sample period (s)
fd = 25; % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5 s and frequency 4 Hz.

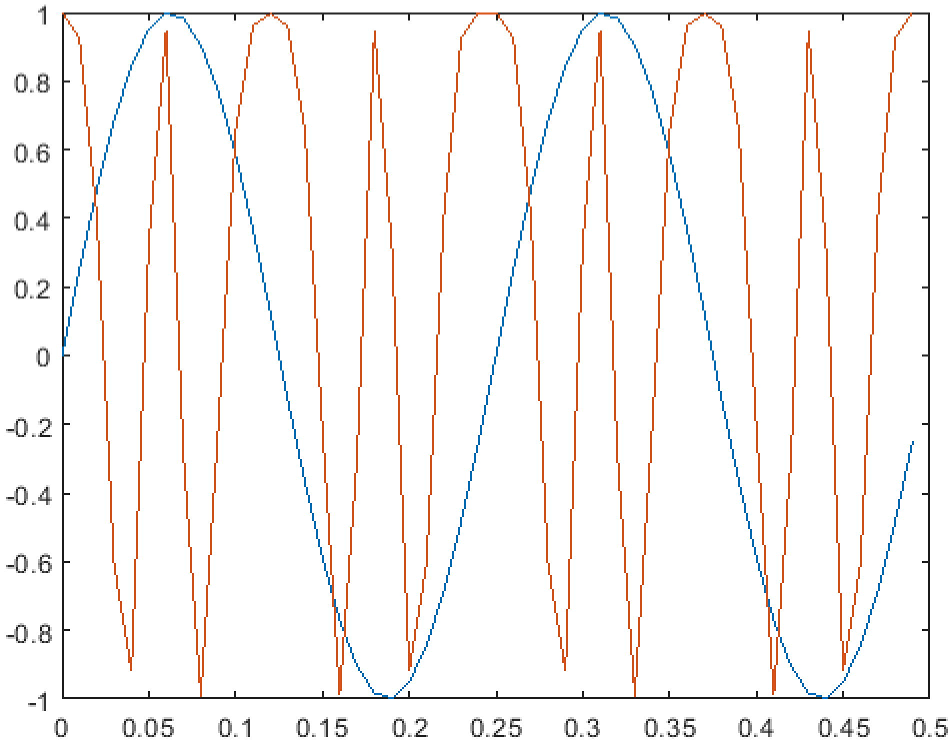
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create FM modulator and demodulator System objects.

```
fmmod = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
fmdemod = comm.FMDemodulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = fmmod(x);
plot(t,[x real(y)])
```

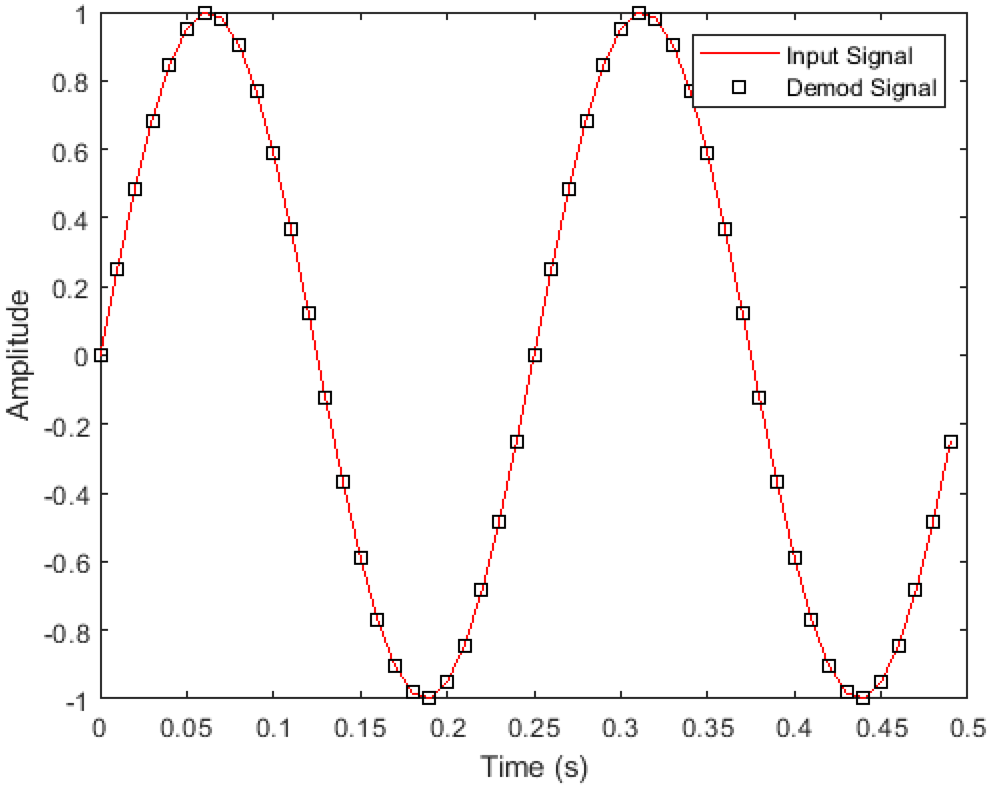


Demodulate the FM modulated signal.

```
z = fmdemod(y);
```

Plot the input and demodulated signals. The demodulator output signal exactly aligns with the input signal.

```
plot(t,x,'r',t,z,'ks')
legend('Input Signal','Demod Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



Selected Bibliography

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems*. New York: McGraw-Hill, 1971, pp. 142–155.

Algorithms

Represent a frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where A is the carrier amplitude, f_c is the carrier frequency, $x(\tau)$ is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming $|x(t)| \le 1$.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$y_s(t) = Y(t)e^{-j2\pi f_c t} = \frac{A}{2}\left[e^{j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t}$$
$$= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t - j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right].$$

Removing the component at -2f_c from $y_s(t)$ leaves the baseband signal representation, *y(t)*, which is expressed as

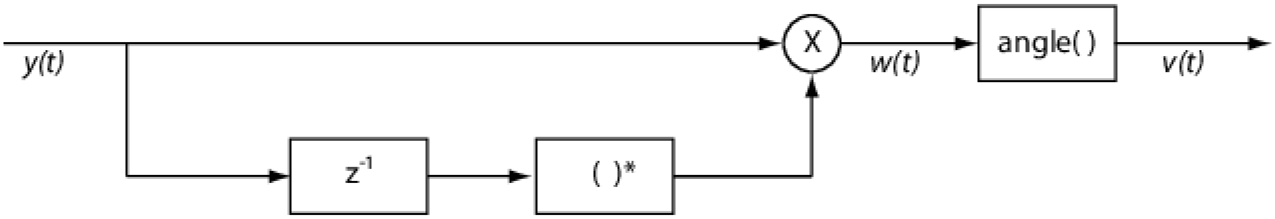
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)},$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, *ϕ(t)*.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself,

$$w(t) = \frac{A^2}{4}e^{j\phi(t)}e^{-j\phi(t-T)} = \frac{A^2}{4}e^{j[\phi(t) - \phi(t-T)]},$$

where *T* is the sample period. In discrete terms, w_n=*w(nT)*, and

$$w_n = \frac{A^2}{4}e^{j[\phi_n - \phi_{n-1}]},$$
$$v_n = \phi_n - \phi_{n-1}.$$

The signal v_n is the approximate derivative of ϕ_n, such that $v_n \approx x_n$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator` | `comm.FMModulator`

Introduced in R2015a

reset

System object: `comm.FMDemodulator`
Package: `comm`

Reset states of the FM demodulator object

Syntax

`reset(H)`

Description

`reset(H)` resets the states of the `FMDemodulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: `comm.FMDemodulator`
Package: `comm`

Applies FM baseband demodulation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` frequency demodulates an input signal, `X`, and returns an output signal, `Y`. The input `X` is real or complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` is real and has the same data type and dimensions as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.FMModulator

Package: `comm`

Modulate using FM method

Description

The `FMModulator` System object applies FM modulation to an input signal.

To FM modulate a signal:

1. Define and set up the `FMModulator` object. See "Construction" on page 3-641.
2. Call `step` to apply FM modulation to a signal according to the properties of `comm.FMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.FMModulator` creates a modulator System object, `H`, that frequency modulates an input signal.

`H = comm.FMModulator(demod)` creates an FM modulator object whose properties are determined by the corresponding FM demodulator object, `demod`.

`H = comm.FMModulator(Name,Value)` creates an FM modulator object with each specified property `Name` set to the specified `Value`. `Name` must appear inside single quotes. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`FrequencyDeviation`

Peak deviation of the output signal frequency (Hz)

Specify the frequency deviation of the FM modulator in Hz as a positive real scalar. The default value is `75e3`. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. This property is nontunable.

`SampleRate`

Sample rate of the input signal (Hz)

Specify the sample rate in Hz as a positive real scalar. The default value is `240e3`. The output sample rate is equal to the input sample rate. This property is nontunable.

Methods

reset	Reset states of the FM modulator object
step	Applies FM baseband modulation

Common to All System Objects	
release	Allow System object property value changes

Examples

FM Modulate a Sinusoidal Signal

Apply baseband modulation to a sine wave input signal and plot its response.

Set the example parameters.

```
fs = 1e3;               % Sample rate (Hz)
ts = 1/fs;              % Sample period (s)
fd = 50;                % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5s and frequency 4 Hz.

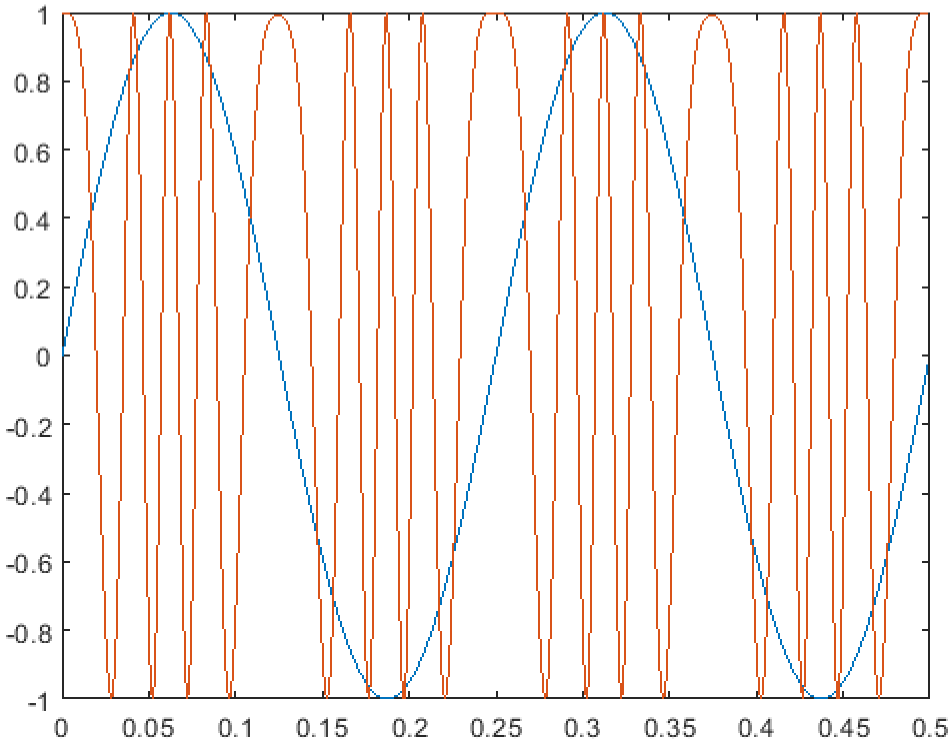
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create an FM modulator System object™.

```
MOD = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = step(MOD,x);
plot(t,[x real(y)])
```



Plot Spectrum of FM Modulated Baseband Signal

Apply FM baseband modulation to a white Gaussian noise source and plot its spectrum.

Set the example parameters.

```
fs = 1e3;                % Sample rate (Hz)
ts = 1/fs;               % Sample period (s)
fd = 10;                 % Frequency deviation (Hz)
```

Create a white Gaussian noise source having a duration of 5s.

```
t = (0:ts:5-ts)';
x = wgn(length(t),1,0);
```

Create an FM modulator System object? and modulate the input signal.

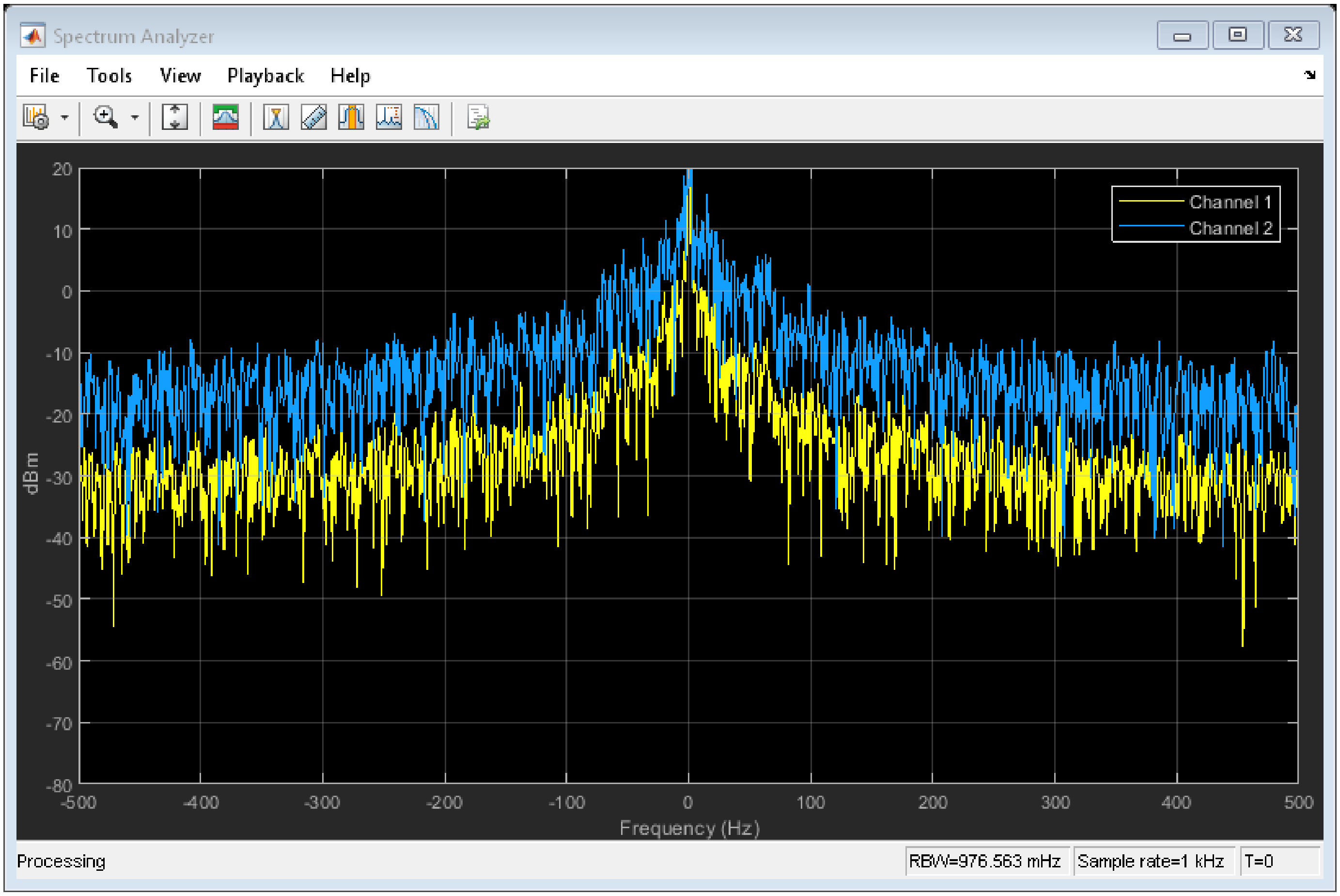
```
MOD1 = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
y = step(MOD1,x);
```

Create another modulator object, `MOD2`, whose frequency deviation is five times larger and apply FM modulation.

```
MOD2 = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',5*fd);
z = step(MOD2,x);
```

Plot the spectra of the two modulated signals. The larger frequency deviation associated with channel 2 results in a noise level that is 10 dB higher.

```
SA = dsp.SpectrumAnalyzer('SampleRate',fs,'ShowLegend',true);
step(SA,[y z])
```



FM Modulate and Demodulate a Sinusoidal Signal

Modulate and demodulate a sinusoidal signal. Plot the demodulated signal and compare it to the original signal.

Set the example parameters.

```
fs = 100; % Sample rate (Hz)
ts = 1/fs; % Sample period (s)
fd = 25; % Frequency deviation (Hz)
```

Create a sinusoidal input signal with duration 0.5 s and frequency 4 Hz.

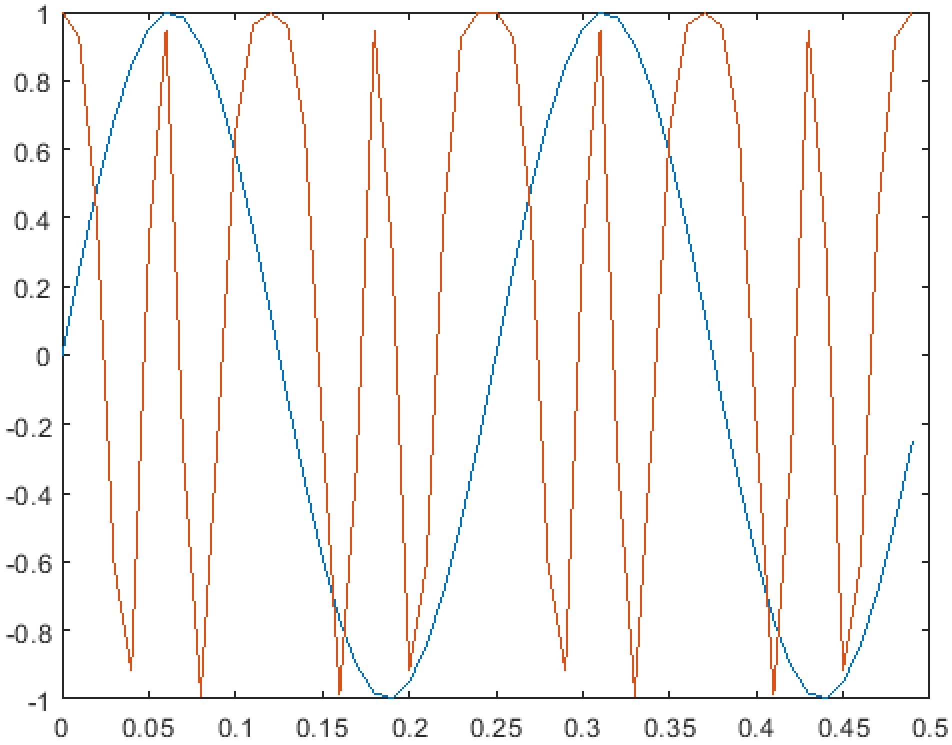
```
t = (0:ts:0.5-ts)';
x = sin(2*pi*4*t);
```

Create FM modulator and demodulator System objects.

```
fmmod = comm.FMModulator('SampleRate',fs,'FrequencyDeviation',fd);
fmdemod = comm.FMDemodulator('SampleRate',fs,'FrequencyDeviation',fd);
```

FM modulate the input signal and plot its real part. You can see that the frequency of the modulated signal changes with the amplitude of the input signal.

```
y = fmmod(x);
plot(t,[x real(y)])
```

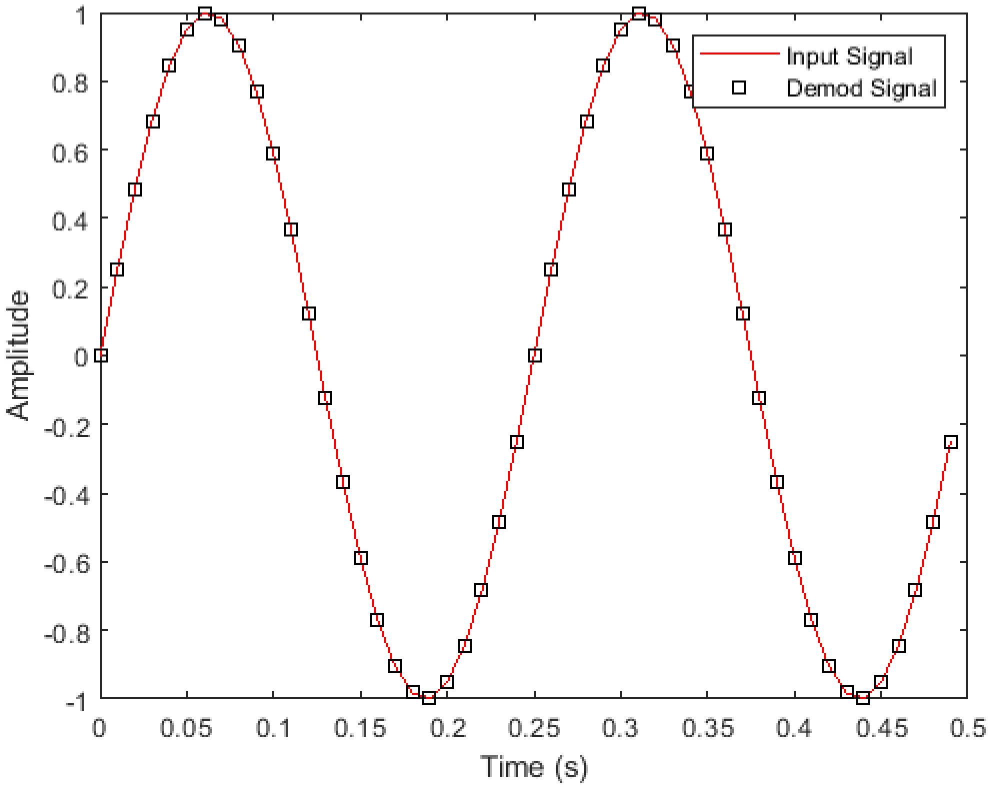


Demodulate the FM modulated signal.

```
z = fmdemod(y);
```

Plot the input and demodulated signals. The demodulator output signal exactly aligns with the input signal.

```
plot(t,x,'r',t,z,'ks')
legend('Input Signal','Demod Signal')
xlabel('Time (s)')
ylabel('Amplitude')
```



Selected Bibliography

[1] Chakrabarti, I. H., and Hatai, I. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems*. New York: McGraw-Hill, 1971, pp. 142–155.

Algorithms

Represent a frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where A is the carrier amplitude, f_c is the carrier frequency, $x(\tau)$ is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming $|x(t)| \leq 1$.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$y_s(t) = Y(t)e^{-j2\pi f_c t} = \frac{A}{2}\left[e^{j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t}$$
$$= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t - j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right].$$

Removing the component at -2f_c from *$y_s(t)$* leaves the baseband signal representation, *y(t)*, which is expressed as

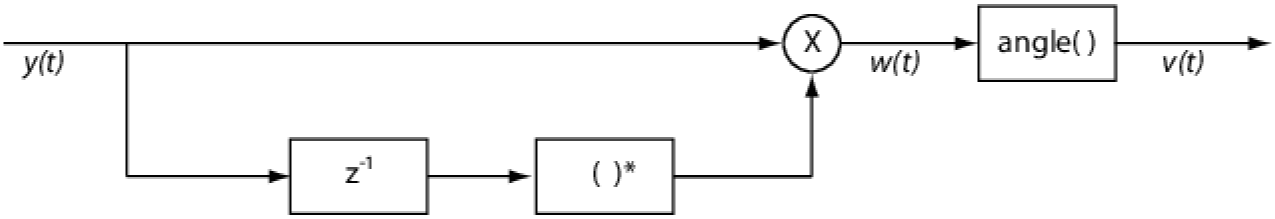
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)},$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, *ϕ(t)*.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself,

$$w(t) = \frac{A^2}{4}e^{j\phi(t)}e^{-j\phi(t-T)} = \frac{A^2}{4}e^{j[\phi(t) - \phi(t-T)]},$$

where *T* is the sample period. In discrete terms, *w_n=w(nT)*, and

$$w_n = \frac{A^2}{4}e^{j[\phi_n - \phi_{n-1}]},$$
$$v_n = \phi_n - \phi_{n-1}.$$

The signal v_n is the approximate derivative of ϕ_n, such that $v_n \approx x_n$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator` | `comm.FMDemodulator`

Introduced in R2015a

reset

System object: `comm.FMModulator`
Package: `comm`

Reset states of the FM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FMModulator` object, `H`.

This method resets the windowed suffix from the last symbol in the previously processed frame.

step

System object: `comm.FMModulator`
Package: `comm`

Applies FM baseband modulation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` frequency modulates an input signal, `X`, and returns a modulated signal, `Y`. The input `X` is real or complex and can be either a scalar or a column vector. Double- and single-precision data types are supported. The output `Y` has the same data type and dimensions as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

---

comm.FSKDemodulator

Package: comm

Demodulate using M-ary FSK method

Description

The FSKDemodulator object demodulates a signal that was modulated using the M-ary frequency shift keying method. The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals.

To demodulate a signal that was modulated using frequency shift keying:

1. Define and set up your FSK demodulator object. See “Construction” on page 3-651.
2. Call step to demodulate a signal according to the properties of FSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.FSKDemodulator creates a demodulator System object, H. This object demodulates an M-ary frequency shift keying (M-FSK) signal using a noncoherent energy detector.

H = comm.FSKDemodulator(Name,Value) creates an M-FSK demodulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.FSKDemodulator(M,FREQSEP,RS,Name,Value) creates an M-FSK demodulator object, H. This object has the ModulationOrder property set to M, the FrequencySeparation property set to FREQSEP, the SymbolRate property set to RS, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of frequencies in modulated signal

Specify the number of frequencies in the modulated signal as a numeric, positive, integer scalar value that is a power of two. The default is 8.

BitOutput

Output data as bits

Specify whether the output is groups of bits or integer values. The default is false.

When you set this property to `false`, the `step` method outputs a column vector of length equal to *N*/ `SamplesPerSymbol` on page 3-0 . *N* is the length of the input data vector to the step method. The elements of the output vector are integers between `0` and `ModulationOrder` on page 3-0 –1. When you set this property to `true`, the `step` method outputs a column vector of length equal to **log2**(`ModulationOrder`)×(*N*/`SamplesPerSymbol`). The property's elements are bit representations of integers between `0` and `ModulationOrder`–1.

`SymbolMapping`

Symbol encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses Gray-coded ordering.

When you set this property to `Binary`, the object uses natural binary-coded ordering.

For either type of mapping, the object maps the lowest frequency to the integer 0 and maps the highest frequency to the integer *M*–1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

`FrequencySeparation`

Frequency separation between successive tones

Specify the frequency separation between successive symbols in the modulated signal in Hertz as a positive, real scalar value. The default is `6` Hz.

`SamplesPerSymbol`

Number of samples per input symbol

Specify the number of samples per input symbol as a positive, integer scalar value. The default is `17`.

`SymbolRate`

Symbol duration

Specify the symbol rate in symbols per second as a positive, double-precision, real scalar value. The default is `100`. To avoid output signal aliasing, specify an output sampling rate, *Fs* = `SamplesPerSymbol` on page 3-0 ×`SymbolRate`, which is greater than `ModulationOrder` on page 3-0 × `FrequencySeparation` on page 3-0 . The symbol duration remain the same, regardless of whether the input is bits or integers.

`OutputDataType`

Data type of output

Specify the output data type as one of `logical` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `double`. The default is `double`. The `logical` type is valid only when you set the `BitOutput` on page 3-0 property to false and the `ModulationOrder` on page 3-0 property to two. When you set the `BitOutput` property to true, the output data requires a type of `logical` | `double`.

Methods

reset	Reset states of M-FSK demodulator object
step	Demodulate using M-ary FSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

FSK Modulation and Demodulation in AWGN

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz.

Set the modulation order and frequency separation parameters.

```
M = 8;
freqSep = 100;
```

Create FSK modulator and demodulator System objects™ with modulation order 8 and 100 Hz frequency separation.

```
fskMod = comm.FSKModulator(M,freqSep);
fskDemod = comm.FSKDemodulator(M,freqSep);
```

Create an additive white Gaussian noise channel, where the noise is specified as a signal-to-noise ratio.

```
ch = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',-2);
```

Create an error rate calculator object.

```
err = comm.ErrorRate;
```

Transmit one hundred 50-symbol frames using 8-FSK in an AWGN channel.

```
for counter = 1:100
    data = randi([0 M-1],50,1);
    modSignal = step(fskMod,data);
    noisySignal = step(ch,modSignal);
    receivedData = step(fskDemod,noisySignal);
    errorStats = step(err,data,receivedData);
end
```

Display the error statistics.

```
es = 'Error rate = %4.2e\nNumber of errors = %d\nNumber of symbols = %d\n';
fprintf(es,errorStats)
```

```
Error rate = 1.40e-02
Number of errors = 70
Number of symbols = 5000
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-FSK Demodulator Baseband block reference page. The object properties correspond to the block parameters, except:

- The **Symbol set ordering** parameter corresponds to the `SymbolMapping` on page 3-0 property.
- The `SymbolRate` on page 3-0 property replaces the block sample rate capability.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.CPFSKDemodulator` | `comm.CPFSKModulator` | `comm.FSKModulator`

Introduced in R2012a

reset

System object: `comm.FSKDemodulator`
Package: `comm`

Reset states of M-FSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FSKDemodulator` object, `H`.

step

System object: `comm.FSKDemodulator`
Package: `comm`

Demodulate using M-ary FSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the FSK demodulator System object, `H`, and returns `Y`. `X` must be a double or single precision data type column vector of length equal to an integer multiple of the number of samples per symbol that you specify in the `SamplesPerSymbol` property. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.FSKModulator

Package: `comm`

Modulate using M-ary FSK method

Description

The `FSKModulator` object modulates using the M-ary frequency shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using frequency shift keying:

1. Define and set up your FSK modulator object. See “Construction” on page 3-657.
2. Call `step` to modulate a signal according to the properties of `comm.FSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.FSKModulator` creates a modulator System object, `H`. This object modulates the input signal using the M-ary frequency shift keying (M-FSK) method.

`H = comm.FSKModulator(Name,Value)` creates an M-FSK modulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.FSKModulator(M,FREQSEP,RS,Name,Value)` creates an M-FSK modulator object, `H`. This object has the `ModulationOrder` property set to `M`, the `FrequencySeparation` property set to `FREQSEP`, the `SymbolRate` property set to `RS`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Number of frequencies in modulated signal

Specify the number of frequencies in the modulated signal as a numeric positive integer scalar value that is a power of two. The default is `8`.

`BitInput`

Assume bit inputs

Specify whether the input is bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input requires a numeric (except single precision data type) column vector of integer values between `0` and `ModulationOrder` on page 3-0 –1. In this case, the input vector can also be of data type logical if `ModulationOrder` equals `2`.

When you set this property to `true`, the `step` method input requires a double-precision or logical data type column vector of bit values. The length of this vector is an integer multiple of **log2**(`ModulationOrder`). This vector contains bit representations of integers between `0` and `ModulationOrder`–1.

SymbolMapping

Symbol encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses Gray-coded ordering.

When you set this property to `Binary`, the object uses natural binary-coded ordering. For either type of mapping, the object maps the lowest frequency to the integer 0 and maps the highest frequency to the integer *M*–1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

FrequencySeparation

Frequency separation between successive tones

Specify the frequency separation between successive tones in the modulated signal in Hertz as a positive, real scalar value. The default is `6` Hz. To avoid output signal aliasing, specify an output sampling rate, *Fs* = `SamplesPerSymbol` on page 3-0 ×`SymbolRate` on page 3-0 , which is greater than `ModulationOrder` on page 3-0 multiplied by `FrequencySeparation` on page 3-0 .

ContinuousPhase

Phase continuity

Specify if the phase of the output modulated signal is continuous or discontinuous. The default is `true`.

When you set this property to `true`, the modulated signal maintains continuous phase even when its frequency changes.

When you set this property to `false`, the modulated signal comprises portions of `ModulationOrder` on page 3-0 sinusoids of different frequencies. In this case, a change in the input value can cause a discontinuous change in the phase of the modulated signal.

SamplesPerSymbol

Number of samples per output symbol

Specify the number of output samples that the object produces for each integer or binary word in the input as a positive, integer scalar value. The default is `17`.

SymbolRate

Symbol duration

Specify the symbol rate in symbols per second as a positive, double-precision, real scalar. The default is `100`. To avoid output signal aliasing, specify an output sampling rate, *Fs* = `SamplesPerSymbol` on page 3-0 ×`SymbolRate`, which is greater than `ModulationOrder` on page 3-0 × `FrequencySeparation` on page 3-0 . The symbol duration remain the same, regardless of whether the input is bits or integers.

OutputDataType

Data type of output

Specify the output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of M-FSK modulator object
step	Modulate using M-ary FSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

FSK Modulation and Demodulation in AWGN

Modulate and demodulate a signal using 8-FSK modulation with a frequency separation of 100 Hz.

Set the modulation order and frequency separation parameters.

```
M = 8;
freqSep = 100;
```

Create FSK modulator and demodulator System objects™ with modulation order 8 and 100 Hz frequency separation.

```
fskMod = comm.FSKModulator(M,freqSep);
fskDemod = comm.FSKDemodulator(M,freqSep);
```

Create an additive white Gaussian noise channel, where the noise is specified as a signal-to-noise ratio.

```
ch = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',-2);
```

Create an error rate calculator object.

```
err = comm.ErrorRate;
```

Transmit one hundred 50-symbol frames using 8-FSK in an AWGN channel.

```
for counter = 1:100
    data = randi([0 M-1],50,1);
    modSignal = step(fskMod,data);
    noisySignal = step(ch,modSignal);
    receivedData = step(fskDemod,noisySignal);
    errorStats = step(err,data,receivedData);
end
```

Display the error statistics.

```
es = 'Error rate = %4.2e\nNumber of errors = %d\nNumber of symbols = %d\n';
fprintf(es,errorStats)
```

```
Error rate = 1.40e-02
Number of errors = 70
Number of symbols = 5000
```

Visualize FSK Modulated Symbol Mapping

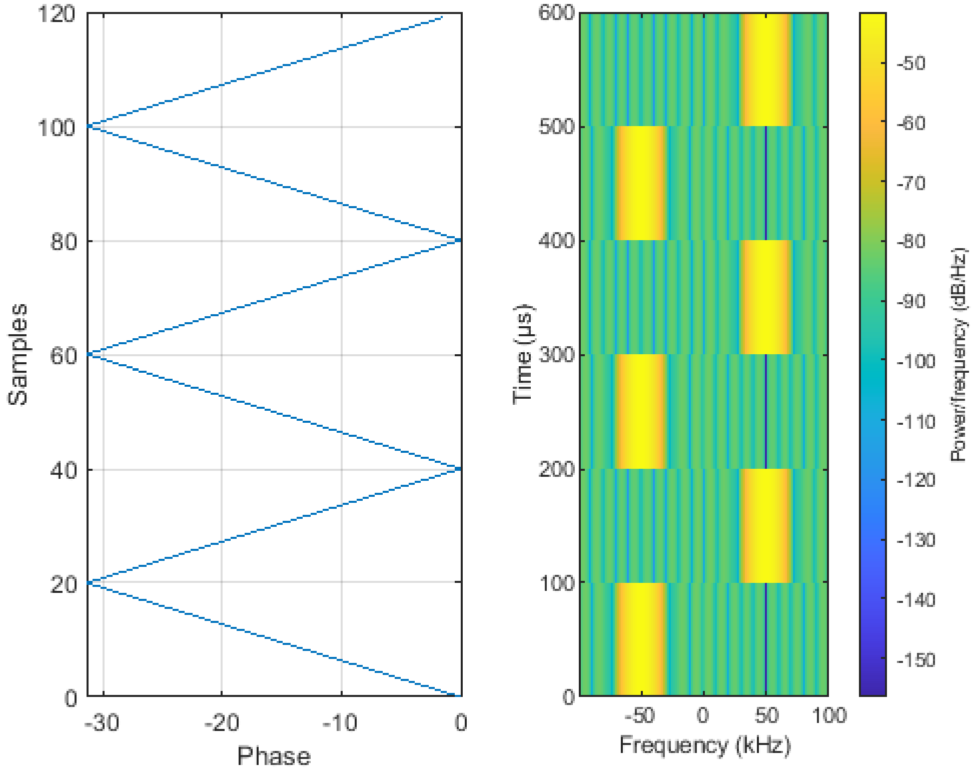
Visualize symbol mapping of an FSK modulated signal with a spectrogram.

Specify 20 samples for each symbol. 0 maps to -50 kHz (negative phase slope) and 1 maps to +50 kHz (positive phase slope).

```
mod = comm.FSKModulator;
mod.ModulationOrder = 2;
mod.FrequencySeparation = 100000;
mod.SamplesPerSymbol = 20;
mod.SymbolMapping = 'Gray';
mod.SymbolRate = 1e4
```

```
mod = 
  comm.FSKModulator with properties:

        ModulationOrder: 2
               BitInput: false
          SymbolMapping: 'Gray'
    FrequencySeparation: 100000
        ContinuousPhase: true
       SamplesPerSymbol: 20
             SymbolRate: 10000
         OutputDataType: 'double'
```

```
x = mod([0 1 0 1 0 1]');
figure; subplot(1,2,1); plot(unwrap(angle(x)),0:length(x)-1);
grid on; xlabel('Phase'); ylabel('Samples')
subplot(1,2,2);
spectrogram(x,20,0,[],mod.SymbolRate*mod.SamplesPerSymbol,'centered')
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the M-FSK Modulator Baseband block reference page. The object properties correspond to the block parameters, except:

- The **Symbol set ordering** parameter corresponds to the `SymbolMapping` on page 3-0 property.
- The `SymbolRate` on page 3-0 property takes the place of the block sample rate capability.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPFSKModulator` | `comm.FSKDemodulator`

Introduced in R2012a

reset

System object: `comm.FSKModulator`
Package: `comm`

Reset states of M-FSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `FSKModulator` object, `H`.

step

System object: `comm.FSKModulator`
Package: `comm`

Modulate using M-ary FSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the FSK modulator System object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit- valued column vector with numeric or logical data types. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol you specify in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GeneralQAMDemodulator

Package: `comm`

Demodulate using arbitrary QAM constellation

Description

The `GeneralQAMDemodulator` object demodulates a signal that was modulated using quadrature amplitude modulation. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using quadrature amplitude modulation:

1. Define and set up your QAM demodulator object. See “Construction” on page 3-664.
2. Call `step` to demodulate a signal according to the properties of `comm.GeneralQAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.GeneralQAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using a general quadrature amplitude modulation (QAM) method.

`H = comm.GeneralQAMDemodulator(Name,Value)` creates a general QAM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.GeneralQAMDemodulator(CONST,Name,Value)` creates a general QAM demodulator object, `H`. This object has the `Constellation` property set to `CONST`, and the other specified properties set to the specified values.

Properties

`Constellation`

Signal constellation

Specify the constellation points as a real or complex, double-precision data type vector. The default is $\mathbf{exp}(2 \times \pi \times 1i \times {}^{(0:7)}\!/_{8})$. The length of the vector determines the modulation order.

When you set the `BitOutput` on page 3-0 property to `false`, the `step` method outputs a vector with integer values. These integers are between `0` and M–1, where M is the length of this property vector. The length of the output vector equals the length of the input signal.

When you set the `BitOutput` property to `true`, the output signal contains bits. For bit outputs, the size of the signal constellation requires an integer power of two and the output length is an integer multiple of the number of bits per symbol.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`.

When you set this property to `true` the `step` method outputs a column vector of bit values with length equal to **log2**(*M*) times the number of demodulated symbols, where *M* is the length of the signal constellation specified in the `Constellation` on page 3-0 property. The length *M* determines the modulation order.

When you set this property to `false`, the `step` method outputs a column vector, of length equal to the input data vector. The vector contains integer symbol values between `0` and *M*–1.

`DecisionMethod`

Demodulation decision method

Specify the decision method the object uses as one of `Hard decision` | `Log-likelihood ratio` | `Approximate log-likelihood ratio`. The default is `Hard decision`. When you set the `BitOutput` on page 3-0 property to false the object always performs hard decision demodulation. This property applies when you set the `BitOutput` property to true.

`VarianceSource`

Source of noise variance

Specify the source of the noise variance as one of `Property` | `Input port`. The default is `Property`. This property applies when you set the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

`Variance`

Noise variance

Specify the variance of the noise as a nonzero, real scalar value. The default is `1`. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or -Inf. This result occurs because the LLR algorithm would compute the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, using approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the `VarianceSource` on page 3-0 property to `Property`. This property is tunable.

`OutputDataType`

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision` .

This property applies only when you set the `BitOutput` on page 3-0 property to `false` or when you set the `BitOutput` property to `true` and the `DecisionMethod` on page 3-0 property to `Hard` decision or Approximate log-likelihood ratio. In this case, when you set the OutputDataType on page 3-0 property to Full precision, the output data type is the same as that of the input when the input data has a single or double-precision data type.

When the input data is of a fixed-point type, the output data type works as if you had set the OutputDataType property to Smallest unsigned integer.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer or Full precision mode.

When you set the BitOutput property to true, and the DecisionMethod property to Hard Decision the data type logical becomes a valid option.

When you set the BitOutput property to true and the DecisionMethod property to Approximate log-likelihood ratio you may only set this property to Full precision | Custom.

If you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio, the output data has the same type as that of the input. In this case, that value can be only single or double precision.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects” on page 3-670.

RoundingMethod

Rounding of fixed-point numeric values

Specify the rounding method as one of Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero. The default is Floor. This property applies when the object is not in a full precision configuration. This property does not apply when you set BitOutput on page 3-0 to true and DecisionMethod on page 3-0 to Log-likelihood ratio.

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of Wrap | Saturate. The default is Wrap. This property applies when the object is not in a full precision configuration. This property does not apply when you set the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Log-likelihood ratio.

ConstellationDataType

Data type of signal constellation

Specify the constellation fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property does not apply when you set the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio`.

`CustomConstellationDataType`

Fixed-point data type of signal constellation

Specify the constellation fixed-point type as an unscaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `ConstellationDataType` on page 3-0 property to `Custom`.

`Accumulator1DataType`

Data type of accumulator 1

Specify the accumulator 1 fixed-point data type as one of `Full precision` | `Custom`. The default is `Full precision`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false. This property does not apply when you set the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio`.

`CustomAccumulator1DataType`

Fixed-point data type of accumulator 1

Specify the accumulator 1 fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `Accumulator1DataType` on page 3-0 property to `Custom`.

`ProductInputDataType`

Data type of product

Specify the product input fixed-point data type as one of `Same as accumulator 1` | `Custom`. The default is `Same as accumulator 1`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false, the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio`.

`CustomProductInputDataType`

Fixed-point data type of product

Specify the product input fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `ProductInputDataType` on page 3-0 property to `Custom`.

`ProductOutputDataType`

Data type of product output

Specify the product output fixed-point data type as one of `Full precision` | `Custom`. The default is `Full precision` . This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false, the `BitOutput` on page 3-0 property to true and the `DecisionMethod` on page 3-0 property to `Log-likelihood ratio`.

CustomProductOutputDataType

Fixed-point data type of product output

Specify the product output fixed-point type as a scaled numerictype object with a Signedness of Auto. The default is numerictype([],32,30). This property applies when you set the FullPrecisionOverride on page 3-0 property to false and the ProductOutputDataType on page 3-0 property to Custom.

Accumulator2DataType

Data type of accumulator 2

Specify the accumulator 2 fixed-point data type as one of Full precision | Custom. The default is Full precision . This property applies when you set the FullPrecisionOverride on page 3-0 property to false, the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Log-likelihood ratio.

CustomAccumulator2DataType

Fixed-point data type accumulator 2

Specify the accumulator 2 fixed-point data type as a scaled numerictype object with a Signedness of Auto. The default is numerictype([],32,30). This property applies when you set the FullPrecisionOverride on page 3-0 property to false and the Accumulator2DataType on page 3-0 property to Custom.

Accumulator3DataType

Data type of accumulator 3

Specify the accumulator 3 fixed-point data type as one of Full precision | Custom. The default is Full precision . This property applies when you set the FullPrecisionOverride on page 3-0 property to false, the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Approximate log-likelihood ratio.

CustomAccumulator3DataType

Fixed-point data type of accumulator 3

Specify the accumulator 3 fixed-point type as a scaled numerictype object with a Signedness of Auto. The default is numerictype([],32,30). This property applies when you set the FullPrecisionOverride on page 3-0 property to false and the Accumulator3DataType on page 3-0 property to Custom.

NoiseScalingInputDataType

Data type of noise-scaling input

Specify the noise-scaling input fixed-point data type as one of Same as accumulator 3 | Custom. The default is Same as accumulator 3. This property applies when you set the FullPrecisionOverride on page 3-0 property to false, the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Approximate log-likelihood ratio.

CustomNoiseScalingInputDataType

Fixed-point data type of noise-scaling input

Specify the noise-scaling input fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `NoiseScalingInputDataType` on page 3-0 property to `Custom`.

InverseVarianceDataType

Data type of inverse noise variance

Specify the inverse noise variance fixed-point data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`. This property applies when you set the `BitOutput` on page 3-0 property to true, the `DecisionMethod` on page 3-0 property to `Approximate log-likelihood ratio`, and the `VarianceSource` on page 3-0 property to `Property`.

CustomInverseVarianceDataType

Fixed-point data type of inverse noise variance

Specify the inverse noise variance fixed-point type as a `numerictype` object with a Signedness of Auto. The default is `numerictype([],16,8)`. This property applies when you set the `InverseVarianceDataType` on page 3-0 property to `Custom`.

CustomOutputDataType

Data type of output

Specify the output fixed-point type as a scaled `numerictype` object with a Signedness of Auto. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `OutputDataType` on page 3-0 property to `Custom`.

Methods

step	Demodulate using arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate and demodulate data using an arbitrary three-point constellation.

```
% Setup a three point constellation
c = [1 1i -1];
hQAMMod = comm.GeneralQAMModulator(c);
hAWGN = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',15,'SignalPower',0.89);
hQAMDemod = comm.GeneralQAMDemodulator(c);
```

```
%Create an error rate calculator
hError = comm.ErrorRate;
for counter = 1:100
    % Transmit a 50-symbol frame
    data = randi([0 2],50,1);
    modSignal = step(hQAMMod, data);
    noisySignal = step(hAWGN, modSignal);
    receivedData = step(hQAMDemod, noisySignal);
    errorStats = step(hError, data, receivedData);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
     errorStats(1), errorStats(2))
```

More About

Full Precision for Fixed-Point System Objects

`FullPrecisionOverride` is a convenience property that, when you set to `true`, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input.

For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Full precision for System objects does not optimize coefficient values. When you set the `FullPrecisionOverride` property to `true`, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set `FullPrecisionOverride` to `false`.

Algorithms

This object implements the algorithm, inputs, and outputs described on the General QAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`genqamdemod` | `qamdemod` | `qammod`

Objects

`comm.GeneralQAMModulator` | `comm.RectangularQAMDemodulator`

Introduced in R2012a

step

System object: `comm.GeneralQAMDemodulator`
Package: `comm`

Demodulate using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates the input data, `X`, with the general QAM demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `BitOutput` property to true and the `DecisionMethod` property to 'Log-likelihood ratio' the input data type must be single or double precision. Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the `VarianceSource` property to 'Input port'.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GeneralQAMModulator

Package: `comm`

Modulate using arbitrary QAM constellation

Description

The `GeneralQAMModulator` object modulates using quadrature amplitude modulation. The output is a baseband representation of the modulated signal.

To modulate a signal using quadrature amplitude modulation:

1. Define and set up your QAM modulator object. See "Construction" on page 3-673.
2. Call `step` to modulate a signal according to the properties of `comm.GeneralQAMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.GeneralQAMModulator` creates a modulator System object, `H`. This object modulates the input signal using a general quadrature amplitude modulation (QAM) method.

`H = comm.GeneralQAMModulator(Name,Value)` creates a QAM modulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.GeneralQAMModulator(CONST,Name,Value)` creates a General QAM modulator object, `H`. This object has the `Constellation` property set to `CONST`, and the other specified properties set to the specified values.

Properties

`Constellation`

Signal constellation

Specify the constellation points as a vector of real or complex double-precision data type. The default is $\mathbf{exp}(2 \times \pi \times 1i \times {}^{(0:7)}\!/\!_{8})$. The length of the vector determines the modulation order. The `step` method inputs requires integers between 0 and M–1, where M indicates the length of this property vector. The object maps an input integer m to the $(m+1)^{st}$ value in the Constellation vector.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `single` | `Custom`. The default is `double`.

Fixed-Point Properties

`CustomOutputDataType`

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` on page 3-0 property to `Custom`.

Methods

step	Modulate using arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate data using an arbitrary 3-point constellation. Then, visualize the data in a scatter plot

```
     hQAMMod = comm.GeneralQAMModulator;
% Setup a three point constellation
     hQAMMod.Constellation = [1 1i -1];
     data = randi([0 2],100,1);
     modData = step(hQAMMod, data);
     scatterplot(modData)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General QAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`genqamdemod` | `qamdemod` | `qammod`

Objects

`comm.GeneralQAMDemodulator`

Introduced in R2012a

step

System object: comm.GeneralQAMModulator
Package: comm

Modulate using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the general QAM modulator System object, `H`. It returns the baseband modulated output, `Y`. The input must be an integer scalar or an integer-valued column vector. The data type of the input can be numeric or unsigned fixed point of word length **ceil**(**log2**(`ModulationOrder`)) (`fi` object).

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GeneralQAMTCMDemodulator

Package: comm

Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Description

The GeneralQAMTCMDemodulator object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using an arbitrary signal constellation.

To demodulate a signal that was modulated using a trellis-coded, general quadrature amplitude modulator:

1. Define and set up your general QAM TCM modulator object. See “Construction” on page 3-677.
2. Call step to demodulate a signal according to the properties of comm.GeneralQAMTCMDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.GeneralQAMTCMDemodulator creates a trellis-coded, general quadrature amplitude (QAM TCM) demodulator System object, H. This object demodulates convolutionally encoded data that has been mapped to an arbitrary QAM constellation.

H = comm.GeneralQAMTCMDemodulator(Name,Value) creates a general QAM TCM demodulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.GeneralQAMTCMDemodulator(TRELLIS,Name,Value) creates a general QAM TCM demodulator object, H. This object has the TrellisStructure property set to TRELLIS, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check if a structure is a valid trellis structure. The default is the value that results from poly2trellis([1 3], [1 0 0; 0 5 2]).

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set this property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

`TracebackDepth`

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The Traceback depth parameter influences the decoding accuracy and delay. The decoding delay indicates the number of zero symbols that precede the first decoded symbol in the output.

When you set the `TerminationMethod` on page 3-0 property to `Continuous`, the decoding delay consists of `TracebackDepth` zero symbols or `TracebackDepth`×*K* zero bits for a rate *K*/*N* convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

`ResetInputPort`

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

`Constellation`

Signal constellation

Specify a double- or single-precision complex vector. This vector lists the points in the signal constellation that were used to map the convolutionally encoded data. The constellation must be specified in set-partitioned order. See documentation for the General TCM Encoder block for more information on set-partitioned order. The length of the constellation vector must equal the number of possible input symbols to the convolutional decoder of the general QAM TCM demodulator object. This corresponds to 2^N for a rate *K*/*N* convolutional code. The default corresponds to a set-partitioned order for the points of an 8-PSK signal constellation. This value is expressed as $\mathbf{exp}(2 \times \pi \times j \times [0\ 4\ 2\ 6\ 1\ 5\ 3\ 7]/8)$.

`OutputDataType`

Data type of output

Specify output data type as one of `logical` | `double`. The default is `double`.

Methods

reset	Reset states of the general QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate and Demodulate Data Using QAM TCM

Modulate and demodulate noisy data using QAM TCM modulation with an arbitrary 4-point constellation. Estimate the resultant BER.

Define a trellis structure with two input symbols and four output symbols using a [171 133] generator polynomial. Define an arbitrary four-point constellation.

```
qamTrellis = poly2trellis(7,[171 133]);
refConst = exp(pi*1i*[1 2 3 6]/4);
```

Create a QAM TCM modulator and demodulator System object™ pair using `qamTrellis` and `refConst`.

```
hMod = comm.GeneralQAMTCMModulator(qamTrellis,'Constellation', refConst);
hDemod = comm.GeneralQAMTCMDemodulator(qamTrellis,'Constellation',refConst);
```

Create an AWGN channel object in which the noise is set by using a signal-to-noise ratio.

```
hAWGN = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)', ...
    'SNR',4);
```

Create an error rate calculator with delay (in bits) equal to the product of `TracebackDepth` and the number of bits per symbol

```
hError = comm.ErrorRate(...
    'ReceiveDelay', hDemod.TracebackDepth*log2(qamTrellis.numInputSymbols));
```

Generate random binary data and apply QAM TCM modulation. Pass the signal through an AWGN channel and demodulate. Collect the error statistics.

```
for counter = 1:10
    % Generate binary data
    data = randi([0 1],500,1);
    % Modulate
    modSignal = step(hMod,data);
    % Pass through an AWGN channel
    noisySignal = step(hAWGN,modSignal);
    % Demodulate
    receivedData = step(hDemod,noisySignal);
    % Calculate the error statistics
    errorStats = step(hError,data,receivedData);
end
```

Display the BER and the number of bit errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 1.16e-02
Number of errors = 58
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General TCM Decoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMDemodulator` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: `comm.GeneralQAMTCMDemodulator`
Package: `comm`

Reset states of the general QAM TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GeneralQAMTCMDemodulator` object, `H`.

step

System object: `comm.GeneralQAMTCMDemodulator`
Package: `comm`

Demodulate convolutionally encoded data mapped to arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates the general QAM modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated convolutionally encoded bits. `X` must be a complex double or single precision column vector. The `step` method outputs a demodulated binary column data vector, `Y`. When the convolutional encoder represents a rate *K*/*N* code, the length of the output vector equals *K*×*L*, where *L* is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder states of the general QAM TCM demodulator System object to the all-zeros state when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar integer. This syntax applies when you set the `ResetInputPort` property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GeneralQAMTCMModulator

Package: `comm`

Convolutionally encode binary data and map using arbitrary QAM constellation

Description

The `GeneralQAMTCMModulator` object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal. The object then maps the result to an arbitrary signal constellation. The Signal constellation property lists the signal constellation points in set-partitioned order.

To modulate a signal using a trellis-coded, general quadrature amplitude modulator:

1. Define and set up your general QAM TCM modulator object. See “Construction” on page 3-683.
2. Call `step` to modulate a signal according to the properties of `comm.GeneralQAMTCMModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.GeneralQAMTCMModulator` creates a trellis-coded, general quadrature amplitude (QAM TCM) modulator System object, `H`. This object convolutionally encodes a binary input signal and maps the result using QAM modulation with a signal constellation specified in the Constellation property.

`H = comm.GeneralQAMTCMModulator(Name,Value)` creates a general QAM TCM modulator System object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.GeneralQAMTCMModulator(TRELLIS,Name,Value)` creates a general QAM TCM modulator System object, `H`. This object has the `TrellisStructure` property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

`TrellisStructure`

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check if a structure is a valid trellis structure. The default is the result of `poly2trellis([1 3], [1 0 0; 0 5 2])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate K/N code, the `step` method outputs the vector with length $y = N \times {}^{(L + S)}\!/_{K}$, where $S = \mathit{constraintLength}–1$. In the case of multiple constraint lengths, $S = \mathbf{sum}(\mathit{constraintLength}(i)–1))$. L represents the length of the input to the `step` method.

ResetInputPort

Enable modulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to their initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

Constellation

Signal constellation

Specify a double- or single-precision complex vector that lists the points in the signal constellation that were used to map the convolutionally encoded data. You must specify the constellation in set-partitioned order. See documentation for the General TCM Encoder block for more information on set-partitioned order. The length of the constellation vector must equal the number of possible input symbols to the convolutional decoder of the general QAM TCM demodulator object. This corresponds to 2^N for a rate K/N convolutional code. The default corresponds to a set-partitioned order for the points of an 8-PSK signal constellation. This value is expressed $\mathbf{exp}(2 \times \pi \times j \times {}^{[0\ 4\ 2\ 6\ 1\ 5\ 3\ 7]}\!/_{8})$.

OutputDataType

Data type of output

Specify the output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of the general QAM TCM modulator object
step	Convolutionally encode binary data and map using arbitrary QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data using QAM TCM with an Arbitrary Constellation

Modulate data using QAM TCM modulation with an arbitrary 4-point constellation. Display a scatter plot of the modulated data.

Create binary data.

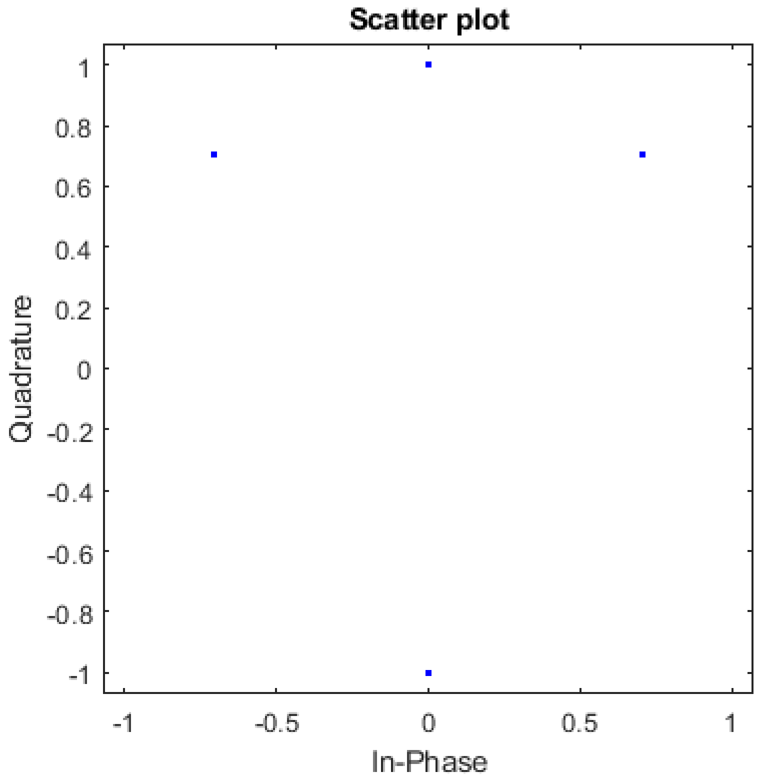
```
data = randi([0 1],1000,1);
```

Use the trellis structure with generating polynomial [171 133] and 4-point arbitrary constellation $\{ e^{j\pi/4}, e^{j\pi/2}, e^{j3\pi/4}, e^{j3\pi/2} \}$ to perform QAM TCM modulation.

```
t = poly2trellis(7,[171 133]);
hMod = comm.GeneralQAMTCMModulator(t,...
    'Constellation',exp(pi*1i*[1 2 3 6]/4));
```

Modulate and plot the data.

```
modData = step(hMod,data);
scatterplot(modData);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the General TCM Encoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMModulator` | `comm.GeneralQAMTCMDemodulator` | `comm.PSKTCMModulator`

Introduced in R2012a

reset

System object: `comm.GeneralQAMTCMModulator`
Package: `comm`

Reset states of the general QAM TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GeneralQAMTCMModulator` object, `H`.

step

System object: comm.GeneralQAMTCMModulator
Package: comm

Convolutionally encode binary data and map using arbitrary QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) convolutionally encodes and modulates the input data, X, and returns the encoded and modulated data, Y. X must be of data type numeric, logical, or unsigned fixed point of word length 1 (fi object). When the convolutional encoder represents a rate *K*/*N* code, the length of the input vector, X, must be *K*×*L*, for some positive integer *L*. The step method outputs a complex column vector, Y, of length *L*.

Y = step(H,X,R) resets the encoder of the general QAM TCM modulator object to the all-zeros state when you input a non-zero reset signal, R. R must be a double precision or logical scalar integer. This syntax applies when you set the ResetInputPort property to true.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.GMSKDemodulator

Package: comm

Demodulate using GMSK method and the Viterbi algorithm

Description

The GMSKDemodulator object uses a Viterbi algorithm to demodulate a signal that was modulated using the Gaussian minimum shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using Gaussian minimum shift keying:

1. Define and set up your GMSK demodulator object. See “Construction” on page 3-651.
2. Call step to demodulate a signal according to the properties of GMSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.GMSKDemodulator creates a demodulator System object, H. This object demodulates the input Gaussian minimum shift keying (GMSK) modulated data using the Viterbi algorithm.

H = comm.GMSKDemodulator(Name,Value) creates a GMSK demodulator object, H. This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

BitOutput

Output data as bits

Specify whether the output is groups of bits or integer values. The default is false.

When you set the BitOutput on page 3-0 property to false, the step method outputs a column vector of length equal to *N*/SamplesPerSymbol on page 3-0 . *N* is the length of the input signal, which is the number of input baseband modulated symbols. The elements of the output vector are –1 or 1.

When you set the BitOutput property to true, the step method outputs a binary column vector of length equal to *N*/SamplesPerSymbol with bit values of 0 or 1.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of bandwidth and symbol time for the Gaussian pulse shape as a real, positive scalar. The default 0.3.

PulseLength

Pulse length

Specify the length of the Gaussian pulse shape in symbol intervals as a real positive integer. The default 4.

SymbolPrehistory

Symbol prehistory

Specify the data symbols used by the modulator prior to the first call to the step method. The default is 1. This property requires a scalar or vector with elements equal to -1 or 1. If the value is a vector, its length must be one less than the value you set in the PulseLength on page 3-0 property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar value. The default is 0.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar value. The default is 8.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar value. The value of this property is also the output delay, and the number of zero symbols that precede the first meaningful demodulated symbol in the output. The default is 16.

OutputDataType

Data type of output

Specify the output data type as one of int8 | int16 | int32 | double, when you set the BitOutput on page 3-0 property to false.

When you set the BitOutput property to true, specify the output data type as one of logical | double. The default is double.

Methods

reset	Reset states of the GMSK demodulator object
step	Demodulate using GMSK method and the Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate a GMSK signal with bit inputs and phase offset

```
% Create a GMSK modulator, an AWGN channel, and a GMSK demodulator. Use a phase offset of pi/4.
    hMod = comm.GMSKModulator('BitInput', true, 'InitialPhaseOffset', pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.GMSKDemodulator('BitOutput', true, ...
                    'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the Viterbi algorithm
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.000133
Number of errors = 4
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex
```

```
   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

Algorithms

This object implements the algorithm, inputs, and outputs described on the GMSK Demodulator Baseband block reference page. The object properties correspond to the block parameters. For GMSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.GMSKModulator`

Introduced in R2012a

reset

System object: `comm.GMSKDemodulator`
Package: `comm`

Reset states of the GMSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GMSKDemodulator` object, `H`.

step

System object: `comm.GMSKDemodulator`
Package: `comm`

Demodulate using GMSK method and the Viterbi algorithm

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the GMSK demodulator object, `H`, and returns `Y`. `X` must be a double or single precision column vector with a length equal to an integer multiple of the number of samples per symbol you specify in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GMSKModulator

Package: comm

Modulate using GMSK method

Description

The GMSKModulator object modulates using the Gaussian minimum shift keying method. The output is a baseband representation of the modulated signal.

To modulate a signal using Gaussian minimum shift keying:

1 Define and set up your GMSK modulator object. See “Construction” on page 3-696.
2 Call step to modulate a signal according to the properties of comm.GMSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.GMSKModulator creates a modulator System object, H. This object modulates the input signal using the Gaussian minimum shift keying (GMSK) modulation method.

H = comm.GMSKModulator(Name,Value) creates a GMSK modulator object, H. This object has each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

BitInput

Assume input is bits

Specify whether the input is bits or integers. The default is false.

When you set the BitInput on page 3-0 property to false, the step method input requires a double-precision or signed integer data type column vector with values of -1 or 1.

When you set the BitInput property to true, step method input requires a double-precision or logical data type column vector of 0s and 1s.

BandwidthTimeProduct

Product of bandwidth and symbol time of Gaussian pulse

Specify the product of the bandwidth and symbol time for the Gaussian pulse shape as a real, positive scalar value. The default is 0.3.

PulseLength

Pulse length

Specify the length of the Gaussian pulse shape in symbol intervals as a real, positive integer. The default is `4`.

SymbolPrehistory

Symbol prehistory

Specify the data symbols the modulator uses prior to the first call to the `step` method in reverse chronological order. The default is `1`. This property requires a scalar or vector with elements equal to `-1` or `1`. If the value is a vector, then its length must be one less than the value in the `PulseLength` on page 3-0 property.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar value. The default is `0`.

SamplesPerSymbol

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar value. The default is `8`. The upsampling factor is the number of output samples that the `step` method produces for each input sample.

OutputDataType

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of the GMSK modulator object
step	Modulate using GMSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate a GMSK signal with bit inputs and phase offset

```
% Create a GMSK modulator, an AWGN channel, and a GMSK demodulator. Use a phase offset of pi/4.
    hMod = comm.GMSKModulator('BitInput', true, 'InitialPhaseOffset', pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.GMSKDemodulator('BitOutput', true, ...
                    'InitialPhaseOffset', pi/4);
% Create an error rate calculator, account for the delay caused by the Viterbi algorithm
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.000133
Number of errors = 4
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

GMSK vs. MSK

Compare Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes by plotting the eye diagram for GMSK with different pulse lengths and for MSK.

Set the samples per symbol variable.

```
sps = 8;
```

Generate random binary data.

```
data = randi([0 1],1000,1);
```

Create GMSK and MSK modulators that accept binary inputs. Set the `PulseLength` property of the GMSK modulator to `1`.

```
gmskMod = comm.GMSKModulator('BitInput',true,'PulseLength',1, ...
    'SamplesPerSymbol',sps);
mskMod = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',sps);
```

Modulate the data using the GMSK and MSK modulators.

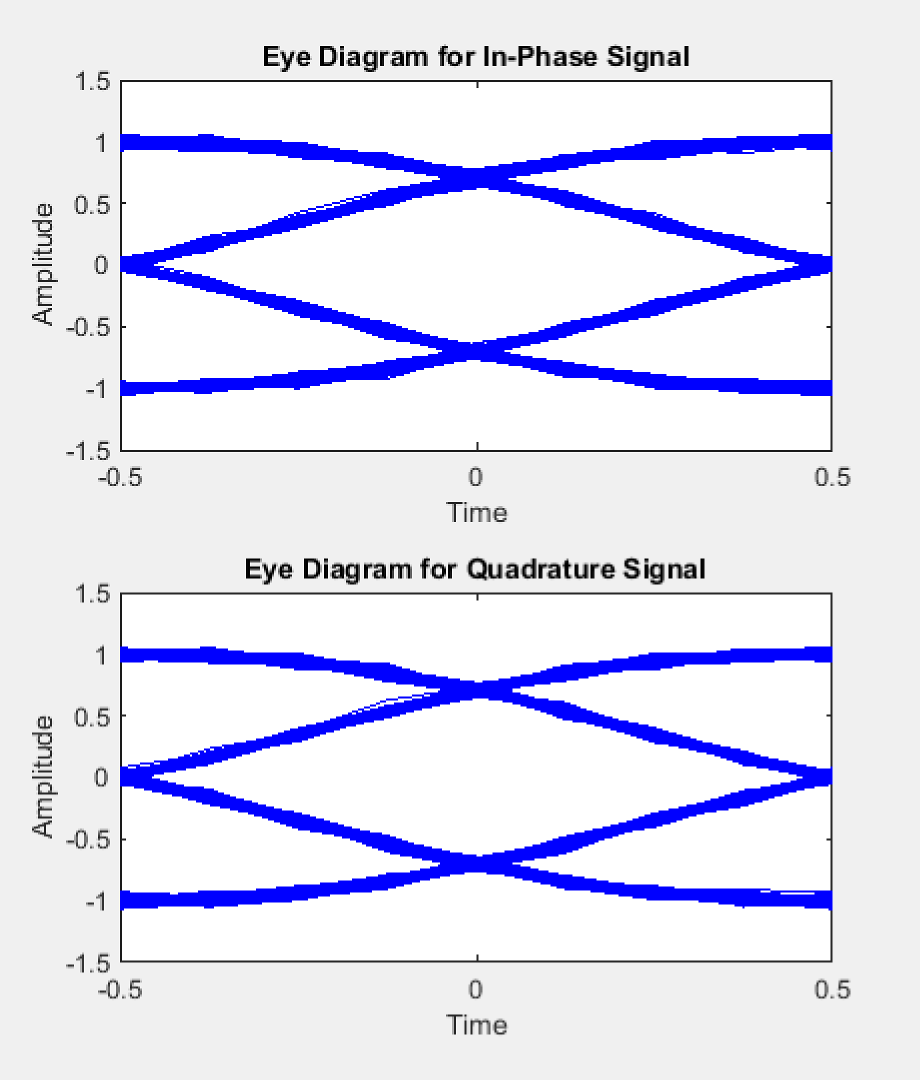
```
modSigGMSK = gmskMod(data);
modSigMSK = mskMod(data);
```

Pass the modulated signals through an AWGN channel having an SNR of `30` dB.

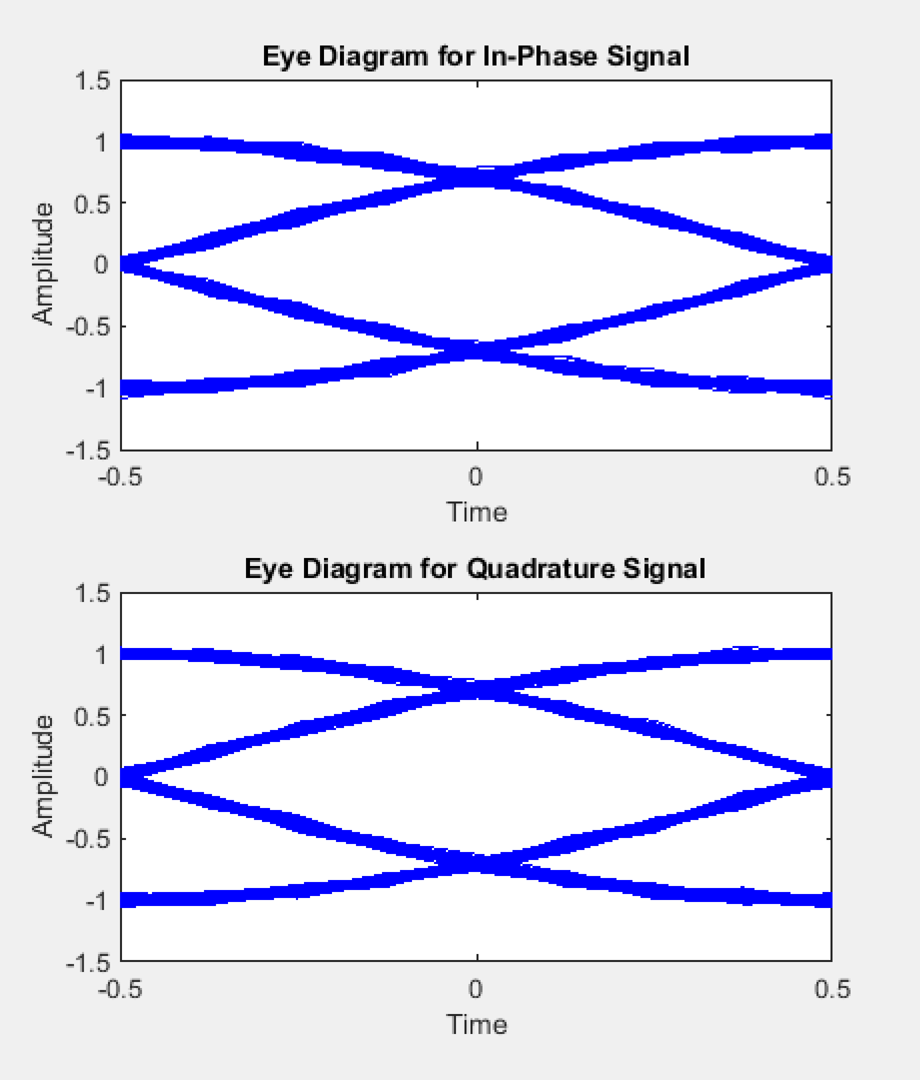
```
rxSigGMSK = awgn(modSigGMSK,30);
rxSigMSK = awgn(modSigMSK,30);
```

Use the `eyediagram` function to plot the eye diagrams of the noisy signals. With the GMSK pulse length set to `1`, the eye diagrams are nearly identical.

```
eyediagram(rxSigGMSK,sps,1,sps/2)
```



```
eyediagram(rxSigMSK,sps,1,sps/2)
```



Set the `PulseLength` property for the GMSK modulator object to `3`. Because the property is nontunable, the object must be released first.

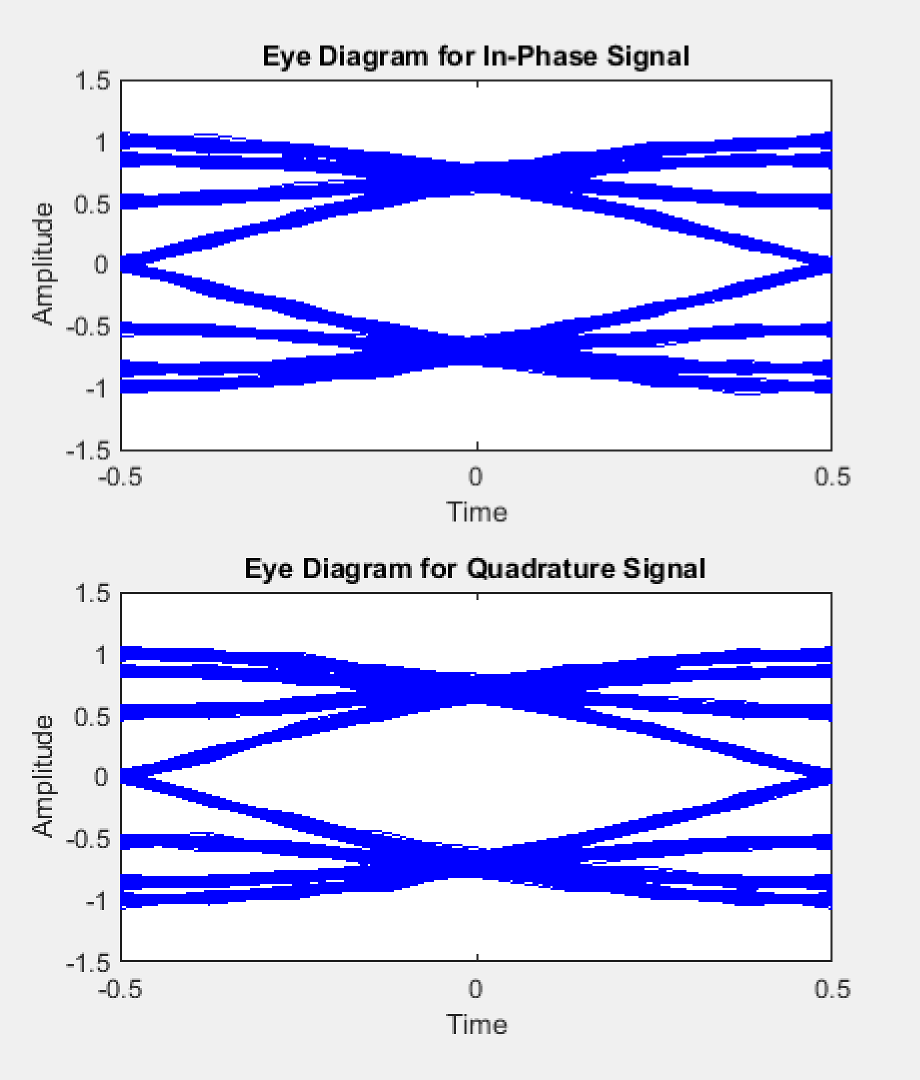
```
release(gmskMod)
gmskMod.PulseLength = 3;
```

Generate a modulated signal using the updated GMSK modulator object and pass it through the AWGN channel.

```
modSigGMSK = gmskMod(data);
rxSigGMSK = awgn(modSigGMSK,30);
```

With continuous phase modulation (CPM) waveforms, such as GSMK, the waveform depends on values of the previous symbols as well as the present symbol. Plot the eye diagram of the GMSK signal to see that the increased pulse length results in an increase in the number of paths in the eye diagram.

```
eyediagram(rxSigGMSK,sps,1,sps/2)
```



Experiment by changing the `PulseLength` parameter of the GMSK modulator object to other values. If you set the property to an even number, you should set `gmskMod.InitialPhaseOffset` to `pi/4` and update the offset argument of the `eyediagram` function from `sps/2` to `0` for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use scopes that display the phase of the signal, as described in the “CPM Phase Tree” example.

Algorithms

This object implements the algorithm, inputs, and outputs described on the GMSK Modulator Baseband block reference page. The object properties correspond to the block parameters. For GMSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.GMSKDemodulator`

Introduced in R2012a

reset

System object: `comm.GMSKModulator`
Package: `comm`

Reset states of the GMSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GMSKModulator` object, `H`.

step

System object: `comm.GMSKModulator`
Package: `comm`

Modulate using GMSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the GMSK modulator object, `H`. It returns the baseband modulated output in `Y`. Depending on the `BitInput` property value, input `X` can be a double precision, signed integer, or logical column vector. The length of vector `Y` is equal to the number of input samples times the number of samples per symbol that you specify in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GMSKTimingSynchronizer

Package: `comm`

Recover symbol timing phase using fourth-order nonlinearity method

Description

The `GMSKTimingSynchronizer` object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general non-data-aided feedback method. This timing synchronization is a non-data-aided feedback method that is independent of carrier phase recovery, but requires prior compensation for the carrier frequency offset. You can use this block for systems that use Gaussian minimum shift keying (GMSK) modulation.

To recover the symbol timing phase of the input signal:

1. Define and set up your GMSK timing synchronizer object. See “Construction” on page 3-706.
2. Call `step` to recover the symbol timing phase of the input signal according to the properties of `comm.GMSKTimingSynchronizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.GMSKTimingSynchronizer` creates a timing phase synchronizer System object, `H`. This object recovers the symbol timing phase of the GMSK input signal using a fourth-order nonlinearity method.

`H = comm.GMSKTimingSynchronizer(Name,Value)` creates a GMSK timing synchronizer object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`SamplesPerSymbol`

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar value greater than `1`. The default is `4`.

`ErrorUpdateGain`

Error update step size

Specify the step size for updating successive timing phase estimates as a positive real scalar value. Typically, this number is less than 1/`SamplesPerSymbol` on page 3-0 , which corresponds to a slowly varying timing phase. The default is `0.05`. This property is tunable.

`ResetInputPort`

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`.

When you set this property to `true`, you must specify a reset input value to the `step` method.

When you specify a nonzero value as the reset input, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

`ResetCondition`

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never` | `Every frame`. The default is `Never`.

When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next.

When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. In this case, the restart occurs at each `step` method call. This property applies when you set the `ResetInputPort` on page 3-0 property to `false`.

Methods

reset	Reset states of GMSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover Timing Phase of GMSK Signal

Create GMSK modulator, variable fractional delay, and GMSK timing synchronizer System objects.

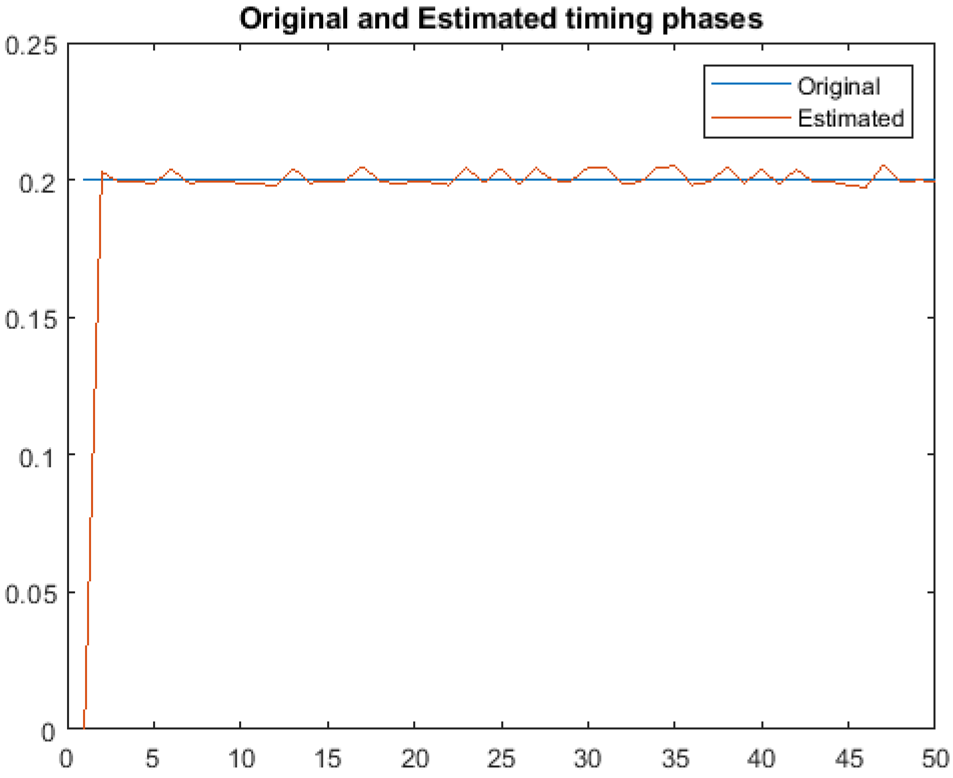
```
gmskMod = comm.GMSKModulator('BitInput', true, ...
    'SamplesPerSymbol', 14);
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
gmskTimingSync = comm.GMSKTimingSynchronizer('SamplesPerSymbol', 14, ...
    'ErrorUpdateGain', 0.05);
```

Main processing loop:

```
phEst = zeros(50,1);
for i = 1:50
    data = randi([0 1],100,1); % Generate data
    modData = gmskMod(data);   % Modulate data

    % Apply timing offset error
    impairedData = varDelay(modData,timingOffset*14);
    % Perform timing phase recovery
    [~,phase] = gmskTimingSync(impairedData);
    phEst(i) = phase(1)/14;
end
```

Plot the results.

```
plot(1:50,[0.2*ones(50,1) phEst])
legend( 'Original','Estimated')
title('Original and Estimated timing phases')
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK-Type Signal Timing Recovery block reference page. The object properties correspond to the block parameters, except:

- The object corresponds to the MSK-Type Signal Timing Recovery block with the **Modulation type** parameter set to GMSK.

- The **Reset** parameter corresponds to the `ResetInputPort` on page 3-0 and `ResetCondition` on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: `comm.GMSKTimingSynchronizer`
Package: `comm`

Reset states of GMSK timing phase synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states for the `GMSKTimingSynchronizer` object `H`.

step

System object: `comm.GMSKTimingSynchronizer`
Package: `comm`

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,PHASE] = step(H,X)` performs timing phase recovery and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. `X` must be a double or single precision complex column vector.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies when you set the `ResetInputPort` property to true.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.GoldSequence

Package: `comm`

Generate Gold sequence

Description

The `GoldSequence` object generates a Gold sequence. Gold sequences form a large class of sequences that have good periodic cross-correlation properties.

To generate a Gold sequence:

1. Define and set up your Gold sequence object. See “Construction” on page 3-712.
2. Call `step` to generate the Gold sequence according to the properties of `comm.GoldSequence`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`H = comm.GoldSequence` creates a Gold sequence generator System object, `H`. This object generates a pseudo-random Gold sequence.

`H = comm.GoldSequence(Name,Value)` creates a Gold sequence generator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`FirstPolynomial`

Generator polynomial for first preferred PN sequence

Specify the polynomial that determines the feedback connections for the shift register of the first preferred PN sequence generator. The default is `'z^6 + z + 1'`. You can specify the polynomial as a character vector. You can also specify the generator polynomial as a numeric, binary vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal `1`, and the length of this vector requires a value of *n*+1, where *n* is the degree of the generator polynomial. Lastly, you can specify the generator polynomial as a numeric vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The degree of the first generator polynomial must equal the degree of the second generator polynomial specified in the `SecondPolynomial` on page 3-0 property.

`FirstInitialConditions`

Initial conditions for first PN sequence generator

Specify the initial conditions for the shift register of the first preferred PN sequence generator. The default is `[0 0 0 0 0 1]`. The initial conditions require a numeric, binary scalar, or a numeric, binary vector with length equal to the degree of the first generator polynomial specified in the `FirstPolynomial` on page 3-0 property. If you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. If you set this property to a scalar, the initial conditions of all shift register cells are the specified scalar value.

SecondPolynomial

Generator polynomial for second preferred PN sequence

Specify the polynomial that determines the feedback connections for the shift register of the second preferred PN sequence generator. The default is `'z^6 + z^5 + z^2 + z + 1'`. You can specify the polynomial as a character vector. You can also specify the generator polynomial as a binary, numeric vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal `1` and the length of this vector requires a value of *n*`+1`, where *n* is the degree of the generator polynomial. Lastly, you can specify the generator polynomial as a numeric vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$. The degree of the second generator polynomial must equal the degree of the first generator polynomial specified in the `FirstPolynomial` on page 3-0 property.

SecondInitialConditionsSource

Source of initial conditions for second PN sequence

Specify the source of the initial conditions that determines the start of the second PN sequence as one of `Property` | `Input port`. The default is `Property`. When you set this property to `Property`, you can specify the initial conditions as a scalar or binary vector using the `SecondInitialConditions` property. When you set this property to `Input port`, you specify the initial conditions as an input to the `step`method. The object accepts a binary scalar or a binary vector input. The length of the input must equal the degree of the generator polynomial that the `SecondPolynomial` on page 3-0 property specifies.

SecondInitialConditions

Initial conditions for second PN sequence generator

Specify the initial conditions for the shift register of the second preferred PN sequence generator as a numeric, binary scalar, or as a numeric, binary vector. The length must equal the degree of the second generator polynomial. You set the second generator polynomial in the `SecondPolynomial` on page 3-0 property.

When you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. The default is `[0 0 0 0 0 1]`.

When you set this property to a scalar, the initial conditions of all shift register cells are the specified scalar value.

Index

Index of output sequence of interest

Specify the index of the output sequence of interest from the set of available sequences as a scalar integer. The default is `0`. The scalar integer must be in the range [–2, 2^n–2], where *n* is the degree of the generator polynomials you specify in the `FirstPolynomial` on page 3-0 and `SecondPolynomial` on page 3-0 properties.

The index values `-2` and `-1` correspond to the first and second preferred PN sequences as generated by the `FirstPolynomial` and `SecondPolynomial`, respectively.

The set $G(u, v)$ of available Gold sequences is defined by $G(u,v) = \{u, v,$ $(u$ `xor` $T^{v})$, $(u$ `xor` $T^{2v})$, ..., $(u$ `xor` $T^{((N-1)v)})\}$.In this case, T represents the operator that shifts vectors cyclically to the left by one place, and u, v represent the two preferred PN sequences. Also, $G(u,v)$ contains N+2 Gold sequences of period N. You select the desired sequence from this set using the `Index` on page 3-0 property.

`Shift`

Sequence offset from initial time

Specify the offset of the Gold sequence from its starting point as a numeric, integer scalar value that can be positive or negative. The default is `0`. The Gold sequence has a period of $N = 2^{n} - 1$, where n is the degree of the generator polynomials specified in the `FirstPolynomial` on page 3-0 and `SecondPolynomial` on page 3-0 properties. The shift value is wrapped with respect to the sequence period.

`VariableSizeOutput`

Enable variable-size outputs

Set this property to true to enable an additional input to the step method. The default is false. When you set this property to true, the enabled input specifies the output size of the Gold sequence used for the step. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to false, the `SamplesPerFrame` property specifies the number of output samples.

`MaximumOutputSize`

Maximum output size

Specify the maximum output size of the Gold sequence as a positive integer 2-element row vector. The second element of the vector must be 1. The default is [10 1].

This property applies when you set the `VariableSizeOutput` property to true.

`SamplesPerFrame`

Number of output samples per frame

Specify the number of Gold sequence samples that the `step` method outputs as a numeric, integer scalar value. The default is `1`. If you set this property to a value of M, then the `step` method outputs M samples of a Gold sequence with a period of $N = 2^{n} - 1$. The value of n represents the degree of the generator polynomials that you specify in the `FirstPolynomial` on page 3-0 and `SecondPolynomial` on page 3-0 properties.

`ResetInputPort`

Enable generator reset input

Set this property to `true` to enable an additional reset input to the `step` method. The default is `false`. This input resets the states of the two shift registers of the Gold sequence generator to the initial conditions specified in the `FirstInitialConditions` on page 3-0 and `SecondInitialConditions` on page 3-0 properties.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `logical` | `Smallest unsigned integer`. The default is `double`.

You must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` mode.

Methods

reset	Reset states of Gold sequence generator object
step	Generate a Gold sequence

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate Gold Sequence Samples

Generate 10 samples of a Gold sequence having period $2^5 - 1$.

```
goldseq = comm.GoldSequence('FirstPolynomial','x^5+x^2+1',...
    'SecondPolynomial','x^5+x^4+x^3+x^2+1',...
    'FirstInitialConditions',[0 0 0 0 1],...
    'SecondInitialConditions',[0 0 0 0 1],...
    'Index',4,'SamplesPerFrame',10);
x = goldseq()
```

```
x = 10×1

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

Algorithms

This object implements the algorithm, inputs, and outputs described on the Gold Sequence Generator block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.KasamiSequence` | `comm.PNSequence`

reset

System object: `comm.GoldSequence`
Package: `comm`

Reset states of Gold sequence generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GoldSequence` object, `H`.

step

System object: comm.GoldSequence
Package: comm

Generate a Gold sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj) and y = obj() perform equivalent operations.

Y = step(H) outputs a frame of the Gold sequence in column vector Y. Specify the frame length with the SamplesPerFrame property. The object uses two PN sequence generators to generate a preferred pair of sequences with period N = 2^n-1. Then the object XORs these sequences to produce the output Gold sequence. The value in n is the degree of the generator polynomials that you specify in the FirstPolynomial and SecondPolynomial properties.

Y = step(H,RESET) uses RESET as the reset signal when you set the ResetInputPort property to true. The data type of the RESET input must be double precision or logical. RESET can be a scalar value or a column vector with length equal to the number of samples per frame specified in the SamplesPerFrame property. When the RESET input is a non-zero scalar, the object resets to the initial conditions that you specify in the FirstInitialConditions and SecondInitialConditions properties. It then generates a new output frame. A column vector RESET input allows multiple resets within an output frame. A non-zero value at the ith element of the vector causes a reset at the ith output sample time.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.gpu.AWGNChannel

Package: `comm`

Add white Gaussian noise to input signal with GPU

Description

The `GPU AWGNChannel` object adds white Gaussian noise to an input signal using a graphics processing unit (GPU).

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" (Parallel Computing Toolbox).

To add white Gaussian noise to an input signal:

1. Define and set up your additive white Gaussian noise channel object. See "Construction" on page 3-719.
2. Call `step` to add white Gaussian noise to the input signal according to the properties of `comm.gpu.AWGNChannel`. The behavior of `step` is specific to each object in the toolbox.

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.AWGNChannel` creates a GPU-based additive white Gaussian noise (AWGN) channel System object, `H`. This object adds white Gaussian noise to a real or complex input signal.

`H = comm.gpu.AWGNChannel(Name,Value)` creates a GPU-based AWGN channel object, `H`, with the specified property name set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

Properties

`NoiseMethod`

Method to specify noise level

Select the method to specify the noise level as one of `Signal to noise ratio (Eb/No)` | `Signal to noise ratio (Es/No)` | `Signal to noise ratio (SNR)` | `Variance`. The default is `Signal to noise ratio (Eb/No)`.

`EbNo`

Energy per bit to noise power spectral density ratio (Eb/No)

Specify the Eb/No ratio in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (Eb/No)`. The default is `10`. This property is tunable.

`EsNo`

Energy per symbol to noise power spectral density ratio (Es/No)

Specify the Es/No ratio in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (Es/No)`. The default is `10`. This property is tunable.

`SNR`

Signal to noise ratio (SNR)

Specify the SNR value in decibels. Set this property to a numeric, real scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (SNR)`. The default is `10`. This property is tunable.

`BitsPerSymbol`

Number of bits in one symbol

Specify the number of bits in each input symbol. You can set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (Eb/No)`. The default is `1` bit.

SignalPower

Input signal power in Watts

Specify the mean square power of the input signal in Watts. Set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)` or `Signal to noise ratio (SNR)`. The default is `1` Watt. The object assumes a nominal impedance of `1` Ohm. This property is tunable.

SamplesPerSymbol

Number of samples per symbol

Specify the number of samples per symbol. Set this property to a numeric, positive, integer scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Signal to noise ratio (Eb/No)` or `Signal to noise ratio (Es/No)`. The default is 1 sample.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property` | `Input port`. The default is `Property`. Set `VarianceSource` to `Input port` to specify the noise variance value via an input to the step method. Set `VarianceSource` to `Property` to specify the noise variance value using the Variance property. This property applies when you set the NoiseMethod property to `Variance`.

Variance

Noise variance

Specify the variance of the white Gaussian noise. You can set this property to a numeric, positive, real scalar or row vector with a length equal to the number of channels. This property applies when you set the NoiseMethod property to `Variance` and the VarianceSource property to `Property`. The default is `1`. This property is tunable.

RandomStream

Source of random number stream

Specify the source of random number stream. The only valid setting for this property is `Global stream`. The object generates the normally distributed random numbers from the current global random number stream.

Seed

Initial seed of `mt19937ar` random number stream

The GPU version of the AWGN Channel System object does not use this property.

Methods

step	Add white Gaussian noise to input signal

Common to All System Objects	
release	Allow System object property value changes

Algorithm

This object uses the same algorithm as the comm.AWGNChannel System object. See the Algorithms section of the comm.AWGNChannel help page for more details. The object properties correspond to the related block parameters, except that:

- This object uses parallel.gpu.RandStream to provide an interface for controlling the properties of one or more random number streams that the GPU uses. Usage is the same as RandStream with the following restrictions:
  - Only the combRecursive (MRG32K3A) generator is supported.
  - Only the Inversion normal transform is supported.
  - Setting the substream property is not allowed.

Enter help parallel.gpu.RandStream at the MATLAB command line for more information.

Examples

GPU AWGN Channel

Specify the modulation order and generate PSK-modulated random data.

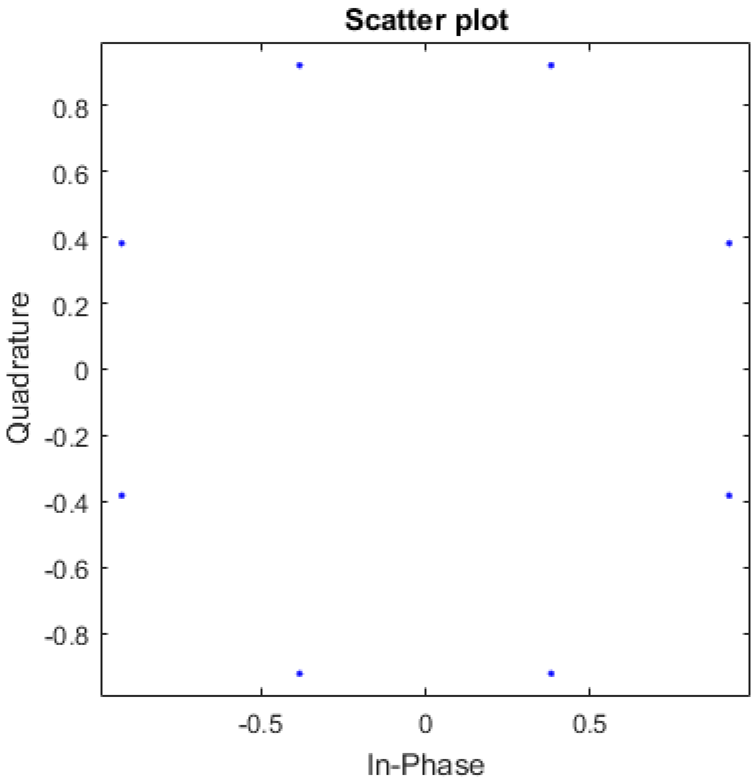
```
M = 8;
modData = pskmod(randi([0 M-1],1000,1),M,pi/M);
```

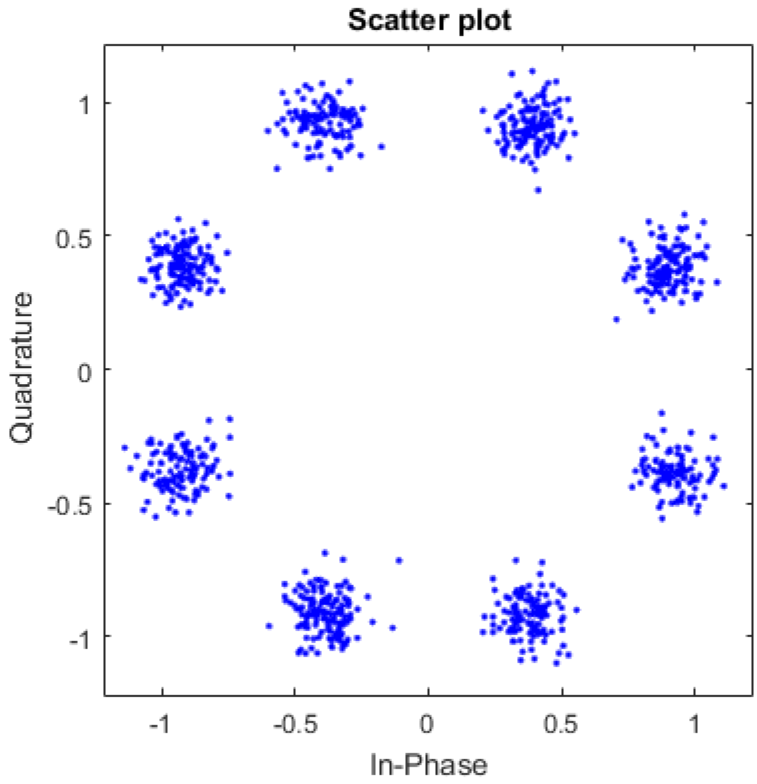
Create an AWGN channel object that uses a GPU. Pass the modulated data through the channel.

```
gpuChannel = comm.gpu.AWGNChannel('EbNo',15,'BitsPerSymbol', ...
                log2(M));
channelOutput = gpuChannel(modData);
```

Visualize the noiseless and noisy data in scatter plots.

```
scatterplot(modData)
scatterplot(channelOutput)
```


Scatter plot
Quadrature
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-0.5
0
0.5
In-Phase



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.AWGNChannel`

Introduced in R2012a

step

System object: `comm.gpu.AWGNChannel`
Package: `comm`

Add white Gaussian noise to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` adds white Gaussian noise to input `X` and returns the result in `Y`. The input `X` can be a double or single precision data type scalar, vector, or matrix with real or complex values. The dimensions of input `X` determine single or multichannel processing. For an `M`-by-`N` matrix input, `M` represents the number of time samples per channel and `N` represents the number of channels. `M` and `N` can be equal to `1`. The object adds frames of length `M` of Gaussian noise to each of the `N` channels independently.

`Y = step(H,X,VAR)` uses input `VAR` as the variance of the white Gaussian noise. This applies when you set the NoiseMethod property to `Variance` and the VarianceSource property to `Input port`. Input `VAR` can be a positive scalar or row vector with a length equal to the number of channels. `VAR` must be of the same data type as input `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.BlockDeinterleaver

Package: `comm`

Restore original ordering of block interleaved sequence with GPU

Description

The `BlockDeinterleaver` System object restores the original ordering of a sequence that was interleaved using the block interleaver System object.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To deinterleave the input vector:

1. Define and set up your block deinterleaver object. See “Construction” on page 3-726.
2. Call `step` to rearrange the elements of the input vector according to the properties of `comm.gpu.BlockDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.BlockDeinterleaver` creates a GPU-based block deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the `BlockInterleaver` System object

`H = comm.gpu.BlockDeinterleaver(Name,Value)` creates a GPU-based block deinterleaver object, `H`, with the specified property name set to the specified value.

`H = comm.gpu.BlockDeinterleaver(PERMVEC)` creates a GPU-based block deinterleaver object, `H`, with the `PermutationVector` property set to `PERMVEC`.

Properties

`PermutationVector`

Permutation vector

Specify the mapping used to permute the input symbols as a column vector of integers. The default is `[5;4;3;2;1]`. The mapping is a vector where the number of elements is equal to the length, `N`, of the input to the step method. Each element must be an integer between `1` and `N`, with no repeated values.

Methods

step	Deinterleave input sequence

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

This object uses the same algorithm as the `comm.BlockDeinterleaver` System object. See Algorithms on the `comm.BlockDeinterleaver` help page for details.

Examples

Block Interleaving and Deinterleaving with GPU

Create interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.gpu.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data intData deIntData]
```

```
ans =

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver(permVec);
deinterleaver = comm.gpu.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.BlockDeinterleaver` | `comm.gpu.BlockInterleaver`

Introduced in R2012a

step

System object: comm.gpu.BlockDeinterleaver
Package: comm

Deinterleave input sequence

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) restores the original ordering of the sequence, X, that was interleaved using a block interleaver. The step method forms the output, Y, based on the mapping specified by the PermutationVector property as Output(PermutationVector(k))=Input(k), for k = 1:N, where N is the length of the permutation vector. The input X must be a column vector of the same length, N. The data type of X can be numeric, logical, or fixed-point (fi objects). Y has the same data type as X.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.gpu.BlockInterleaver

Package: `comm`

Create block interleaved sequence with GPU

Description

The `GPU BlockInterleaver` object permutes the symbols in the input signal using a graphics processing unit (GPU).

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To interleave the input signal:

1. Define and set up your block interleaver object. See “Construction” on page 3-730.
2. Call `step` to reorder the input symbols according to the properties of `comm.gpu.BlockInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.BlockInterleaver` creates a GPU-based block interleaver System object, `H`. This object permutes the symbols in the input signal based on a permutation vector.

`H = comm.gpu.BlockInterleaver(Name,Value)` creates a GPU-based block interleaver object, `H`, with the specified property `Name` set to the specified `Value`.

`H = comm.gpu.BlockInterleaver(PERMVEC)` creates a GPU-based block deinterleaver object, `H`, with the `PermutationVector` property set to `PERMVEC`.

Properties

`PermutationVector`

Permutation vector

Specify the mapping used to permute the input symbols as a column vector of integers. The default is `[5;4;3;2;1]`. The mapping is a vector where the number of elements is equal to the length, `N`, of the input to the step method. Each element must be an integer between `1` and `N`, with no repeated values.

Methods

step	Permute input symbols using a permutation vector

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

The `GPU Block Interleaver` System object uses the same algorithm as the `comm.BlockInterleaver` System object. See Algorithms on the `comm.BlockInterleaver` help page for details.

Examples

Block Interleaving and Deinterleaving with GPU

Create interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver([3 4 1 2]');
deinterleaver = comm.gpu.BlockDeinterleaver([3 4 1 2]');
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(7,4,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence, and restored sequence.

```
[data intData deIntData]
```

```
ans =

     6     1     6
     7     7     7
     1     6     1
     7     7     7
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

Generate a random vector of unique integers as a permutation vector.

```
permVec = randperm(7)';
```

Specify `permVec` as the permutation vector for the interleaver and deinterleaver objects.

```
interleaver = comm.gpu.BlockInterleaver(permVec);
deinterleaver = comm.gpu.BlockDeinterleaver(permVec);
```

Pass random data through the interleaver and deinterleaver.

```
data = randi(10,7,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans =

     1
```

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.BlockInterleaver` | `comm.gpu.BlockDeinterleaver`

Introduced in R2012a

step

System object: `comm.gpu.BlockInterleaver`
Package: `comm`

Permute input symbols using a permutation vector

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The step method forms the output `Y`, based on the mapping defined by the `PermutationVector` property as `Output(k)=Input(PermutationVector(k))`, for `k` = 1:`N`, where `N` is the length of the `PermutationVector` property. The input `X` must be a column vector of length `N`. The data type of `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.ConvolutionalEncoder

Package: `comm.gpu`

Convolutionally encode binary data with GPU

Description

The `GPU ConvolutionalEncoder` object encodes a sequence of binary input vectors to produce a sequence of binary output vectors.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To convolutionally encode a binary signal:

1. Define and set up your convolutional encoder object. See “Construction” on page 3-734.
2. Call `step` to encode a sequence of binary input vectors to produce a sequence of binary output vectors according to the properties of `comm.gpu.ConvolutionalEncoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.ConvolutionalEncoder` creates a System object, `H`, that convolutionally encodes binary data.

`H = comm.gpu.ConvolutionalEncoder(Name,Value)` creates a convolutional encoder object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

H = comm.gpu.ConvolutionalEncoder(TRELLIS,Name,Value) creates a convolutional encoder object, H. This object has the TrellisStructure on page 3-0 property set to TRELLIS, and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. The default is the result of poly2trellis(7, [171 133]). Use the istrellis function to check if a structure is a valid trellis structure.

TerminationMethod

Termination method of encoded frame

Specify how the encoded frame is terminated as one of Continuous | Truncated | Terminated. The default is Continuous.

When you set this property to Continuous, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to Truncated, the object treats each input vector independently and resets its states to the all-zeros state.

When you set this property to Terminated, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder states to the all-zeros state at the end of the vector. For a rate *K*/*N* code, the step method outputs a vector with length $N \times {}^{(L+S)}\!/_{K}$, where $S = constraintLength–1$. In the case of multiple constraint lengths, $S = \textbf{sum}(constraintLength(i)–1))$. L is the length of the input to the step method.

ResetInputPort

Enable encoder reset input

You cannot reset this encoder object using an input port. The only valid property setting is false.

DelayedResetAction

Delay output reset

You cannot reset this encoder object using an input port. The only valid property setting is false.

InitialStateInputPort

You cannot set the initial state of this encoder object. The only valid property setting is false.

FinalStateOutputPort

You cannot output the final state of this encoder object. The only valid property setting is false.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of `None` | `Property`. The default is `None`. When you set this property to `None` the object does not apply puncturing. When you set this property to `Property`, the object punctures the code. This puncturing is based on the puncture pattern vector that you specify in the `PuncturePattern` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous` or `Truncated`.

PuncturePattern

Puncture pattern vector

Specify the puncture pattern that the object uses to puncture the encoded data as a column vector. The default is `[1; 1; 0; 1; 0; 1]`. The vector contains `1`s and `0`s, where `0` indicates a punctured, or excluded, bit. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous` or `Truncated` and the `PuncturePatternSource` on page 3-0 property to `Property`.

NumFrames

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames contained in a single data input/output vector. The default value of this property is `1`. The objects segments the input vector into `NumFrames` segments and encodes them independently. The output contains `NumFrames` encoded segments. This property is applicable when you set the `TerminationMethod` on page 3-0 to `Terminated` or `Truncated`.

Methods

reset	Reset states of the convolutional encoder object
step	Convolutionally encode binary data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

8-PSK-Modulation With Convolutional Encoding

Transmit a Convolutionally Encoded, 8-PSK-Modulated Bit Stream Through an AWGN Channel.

Create a GPU-based Convolutional Encoder System object.

```
hConEnc = comm.gpu.ConvolutionalEncoder;
```

Create a GPU-based PSK Modulator System object that accepts a bit input signal.

```
hMod = comm.gpu.PSKModulator('BitInput',true);
```

Create a GPU-based AWGN Channel System object with a signal-to-noise ratio of seven.

```
hChan = comm.gpu.AWGNChannel('NoiseMethod', ...
         'Signal to noise ratio (SNR)',...
         'SNR',7);
```

Create a GPU-based PSK Demodulator System object that outputs a column vector of bit values.

```
hDemod = comm.gpu.PSKDemodulator('BitOutput',true);
```

Create a GPU-based Viterbi Decoder System object that accepts an input vector of hard decision values, which are zeros or ones.

```
hDec = comm.gpu.ViterbiDecoder('InputFormat','Hard');
```

Create an Error Rate System object that ignores 3 data samples before makings comparisons. The received data lags behind the transmitted data by 34 samples.

```
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay', 34);
```

Run the simulation by using the step method to process data.

```
for counter = 1:20
  data = randi([0 1],30,1);
  encodedData = step(hConEnc, gpuArray(data));
  modSignal = step(hMod, encodedData);
  receivedSignal = step(hChan, modSignal);
  demodSignal = step(hDemod, receivedSignal);
  receivedBits = step(hDec, demodSignal);
  errors = step(hError, data, gather(receivedBits));
end
```

Display the errors.

```
disp(errors)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Encoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.ConvolutionalEncoder` | `comm.gpu.ConvolutionalDeinterleaver` | `comm.gpu.ConvolutionalInterleaver` | `comm.gpu.ViterbiDecoder`

Introduced in R2012a

reset

System object: `comm.gpu.ConvolutionalEncoder`
Package: `comm.gpu`

Reset states of the convolutional encoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalEncoder` object, `H`.

step

System object: `comm.gpu.ConvolutionalEncoder`
Package: `comm.gpu`

Convolutionally encode binary data

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` encodes the binary data, `X`, using the convolutional encoding that you specify in the TrellisStructure property. It returns the encoded data, `Y`. Both `X` and `Y` are column vectors of data type single, double, or logical. When the convolutional encoder represents a rate *K*/*N* code, the length of the input vector equals *K* × *L*, for a positive integer, *L*. The step method sets the length of the output vector, `Y`, to *L* × *N*.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.ConvolutionalInterleaver

Package: comm

Permute input symbols using shift registers with GPU

Description

The GPU ConvolutionalInterleaver object permutes the symbols in the input signal using a graphics processing unit (GPU). Internally, this class uses a set of shift registers.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the gpuArray class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a gpuArray, the data remains on the GPU. The output signal is a gpuArray. When the object is given a gpuArray, calculations take place entirely on the GPU, and no data transfer occurs. Passing gpuArray arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To convolutionally interleave binary data:

1. Define and set up your convolutional interleaver object. See "Construction" on page 3-740.
2. Call step to convolutionally interleave according to the properties of comm.gpu.ConvolutionalInterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.gpu.ConvolutionalInterleaver creates a GPU-based convolutional interleaver System object, H. This object permutes the symbols in the input signal using a set of shift registers.

H = comm.gpu.ConvolutionalInterleaver(Name,Value) creates a GPU-based convolutional interleaver System object, H, with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.gpu.ConvolutionalInterleaver(M,B,IC) creates a GPU-based convolutional interleaver System object H, with the NumRegisters property set to M, the RegisterLengthStep property set to B, and the InitialConditions property set to IC. M, B, and IC are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments.

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is 6.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is 2. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register as a numeric scalar or vector. You do not need to specify a value for the first shift register, which has zero delay. The default is 0. The value of the first element of this property is unimportant because the first shift register has zero delay. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the NumRegisters on page 3-0 property, then the *i*-th shift register stores the *i*-th element of the specified vector.

Methods

reset	Reset states of the convolutional interleaver object
step	Permute input symbols using shift registers

Common to All System Objects	
release	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving with GPU

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.gpu.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.gpu.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans =

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
    10    10     4
    11     5     5
    12    12     6
    13     7     7
    14    14     8
    15     9     9
    16    16    10
    17    11    11
    18    18    12
    19    13    13
    20    20    14
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
intrlvDelay =

     6


numSymErrors =

     0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Interleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.ConvolutionalInterleaver` | `comm.gpu.ConvolutionalDeinterleaver`

Introduced in R2012a

reset

System object: `comm.gpu.ConvolutionalInterleaver`
Package: `comm`

Reset states of the convolutional interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalInterleaver` object, `H`.

step

System object: `comm.gpu.ConvolutionalInterleaver`
Package: `comm`

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input X must be a column vector. The data type can be of type `double`, `single`, `uint32`, `int32`, or `logical`. `Y` has the same data type as `X`. The convolutional interleaver object uses a set of *N* shift registers, where *N* is the value specified by the `NumRegisters` property. The object sets the delay value of the k^{th} shift register to the product of (*k*-1) and the RegisterLengthStep property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the N^{th} register and the next new input occurs, it returns to the first register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.ConvolutionalDeinterleaver

Package: `comm`

Restore ordering of symbols using shift registers with GPU

Description

The `GPU ConvolutionalDeinterleaver` object recovers a signal that was interleaved using the GPU-based convolutional interleaver object. The parameters in the two blocks should have the same values.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To recover convolutionally interleaved binary data:

1. Define and set up your convolutional deinterleaver object. See "Construction" on page 3-746.
2. Call `step` to convolutionally deinterleave according to the properties of `comm.gpu.ConvolutionalDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.ConvolutionalDeinterleaver` creates a GPU-based convolutional deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using a convolutional interleaver.

`H = comm.gpu.ConvolutionalDeinterleaver(Name,Value)` creates a GPU-based convolutional deinterleaver System object, `H`, with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

H = comm.gpu.ConvolutionalDeinterleaver(M,B,IC) creates a convolutional deinterleaver System object H, with the NumRegisters property set to M, the RegisterLengthStep property set to B, and the InitialConditions property set to IC. M, B, and IC are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments.

Properties

NumRegisters

Number of internal shift registers

Specify the number of internal shift registers as a scalar, positive integer. The default is 6.

RegisterLengthStep

Number of additional symbols that fit in each successive shift register

Specify the number of additional symbols that fit in each successive shift register as a positive, scalar integer. The default is 2. The first register holds zero symbols.

InitialConditions

Initial conditions of shift registers

Specify the values that are initially stored in each shift register (except the first shift register, which has zero delay) as a numeric scalar or vector. The default is 0. If you set this property to a scalar, then all shift registers, except the first one, store the same specified value. If you set it to a column vector with length equal to the value of the NumRegisters on page 3-0 property, then the *i*-th shift register stores the *i*-th element of the specified vector. The value of the first element of this property is unimportant, since the first shift register has zero delay.

Methods

step	Permute input symbols using shift registers
reset	Reset states of the convolutional deinterleaver object

Common to All System Objects	
release	Allow System object property value changes

Examples

Convolutional Interleaving and Deinterleaving with GPU

Create convolutional interleaver and deinterleaver objects.

```
interleaver = comm.gpu.ConvolutionalInterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
deinterleaver = comm.gpu.ConvolutionalDeinterleaver('NumRegisters',2, ...
    'RegisterLengthStep',3);
```

Generate data, and pass the data through the convolutional interleaver. Pass the interleaved data through the convolutional deinterleaver.

```
data = (0:20)';
intrlvData = interleaver(data);
deintrlvData = deinterleaver(intrlvData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intrlvData deintrlvData]
```

```
ans =

     0     0     0
     1     0     0
     2     2     0
     3     0     0
     4     4     0
     5     0     0
     6     6     0
     7     1     1
     8     8     2
     9     3     3
    10    10     4
    11     5     5
    12    12     6
    13     7     7
    14    14     8
    15     9     9
    16    16    10
    17    11    11
    18    18    12
    19    13    13
    20    20    14
```

The delay through the interleaver and deinterleaver pair is equal to the product of the `NumRegisters` and `RegisterLengthStep` properties. After accounting for this delay, confirm that the original and deinterleaved data are identical.

```
intrlvDelay = interleaver.NumRegisters * interleaver.RegisterLengthStep
numSymErrors = symerr(data(1:end-intrlvDelay),deintrlvData(1+intrlvDelay:end))
```

```
intrlvDelay =

     6


numSymErrors =
     0
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Convolutional Deinterleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see "Simulation Acceleration Using GPUs".

See Also

`comm.ConvolutionalDeinterleaver` | `comm.gpu.ConvolutionalInterleaver`

Introduced in R2012a

step

System object: `comm.gpu.ConvolutionalDeinterleaver`
Package: `comm`

Permute input symbols using shift registers

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a convolutional interleaver and returns `Y`. The input `X` must be a column vector. The data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The convolutional deinterleaver object uses a set of `N` shift registers, where `N` represents the value specified by the `NumRegisters` property. The object sets the delay value of the k^{th} shift register to the product of (k-1) and the RegisterLengthStep property value. With each new input symbol, a commutator switches to a new register and the new symbol shifts in while the oldest symbol in that register shifts out. When the commutator reaches the *N*th register and the next new input occurs, it returns to the first register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

reset

System object: `comm.gpu.ConvolutionalDeinterleaver`
Package: `comm`

Reset states of the convolutional deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU ConvolutionalDeinterleaver` object, `H`.

comm.gpu.LDPCDecoder

Package: `comm`

Decode binary low-density parity-check (LDPC) code with GPU

Note To use this object, you must install Parallel Computing Toolbox™ and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

Description

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

The `comm.gpu.LDPCDecoder` System object uses the belief propagation algorithm to decode a binary LDPC code, which is input to the object as the soft-decision output (log-likelihood ratio of received bits) from demodulation. The object decodes generic binary LDPC codes where no patterns in the parity-check matrix are assumed. For more information, see “Belief Propagation Decoding” on page 3-758.

To decode an LDPC-encoded signal:

1. Create the `comm.gpu.LDPCDecoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
gpu_ldpcdecoder = comm.gpu.LDPCDecoder
gpu_ldpcdecoder = comm.gpu.LDPCDecoder(parity)
gpu_ldpcdecoder = comm.gpu.LDPCDecoder(___,Name,Value)
```

Description

`gpu_ldpcdecoder = comm.gpu.LDPCDecoder` creates a GPU-based binary LDPC decoder System object. This object performs LDPC decoding based on the specified parity-check matrix.

`gpu_ldpcdecoder = comm.gpu.LDPCDecoder(parity)` sets the `ParityCheckMatrix` property to `parity` and creates a GPU-based LDPC decoder System object. The `parity` input must be specified as described by the `ParityCheckMatrix` property.

`gpu_ldpcdecoder = comm.gpu.LDPCDecoder( ___ ,Name,Value)` sets properties using one or more name-value pairs, in addition to inputs from any of the prior syntaxes. For example, `comm.LDPCDecoder('DecisionMethod','Soft decision')` configures an LDPC decoder System object to decode data using the soft-decision method and output log-likelihood ratios of data type `double`. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`ParityCheckMatrix` — Parity-check matrix

`dvbs2ldpc(1/2)` (default) | sparse binary matrix | nonsparse index matrix

Parity-check matrix, specified as a sparse (N – K)-by-N binary-valued matrix. N is the length of the received signal and must be in the range (0, 2^{31}). K is the length of the uncoded message and must be less than N. The last (N – K) columns in the parity-check matrix must be an invertible matrix in the Galois field of order 2, `gf`(2).

You can also specify the parity-check matrix as a two-column nonsparse index matrix, `I`, that defines the row and column indices of the `1`s in the parity-check matrix such that `sparse(I(:,1),I(:,2),1)`.

This property accepts numeric data types. When you set this property to a sparse binary matrix, this property also accepts the `logical` data type.

The default value uses the `dvbs2ldpc` function to configure a sparse parity-check matrix for half-rate LDPC coding, as specified in the DVB-S.2 standard.

Example: `dvbs2ldpc(R,'indices')` configures the index matrix for the DVB-S.2 standard, where `R` is the code rate, and `'indices'` specifies the output format of `dvbs2ldpc` as a two-column double-precision matrix that defines the row and column indices of the `1`s in the parity-check matrix.

Data Types: `double` | `logical`

`OutputValue` — Output value format

`'Information part'` (default) | `'Whole codeword'`

Output value format, specified as one of these values:

- `'Information part'` — The object outputs a K-by-1 column vector containing only the information-part of the received log-likelihood ratio vector. K is the length of the uncoded message.
- `'Whole codeword'` — The object outputs an N-by-1 column vector containing the whole log-likelihood ratio vector. N is the length of the received signal.

N and *K* must align with the dimension of the (*N*–*K*)-by-*K* parity-check matrix.

Data Types: `char`

`DecisionMethod` — Decision method
`'Hard decision'` (default) | `'Soft decision'`

Decision method used for decoding, specified as one of these values:

- `'Hard decision'` — The object outputs decoded data of data type `logical`.
- `'Soft decision'` — The object outputs log-likelihood ratios of data type `double`.

Data Types: `char`

`IterationTerminationCondition` — Condition for iteration termination
`'Maximum iteration count'` (default) | `'Parity check satisfied'`

Condition for iteration termination, specified as one of these values:

- `'Maximum iteration count'` — Decoding terminates after the number of iterations specified by the `MaximumIterationCount` property.
- `'Parity check satisfied'` — Decoding terminates after all parity checks are satisfied. If not all parity checks are satisfied, decoding terminates after the number of iterations specified by the `MaximumIterationCount` property.

Data Types: `char`

`MaximumIterationCount` — Maximum number of decoding iterations
`50` (default) | positive integer

Maximum number of decoding iterations, specified as a positive integer.

Data Types: `double`

`NumIterationsOutputPort` — Output number of iterations performed
`false` (default) | `true`

Output number of iterations executed, specified as `false` or `true`. To output the number of iterations executed, set this property to `true`.

Data Types: `logical`

`FinalParityChecksOutputPort` — Output final parity checks
`false` (default) | `true`

Output final parity checks, specified as `false` or `true`. To output the final calculated parity checks, set this property to `true`.

Data Types: `logical`

Usage

Syntax

```
y = gpu_ldpcdecoder(x)
```

```
[y,numiter] = gpu_ldpcdecoder(x)
[y,parity] = gpu_ldpcdecoder(x)
[y,numiter,parity] = gpu_ldpcdecoder(x)
```

Description

`y = gpu_ldpcdecoder(x)` decodes input data using an LDPC code based on the default parity-check matrix.

`[y,numiter] = gpu_ldpcdecoder(x)` returns the decoded data, `y`, and number of iterations performed, `numiter`. To use this syntax, set the `NumIterationsOutputPort` property to `true`.

`[y,parity] = gpu_ldpcdecoder(x)` returns the decoded data, `y`, and final parity checks, `parity`. To use this syntax, set the `FinalParityChecksOutputPort` property to `true`.

`[y,numiter,parity] = gpu_ldpcdecoder(x)` returns the decoded data, number of iterations performed, and final parity checks. To use this syntax, set the `NumIterationsOutputPort` and `FinalParityChecksOutputPort` properties to `true`.

Input Arguments

x — Log-likelihood ratios

column vector

Log-likelihood ratios, specified as an *N*-by-1 column vector containing the soft-decision output from demodulation. *N* is the number of bits in the LDPC codeword before modulation. Each element is the log-likelihood ratio for a received bit. Element values are more likely to be `0` if the log-likelihood ratio is positive. The first *K* elements correspond to the information-part of the input message.

Data Types: `double`

Output Arguments

y — Decoded data

column vector

Decoded data, returned as a column vector. The `DecisionMethod` property specifies whether the object outputs hard decisions or soft decisions (log-likelihood ratios).

- If the `OutputValue` property is set to `'Information part'`, the output includes only the information-part of the received log-likelihood ratio vector.
- If the `OutputValue` property is set to `'Whole codeword'`, the output includes the whole log-likelihood ratio vector.

Data Types: `double` | `logical`

`numiter` — Number of executed decoding iterations

positive integer

Number of executed decoding iterations, returned as a positive integer.

Dependencies

To enable this output, set the `NumIterationsOutputPort` property to `true`.

`parity` — Final parity checks

column vector

Final parity checks after decoding the input LDPC code, returned as an (*N*-*K*)-by-1 column vector. *N* is the number of bits in the LDPC codeword before modulation. *K* is the length of the uncoded message.

Dependencies

To enable this output, set the `FinalParityChecksOutputPort` property to `true`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

LDPC Encode and Decode QPSK-Modulated Signal Using GPU

Using a `comm.gpu.LDPCDecoder` System Object™ to decode the signal, transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel. After adding AWGN, demodulate and decode the received signal. Compute the error statistics for the reception of uncoded and LDPC-coded signals. For more information, see “Simulation Acceleration Using GPUs”.

Define simulation variables. Create System objects for the LDPC encoder, LDPC decoder, QPSK modulator, and QPSK demodulators.

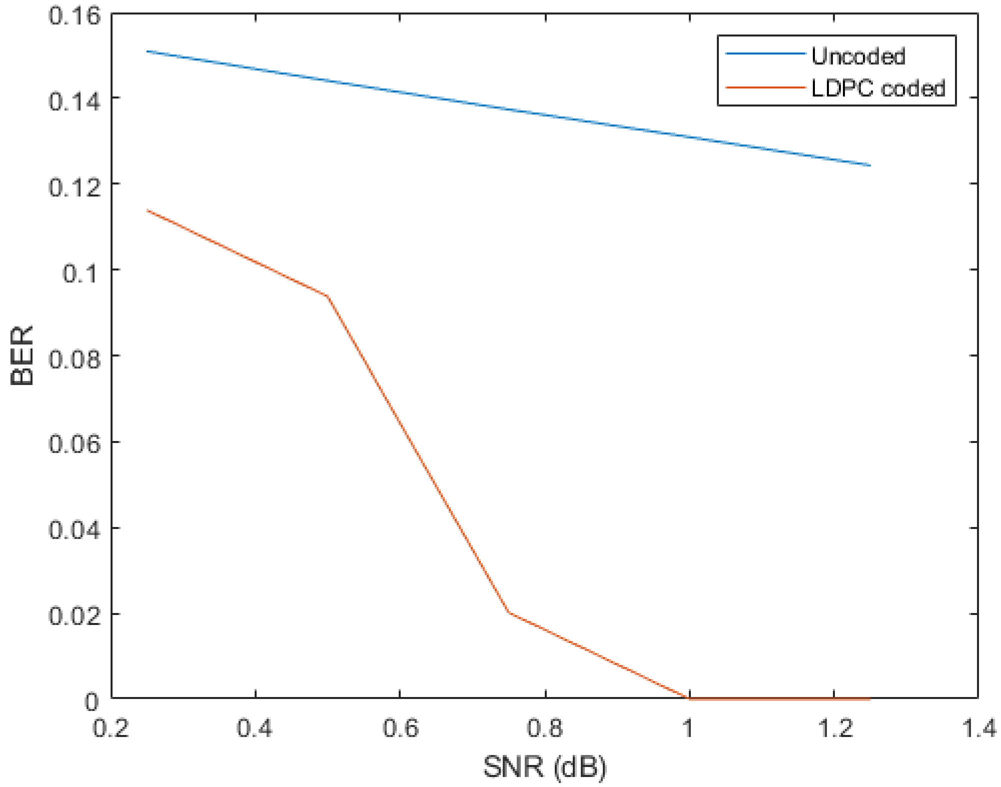
```
M = 4; % Modulation order (QPSK)
snr = [0.25,0.5,0.75,1.0,1.25];
numFrames = 10;
ldpcEncoder = comm.LDPCEncoder;
gpuldpcDecoder = comm.gpu.LDPCDecoder;
pskMod = comm.PSKModulator(M,'BitInput',true);
pskDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Approximate log-likelihood ratio');
pskuDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Hard decision');
errRate = zeros(1,length(snr));
uncErrRate = zeros(1,length(snr));
```

For each SNR setting and all frames, compute the error statistics for uncoded and LDPC-coded signals.

```
for ii = 1:length(snr)
    ttlErr = 0;
    ttlErrUnc = 0;
    pskDemod.Variance = 1/10^(snr(ii)/10); % Set variance using current SNR
    for counter = 1:numFrames
        data = logical(randi([0 1],32400,1));
        % Transmit and receiver uncoded signal data
        mod_uncSig = pskMod(data);
        rx_uncSig = awgn(mod_uncSig,snr(ii),'measured');
        demod_uncSig = pskuDemod(rx_uncSig);
        numErrUnc = biterr(data,demod_uncSig);
        ttlErrUnc = ttlErrUnc + numErrUnc;
        % Transmit and receive LDPC coded signal data
        encData = ldpcEncoder(data);
        modSig = pskMod(encData);
        rxSig = awgn(modSig,snr(ii),'measured');
        demodSig = pskDemod(rxSig);
        rxBits = gpuldpcDecoder(demodSig);
        numErr = biterr(data,rxBits);
        ttlErr = ttlErr + numErr;
    end
    ttlBits = numFrames*length(rxBits);
    uncErrRate(ii) = ttlErrUnc/ttlBits;
    errRate(ii) = ttlErr/ttlBits;
end
```

Plot the error statistics for uncoded and LDPC-coded data.

```
plot(snr,uncErrRate,snr,errRate)
legend('Uncoded','LDPC coded')
xlabel('SNR (dB)')
ylabel('BER')
```

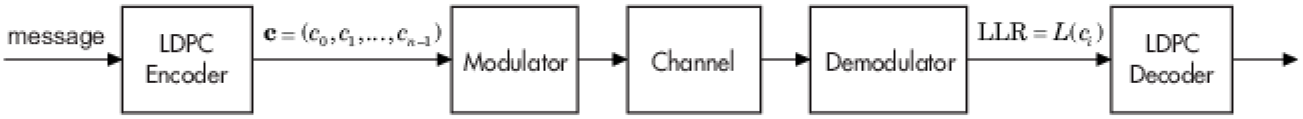


Algorithms

This object performs LDPC decoding using the belief propagation algorithm, also known as a message-passing algorithm.

Belief Propagation Decoding

The implementation of the belief propagation algorithm is based on the decoding algorithm presented by Gallager.



For transmitted LDPC-encoded codeword $c = c_0, c_1, \ldots, c_{n-1}$, the input to the LDPC decoder is the log-likelihood ratio (LLR) value $L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid \text{channel output for } c_i)}{\Pr(c_i = 1 \mid \text{channel output for } c_i)}\right)$.

In each iteration, the key components of the algorithm are updated based on these equations:

$$L(r_{ji}) = 2\,\mathrm{atanh}\left(\prod_{i' \in V_j \backslash i} \tanh\left(\frac{1}{2} L(q_{i'j})\right)\right),$$

$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \backslash j} L(r_{j'i})$, initialized as $L(q_{ij}) = L(c_i)$ before the first iteration, and

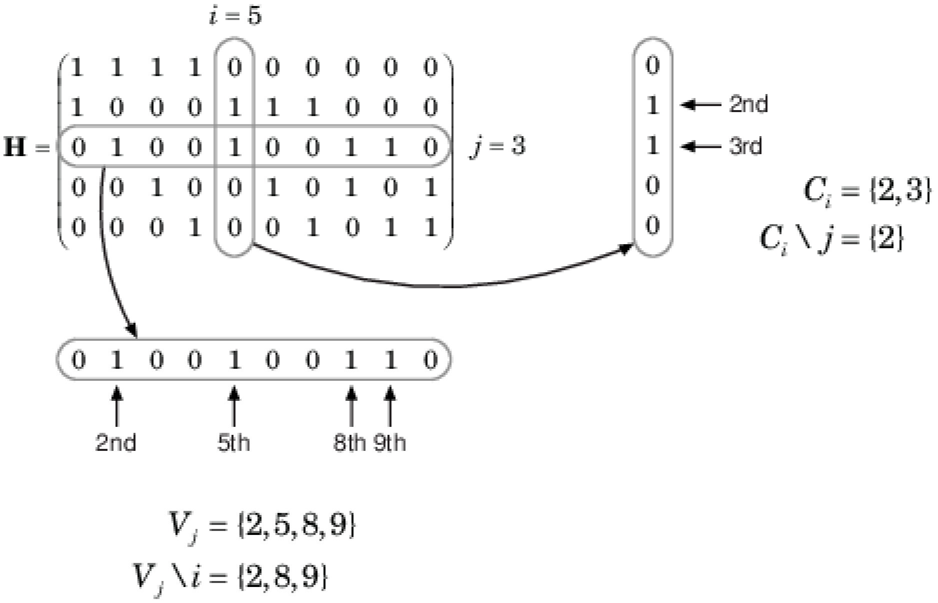
$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i})$.

At the end of each iteration, $L(Q_i)$ contains the updated estimate of the LLR value for transmitted bit c_i. The value $L(Q_i)$ is the soft-decision output for c_i. If $L(Q_i) < 0$, the hard-decision output for c_i is 1. Otherwise, the hard-decision output for c_i is 0.

If configured to stop when all parity checks are satisfied, the algorithm verifies the parity-check equation ($H\ c' = 0$) at the end of each iteration. When all parity checks are satisfied, or if the maximum number of iterations is reached, decoding stops.

Index sets $C_i \backslash j$ and $V_j \backslash i$ are based on the parity-check matrix (PCM). Index sets C_i and V_j correspond to all nonzero elements in column i and row j of the PCM, respectively.

This figure highlights the computation of these index sets in a given PCM for $i = 5$ and $j = 3$.



To avoid infinite numbers in the algorithm equations, atanh(1) and atanh(-1) are set to 19.07 and –19.07, respectively. Due to finite precision, MATLAB returns 1 for tanh(19.07) and –1 for tanh(-19.07).

References

[1] Gallager, Robert G. *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

Objects
`comm.BCHDecoder` | `comm.LDPCDecoder` | `comm.LDPCEncoder`

Functions
`dvbs2ldpc`

Blocks
LDPC Decoder

Topics
“Simulation Acceleration Using GPUs”

Introduced in R2012a

comm.gpu.PSKDemodulator

Package: `comm`

Demodulate using M-ary PSK method with GPU

Description

The `GPU PSKDemodulator` object demodulates an input signal using the M-ary phase shift keying (M-PSK) method.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To demodulate a signal that was modulated using phase shift keying:

1. Define and set up your PSK demodulator object. See “Construction” on page 3-761.
2. Call `step` to demodulate the signal according to the properties of `comm.gpu.PSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.PSKDemodulator` returns a GPU-based demodulator System object, `H`. This object demodulates the input signal using the M-ary phase shift keying (M-PSK) method.

`H = comm.gpu.PSKDemodulator(Name,Value)` creates a GPU-based M-PSK demodulator object, `H,` with the specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN)

`H = comm.gpu.PSKDemodulator(M,PHASE,Name,Value)` creates a GPU-based M-PSK demodulator object, `H`, with the ModulationOrder property set to `M`, the PhaseOffset property set to `PHASE` and other specified property names set to the specified values. `M` and `PHASE` are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar. The default is 8.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar. The default is π/8.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. When you set this property to true, the step method outputs a column vector of bit values with length equal to log2(ModulationOrder) times the number of demodulated symbols. When you set this property to false, the step method outputs a column vector, with a length equal to the input data vector that contains integer symbol values between 0 and ModulationOrder-1. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of log2(ModulationOrder) bits to the corresponding symbol as Binary | Gray | Custom. The default is Gray. When you set this property to Gray, the object uses a Gray-encoded signal constellation. When you set this property to Binary, the integer m (0 ≤ m ≤ModulationOrder-1) maps to the complex value. This value is represented as exp(j*PhaseOffset + j*2*pi*m/ModulationOrder). When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is 0:7. This property must be a row or column vector of size ModulationOrder with unique integer values in the range [0, ModulationOrder-1]. The values must be of data type double. The first element of this vector corresponds to the constellation point at an angle of 0 + PhaseOffset, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of -π/ModulationOrder + PhaseOffset. This property applies when you set the SymbolMapping property to Custom.

DecisionMethod

Demodulation decision method

Specify the decision method that the object uses as one of `Hard decision` | `Log-likelihood ratio` | `Approximate log-likelihood ratio`. The default is `Hard decision`. When you set `DecisionMethod` to false, the object always performs hard decision demodulation. This property applies when you set the BitOutput property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property` | `Input port`. The default is `Property`. This property applies when you set the BitOutput property to `true` and the DecisionMethod property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Variance

Specify the variance of the noise as a positive, real scalar. The default is `1`. If this value is very small (i.e., SNR is very high), then log-likelihood ratio (LLR) computations may yield Inf or -Inf. This occurs because the LLR algorithm computes the exponential value of very large or very small numbers using finite precision arithmetic. In such cases, use approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the BitOutput property to true, the DecisionMethod property to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, and the VarianceSource property to `Property`. This property is tunable.

OutputDataType

Data type of output

When you set this property to `Full precision`, the output signal inherits its data type from the input signal.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using M-ary PSK method

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

The `GPU PSK Demodulator` System object uses the same algorithm as the `comm.PSKDemodulator` System object. See Decoding Algorithm for details.

Examples

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel. Then demodulate, decode, and count errors.

16-PSK Modulation and Demodulation

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel.

Create a GPU-based PSK Modulator System object.

```
hMod = comm.gpu.PSKModulator(16, 'PhaseOffset',pi/16);
```

Create a GPU-based AWGN Channel System object with a signal-to-noise ratio of 15.

```
hAWGN = comm.gpu.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)','SNR',15);
```

Create a GPU-based PSK Demodulator System object.

```
 hDemod = comm.gpu.PSKDemodulator(16, 'PhaseOffset',pi/16);
```

Create an error rate calculator System object.

```
hError = comm.ErrorRate;
```

Transmit a frame of data containing 50 symbols.

```
for counter = 1:100
data = gpuArray.randi([0 hMod.ModulationOrder-1], 50, 1);
```

Run the simulation by using the step method to process data.

```
modSignal = step(hMod, data);
noisySignal = step(hAWGN, modSignal);
receivedData = step(hDemod, noisySignal);
errorStats = step(hError, gather(data), gather(receivedData));
end
```

Compute the error rate results.

```
fprintf('Error rate = %f\nNumber of errors = %d\n',...
    errorStats(1), errorStats(2))
```

GPU PSK Demodulator

Create GPU PSK modulator and demodulator pair.

```
gpuMod = comm.gpu.PSKModulator;
gpuDemod = comm.gpu.PSKDemodulator;
```

Generate random data symbols. Modulate the data.

```
txData = randi([0 7],1000,1);
txSig = gpuMod(txData);
```

Pass the signal through an AWGN channel.

```
rxSig = awgn(txSig,20);
```

Demodulate the received signal.

```
rxData = gpuDemod(rxSig);
```

Determine the number of symbol errors.

```
numSymErrors = symerr(txData,rxData)
```

```
numSymErrors =
```

736

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.PSKDemodulator` | `comm.gpu.PSKModulator`

Introduced in R2012a

constellation

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.gpu.PSKDemodulator

Create a `comm.gpu.PSKDemodulator` System object, and then calculate its ideal signal constellation.

Create a `comm.gpu.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKDemodulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.gpu.PSKDemodulator

Create a `comm.gpu.PSKDemodulator` System object, and then plot the ideal signal constellation.

Create a `comm.gpu.PSKDemodulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKDemodulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

step

Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates data, `X`, with the `GPU PSK Demodulator` System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double- or single- precision data type. Depending on the BitOutput property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the BitOutput property to true, the DecisionMethod property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the VarianceSource property to `Input port`. The data type of input `VAR` must be double or single precision.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.PSKModulator

Package: `comm`

Modulate using M-ary PSK method with GPU

Description

The `GPU PSKModulator` object modulates a signal using the M-ary phase shift keying method implemented on a graphics processing unit (GPU). The input is a baseband representation of the modulated signal. The input and output for this object are discrete-time signals. This object accepts a scalar-valued or column vector input signal.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see “GPU Computing” (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see “Establish Arrays on a GPU” (Parallel Computing Toolbox).

To modulate a signal using phase shift keying:

1. Define and set up your PSK modulator object. See “Construction” on page 3-768.
2. Call `step` to modulate the signal according to the properties of `comm.gpu.PSKModulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.PSKModulator` returns a GPU-based demodulator System object, `H`. This object modulates the input signal using the M-ary phase shift keying (M-PSK) method with soft decision using the approximate log-likelihood ratio algorithm.

`H = comm.gpu.PSKModulator(Name,Value)` creates a GPU-based M-PSK modulator object, `H`, with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN)

`H = comm.gpu.PSKModulator(M,PHASE,Name,Value)` creates a GPU-based M-PSK modulator object, `H`, with the ModulationOrder property set to `M`, the `PhaseOffset` property set to PHASE and other specified property Names set to the specified Values. `M` and PHASE are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar. The default is `8`.

`PhaseOffset`

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar. The default is π/8.

`BitInput`

Assume bit inputs

Specify whether the input is bits or integers. The default is false. When you set this property to true, the step method input must be a column vector of bit values whose length is an integer multiple of log2(ModulationOrder). This vector contains bit representations of integers between 0 and ModulationOrder-1. The input data type can be numeric or logical. When you set the BitInput property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder-1. The data type of the input must be numeric.

`SymbolMapping`

Constellation encoding

Specify how the object maps an integer or group of log2(ModulationOrder) bits to the corresponding symbol as one of `Binary` | `Gray` | `Custom`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer m (0 ≤ m ≤ ModulationOrder-1) maps to the complex value exp(j*PhaseOffset + j*2*pi*m/ModulationOrder). When you set this property to `Custom`, the object uses the signal constellation defined in the CustomSymbolMapping property.

`CustomSymbolMapping`

Custom constellation encoding

Specify a custom constellation symbol mapping vector. This property must be a row or column vector of size `ModulationOrder` with unique integer values in the range [0, ModulationOrder-1]. The values must be of data type double. The first element of this vector corresponds to the constellation point at an angle of 0 + `PhaseOffset`, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of -π/ModulationOrder + `PhaseOffset`. This property applies when you set the `SymbolMapping` property to Custom. The default is `0:7`.

OutputDataType

Data type of output

Specify the output data type as one of `double` | `single`. The default is `double`.

Methods

constellation	Calculate or plot ideal signal constellation
step	Modulate using M-ary PSK method with GPU

Common to All System Objects	
`release`	Allow System object property value changes

Algorithm

The `GPU PSK Modulator` System object supports floating-point and integer input data types. This object uses the same algorithm as the `comm.PSKModulator` System object. See the Algorithms section of the `comm.PSKModulator` help page for details.

Examples

GPU PSK Modulator

Create binary data for 100, 4-bit symbols

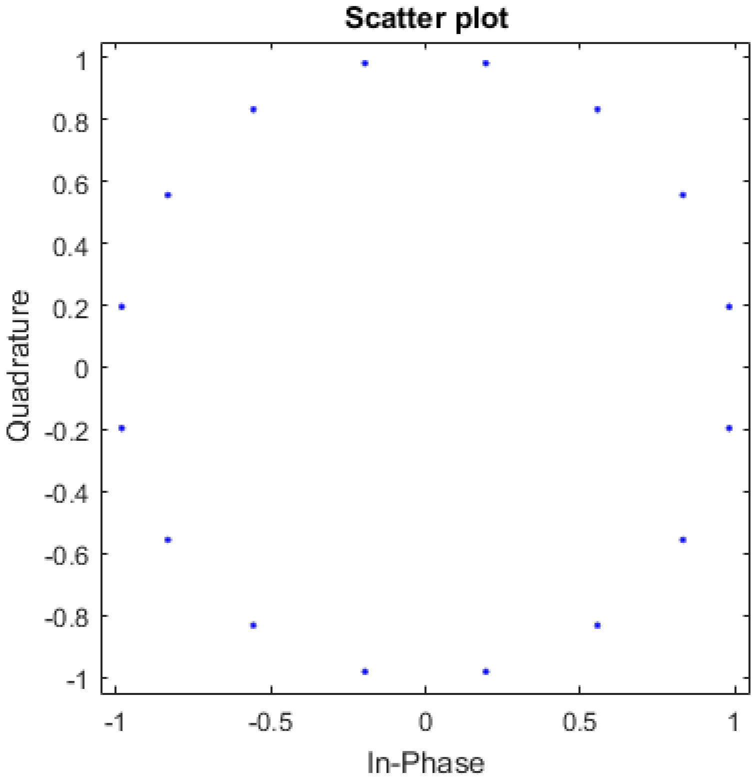
```
data = randi([0 1],400,1);
```

Create a 16-PSK modulator System object with bits as inputs and Gray-coded signal constellation. Change the phase offset to $\pi/16$.

```
gpuMod = comm.gpu.PSKModulator(16,'BitInput',true);
gpuMod.PhaseOffset = pi/16;
```

Modulate and plot the data

```
modData = gpuMod(data);
scatterplot(modData)
```



Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see "Simulation Acceleration Using GPUs".

See Also

`comm.gpu.PSKDemodulator`

Introduced in R2012a

constellation

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Calculate Ideal Signal Constellation for comm.gpu.PSKModulator

Create a `comm.gpu.PSKModulator` System object, and then calculate its ideal signal constellation.

Create a `comm.gpu.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKModulator
```

Calculate and display the ideal signal constellation by calling the `constellation` method.

```
a = constellation(h)
```

Plot Ideal Signal Constellation for comm.gpu.PSKModulator

Create a `comm.gpu.PSKModulator` System object, and then plot the ideal signal constellation.

Create a `comm.gpu.PSKModulator` System object by entering the following at the MATLAB command line:

```
h = comm.gpu.PSKModulator
```

Plot the ideal signal constellation by calling the `constellation` method.

```
constellation(h)
```

step

Modulate using M-ary PSK method with GPU

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates the input data, `X`, using the GPU-based PSK modulator System object, `H`. The object returns the baseband modulated output `Y`. Depending upon the value of the BitInput property, input `X` can be an integer or bit-valued column vector with numeric or logical data types.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.TurboDecoder

Package: `comm.gpu`

Decode input signal using parallel concatenation decoding with GPU

Description

The `GPU Turbo Decoder` System object decodes the input signal using a parallel concatenated decoding scheme. This scheme uses the *a-posteriori* probability (APP) decoder as the constituent decoder. Both constituent decoders use the same trellis structure and algorithm.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To decode an input signal using a turbo decoding scheme:

1. Define and set up your turbo decoder object. See "Construction" on page 3-774.
2. Call `step` to decode a binary signal according to the properties of `comm.gpu.TurboDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.TurboDecoder` creates a GPU-based turbo decoder System object, `H`. This object uses the *a-posteriori* probability (APP) constituent decoder to iteratively decode the parallel-concatenated convolutionally encoded input data.

`H = comm.gpu.TurboDecoder(Name, Value)` creates a GPU-based turbo decoder object, `H`, with the specified property name set to the specified value. `Name` must appear inside single quotes (`''`). You can specify several name-value pair arguments in any order as `Name1,Value1,…,NameN,ValueN`.

H = comm.gpu.TurboDecoder(TRELLIS, INTERLVRINDICES, NUMITER) creates a GPU-based turbo decoder object, H. In this object, the TrellisStructure property is set to TRELLIS, the InterleaverIndices property set to INTERLVRINDICES, and the NumIterations property set to NUMITER.

Properties

TrellisStructure

Trellis structure of constituent convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. The default is the result of poly2trellis(4, [13 15], 13). Use the istrellis function to check if a structure is a valid trellis structure.

InterleaverIndicesSource

Source of interleaver indices

Specify the source of the interleaver indices. The only valid setting for this property is Property.

InterleaverIndices

Interleaver indices

Specify the mapping used to permute the input bits at the encoder as a column vector of integers. The default is (64:-1:1).'. This mapping is a vector with the number of elements equal to the length, *L*, of the output of the step method. Each element must be an integer between 1 and *L*, with no repeated values.

Algorithm

Decoding algorithm

Specify the decoding algorithm. This object implements true *a posteriori* probability decoding. The only valid setting is True APP.

NumScalingBits

Number of scaling bits

The GPU version of the Turbo Decoder does not use this property.

NumIterations

Number of decoding iterations

Specify the number of decoding iterations used for each call to the step method. The default is 6. The object iterates and provides updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the step method is the hard-decision output of the final LLR update.

NumFrames

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames that a single data input/output vector contains. The default value of this property is `1`. This object segments the input vector into `NumFrames` segments and decodes the segments independently. The output contains `NumFrames` decoded segments.

Methods

reset	Reset states of the turbo decoder object
step	Decode input signal using parallel concatenated decoding scheme

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit and decode using turbo coding

Transmit turbo-encoded blocks of data over a BPSK-modulated AWGN channel. Then, decode using an iterative turbo decoder and display errors.

Define a noise variable, establish a frame length of 256, and use the random stream property so that the results are repeatable.

```
noiseVar = 4; frmLen = 256;
s = RandStream('mt19937ar', 'Seed', 11);
intrlvrIndices = randperm(s, frmLen);
```

Create a Turbo Encoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The `InterleaverIndices` property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
turboEnc = comm.TurboEncoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices);
```

Create a BPSK Modulator System object.

```
bpsk = comm.BPSKModulator;
```

Create an AWGN Channel System object.

```
channel = comm.AWGNChannel('NoiseMethod', 'Variance', 'Variance', ...
      noiseVar);
```

Create a GPU-Based Turbo Decoder System object. The trellis structure for the constituent convolutional code is poly2trellis(4, [13 15 17], 13). The `InterleaverIndicies` property specifies the mapping the object uses to permute the input bits at the encoder as a column vector of integers.

```
turboDec = comm.gpu.TurboDecoder('TrellisStructure', poly2trellis(4, ...
      [13 15 17], 13), 'InterleaverIndices', intrlvrIndices, ...
      'NumIterations', 4);
```

Create an Error Rate System object.

```
errorRate = comm.ErrorRate;
```

Run the simulation.

```
for frmIdx = 1:8
 data = randi(s, [0 1], frmLen, 1);
 encodedData = turboEnc(data);
 modSignal = bpsk(encodedData);
 receivedSignal = channel(modSignal);
```

Convert the received signal to log-likelihood ratios for decoding.

```
receivedBits  = turboDec(-2/(noiseVar/2))*real(receivedSignal));
```

Compare original the data to the received data and then calculate the error rate results.

```
errorStats = errorRate(data,receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\nTotal bits = %d\n', ...
errorStats(1), errorStats(2), errorStats(3))
```

Algorithms

This object implements the inputs and outputs described on the Turbo Decoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

comm.TurboDecoder | comm.TurboEncoder

Introduced in R2012a

reset

System object: `comm.gpu.TurboDecoder`
Package: `comm.gpu`

Reset states of the turbo decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `GPU TurboDecoder` object, `H`.

step

System object: `comm.gpu.TurboDecoder`
Package: `comm.gpu`

Decode input signal using parallel concatenated decoding scheme

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` decodes the input data, `X`, using the parallel concatenated convolutional coding scheme. You specify this scheme using the TrellisStructure and InterleaverIndices properties. It returns the binary decoded data, `Y`. Both `X` and `Y` are column vectors of double-precision data type. When the constituent convolutional code represents a rate 1/*N* code, the step method sets the length of the output vector, `Y`, to (M-2*N_{Tails})/(2*N-1). *M* represents the input vector length and N_{Tails} is given by log2(`TrellisStructure.numStates`)**N*. The output length, `L`, is the same as the length of the interleaver indices.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.gpu.ViterbiDecoder

Package: `comm`

Decode convolutionally encoded data using Viterbi algorithm with GPU

Description

The `GPU ViterbiDecoder` System object decodes input symbols to produce binary output symbols using a graphics processing unit (GPU). This object processes variable-size signals; however, variable-size signals cannot be applied for erasure inputs.

Note To use this object, you must install a Parallel Computing Toolbox license and have access to an appropriate GPU. For more about GPUs, see "GPU Computing" (Parallel Computing Toolbox).

A GPU-based System object accepts typical MATLAB arrays or objects created using the `gpuArray` class. A GPU-based System object supports input signals with double- or single-precision data types. The output signal inherits its data type from the input signal.

- If the input signal is a MATLAB array, the System object handles data transfer between the CPU and the GPU. The output signal is a MATLAB array.
- If the input signal is a `gpuArray`, the data remains on the GPU. The output signal is a `gpuArray`. When the object is given a `gpuArray`, calculations take place entirely on the GPU, and no data transfer occurs. Passing `gpuArray` arguments provides increased performance by reducing simulation time. For more information, see "Establish Arrays on a GPU" (Parallel Computing Toolbox).

To decode input symbols and produce binary output symbols:

1. Define and set up your Viterbi decoder object. See "Construction" on page 3-780.
2. Call `step` to decode input symbols according to the properties of `comm.gpu.ViterbiDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.gpu.ViterbiDecoder` creates a Viterbi decoder System object, `H`. This object uses the Viterbi algorithm to decode convolutionally encoded input data.

`H = comm.gpu.ViterbiDecoder(Name,Value)` creates a Viterbi decoder object, `H`, with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`.

H = comm.gpu.ViterbiDecoder(TRELLIS,Name,Value) creates a Viterbi decoder object, H, with the TrellisStructure property set to TRELLIS, and other specified property Names set to the specified Values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check if a structure is a valid trellis structure. This object supports rate 1/2, 1/3 and 1/4 trellises from simple feedforward encoders. The default value is the result of poly2trellis(7, [171 133]).

InputFormat

Input format

Specify the format of the input to the decoder as one of Unquantized | Hard | Soft. The default is Unquantized.

When you set this property to Unquantized, the input must be a real vector of double or single precision unquantized soft values. The object considers negative numbers to be ones and positive numbers to be zeros. When you set this property to Hard, the input must be a vector of hard decision values, which are zeros or ones. The data type of the inputs can be double precision or single precision. When you set this property to Soft, the input must be a vector of quantized soft values represented as integers between 0 and 2^SoftInputWordLength-1. The data type of the inputs can be double precision or single precision.

SoftInputWordLength

Soft input word length

Specify the number of bits used to represent each quantized soft input value as a positive, integer scalar. This property applies when you set the InputFormat property to Soft. The default is 4 bits.

InvalidQuantizedInputAction

Action when input values are out of range

The only valid setting is Ignore which ignores out of range inputs.

TracebackDepth

Traceback depth

Specify the number of trellis branches used to construct each traceback path as a positive, integer scalar less than or equal to 256. The traceback depth influences the decoding accuracy and delay. The number of zero symbols that precede the first decoded symbol in the output represent a decoding delay. When you set the TerminationMethod property to Continuous, the decoding delay consists of TracebackDepth zero symbols, or TracebackDepth zero bits for a rate 1/N convolutional code. When you set the TerminationMethod property to Truncated or Terminated, there is no output delay and TracebackDepth must be less than or equal to the number of symbols in each input. If the code rate is 1/2, a typical traceback depth value is about five times the constraint length of the code. The default is 34.

TerminationMethod

Termination method of encoded frame

Specify TerminationMethod as one of Continuous | Truncated | Terminated. The default is Continuous. In Continuous mode, the object saves its internal state metric at the end of each frame for use with the next frame. The object treats each traceback path independently. Select Continuous mode when the input signal contains only one symbol. In Truncated mode, the object treats each frame independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. In Terminated mode, the object treats each frame independently, and the traceback path always starts and ends in the all-zeros state.

ResetInputPort

Enable decoder reset input

Set this property to true to enable an additional step method input. When the reset input is a non-zero value, the object resets the internal states of the decoder to initial conditions. This property applies when you set the TerminationMethod property to Continuous. The default is false.

DelayedResetAction

Delay output reset

Delaying the output reset is not supported. The only valid setting is false.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as one of None | Property. The default is None. When you set this property to None the object assumes no puncturing. Set this property to Property to decode punctured codewords based on a puncture pattern vector specified via the PuncturePattern property.

PuncturePattern

Puncture pattern vector

Specify puncture pattern used to puncture the encoded data. The default is [1; 1; 0; 1; 0; 1]. The puncture pattern is a column vector of ones and zeros, where the zeros indicate where to insert dummy bits. The puncture pattern must match the puncture pattern used by the encoder. This property applies when you set the PuncturePatternSource property to Property.

ErasuresInputPort

Enable erasures input

Erasures are not supported. The only valid setting is false.

OutputDataType

Data type of output

The only valid setting is `Full precision` which makes the output data type match the input data type.

`NumFrames`

Number of independent frames present in the input and output data vectors.

Specify the number of independent frames contained in a single data input/output vector. The input vector will be segmented into `NumFrames` segments and decoded independently. The output will contain `NumFrames` decoded segments. The default value of this property is `1`. This property is applies when you set the TerminationMethod is set to `Terminated` or `Truncated`.

Methods

info	Display information about GPU-based Viterbi Decoder object
reset	Reset states of the GPU-based Viterbi Decoder modulator object
step	Decode convolutionally encoded data using Viterbi algorithm

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit a convolutionally encoded 8-DPSK-modulated bit stream through an AWGN channel. Then, demodulate, decode using a Viterbi decoder, and count errors.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.gpu.AWGNChannel('NoiseMethod', ...
            'Signal to noise ratio (SNR)', 'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.gpu.ViterbiDecoder('InputFormat','Hard');
% Delay in bits is TracebackDepth times the number of
% bits per symbol
delay = hDec.TracebackDepth* ...
   log2(hDec.TrellisStructure.numInputSymbols);
hError = comm.ErrorRate( ...
   'ComputationDelay',3,'ReceiveDelay',delay);
for counter = 1:20
   data = randi([0 1],30,1);
   encodedData = step(hConEnc, data);
   modSignal = step(hMod, encodedData);
   receivedSignal = step(hChan, modSignal);
   demodSignal = step(hDemod, receivedSignal);
   receivedBits = step(hDec, demodSignal);
   errorStats = step(hError, data, receivedBits);
end
fprintf('Error rate = %f\nNumber of errors = %d\n', ...
errorStats(1), errorStats(2))
```

References

[1] Fettweis, G., H. Meyr. "Feedforward Architecture for Parallel Viterbi Decoding,” *Journal of VLSI Signal Processing*, Vol. 3, June 1991.

Extended Capabilities

GPU Arrays

Accelerate code by running on a graphics processing unit (GPU) using Parallel Computing Toolbox™.

This System object runs on a GPU, and also supports GPU array inputs. For more information, see “Simulation Acceleration Using GPUs”.

See Also

`comm.ViterbiDecoder`

Introduced in R2012a

info

System object: comm.gpu.ViterbiDecoder
Package: comm

Display information about GPU-based Viterbi Decoder object

Syntax

```
S = info(OBJ)
```

Description

S = info(OBJ) returns a structure, S, containing characteristic information for the System object, OBJ. If OBJ has no characteristic information, S is empty. If OBJ has characteristic information, the fields of S vary depending on OBJ. For object specific details, refer to the help on the infoImpl method of that object.

reset

System object: `comm.gpu.ViterbiDecoder`
Package: `comm`

Reset states of the GPU-based Viterbi Decoder modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the GPU-based `ViterbiDecoder` object, `H`.

step

System object: `comm.gpu.ViterbiDecoder`
Package: `comm`

Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` decodes encoded data, `X`, using the Viterbi algorithm and returns `Y`. `X`, must be a column vector with data type and values that depend on how you set the `InputFormat` property. If the convolutional code uses an alphabet of 2^N possible symbols, the length of the input vector, `X`, must be `L*N` for some positive integer `L`. Similarly, if the decoded data uses an alphabet of 2^K possible output symbols, the length of the output vector, `Y`, is `L*K`.

`Y = step(H,X,R)` resets the internal states of the decoder when you input a non-zero reset signal, `R`. `R` must be a double precision, single precision or logical scalar. This syntax applies when you set the `TerminationMethod` property to `Continuous` and the `ResetInputPort` property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

gsmDownlinkConfig

Create GSM downlink TDMA frame configuration object

Description

The `gsmDownlinkConfig` object is a GSM downlink TDMA frame configuration object. Use `gsmDownlinkConfig` objects to create GSM downlink waveforms.

Creation

Syntax

```
cfggsmdl = gsmDownlinkConfig
cfggsmdl = gsmDownlinkConfig(sps)
cfggsmdl = gsmDownlinkConfig( ___ ,Name,Value)
```

Description

`cfggsmdl = gsmDownlinkConfig` creates a GSM downlink TDMA frame configuration object.

`cfggsmdl = gsmDownlinkConfig(sps)` sets the `SamplesPerSymbol` property to `sps`.

`cfggsmdl = gsmDownlinkConfig( ___ ,Name,Value)` sets one or more name-value pair arguments using any of the previous syntaxes. For example, `'RiseTime',4` sets the burst rise time to 4 symbols. Enclose each property in quotes. Specify name-value pair arguments after all other input arguments.

Properties

`SamplesPerSymbol` — Samples per symbol
`16` (default) | positive integer multiple of 4

Samples per symbol, specified as a positive integer multiple of 4.

Data Types: `double`

`BurstType` — Burst types
`["NB" "NB" "NB" "NB" "NB" "NB" "NB" "NB"]` (default) | string row vector with 8 elements | `"NB"` | `"FB"` | `"SB"` | `"Dummy"` | `"Off"`

Burst types for time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector where each value is `"NB"`, `"FB"`, `"SB"`, `"Dummy"`, or `"Off"` — Each element specifies the burst type for the corresponding time slot.
- `"NB"` — Transmit data using a normal burst for every time slot.
- `"FB"` — Transmit data using a frequency correction burst for every time slot.

- `"SB"` — Transmit data using a time synchronization burst for every time slot.
- `"Dummy"` — Transmit data using a dummy burst for every time slot.
- `"Off"` — All eight time slots contain no data.

For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-801.

Note The `BurstType` property is an enumeration. To perform code generation, see “Code Generation” on page 3-822 and the “MEX Generation for GSM Downlink Waveform” on page 3-796 example.

Example: `["NB" "AB" "AB" "NB" "Off" "NB" "AB" "Off"]` configures the frame to use normal bursts in time slots 0, 3, and 5, use access bursts in time slots 1, 2, and 6, and transmit no data in time slots 4 and 7.

TSC — Training sequence code

`[0 1 2 3 4 5 6 7]` (default) | eight-element row vector | integer in the range [0, 7]

Training sequence code (TSC) for normal bursts in time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector of integers in the range [0, 7] — Each element specifies the TSC value for the corresponding normal burst time slot.
- Integer in the range [0, 7] — Specifies the TSC value for every normal burst time slot.

For more information, see “Training Sequence Code (TSC)” on page 3-803.

Example: `[5 7 0 0 0 0 0 0]` configures the frame to use training sequence `5` in time slot 0, training sequence `7` in time slot 1, and training sequence `0` in time slots 2 through 7.

Dependencies

To enable this property for a time slot, set the corresponding element of `BurstType` to `"NB"`.

Data Types: `double`

`Attenuation` — Power attenuation

`[0 0 0 0 0 0 0 0]` (default) | eight-element row vector | nonnegative integer

Power attenuation in dB for time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector of nonnegative integers — Each element specifies the attenuation power value for the corresponding time slot.
- Nonnegative integer — Specifies the power attenuation value for every time slot.

Example: `[0 0 0 0 0 0 0 3]` configures the frame to apply 0 dB attenuation to the burst signal power in time slot 0 through 6 and 3 dB of attenuation to the burst signal power in time slot 7.

Data Types: `double`

`RiseTime` — Burst rise time

2 (default) | positive scalar

Burst rise time in symbols, specified as a positive scalar in the range [1/`SamplesPerSymbol`, 29], where the increment resolution is 1/`SamplesPerSymbol`. The total ramp-up and ramp-down duration

(RiseTime - RiseDelay + FallTime + FallDelay) must be less than 9.25 symbols. The characteristic shape of the rising edge of the burst is sinusoidal.

For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-801.

Data Types: double

RiseDelay — Burst rise delay
0 (default) | positive scalar

Burst rise delay in symbols, specified as a positive scalar in the range [–10, 10], where the increment resolution is 1/SamplesPerSymbol. The total ramp-up and ramp-down duration (RiseTime - RiseDelay + FallTime + FallDelay) must be less than 9.25 symbols. The burst rise delay is measured with respect to the start of the useful part of the burst. For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-801.

When the burst rise delay is 0, the burst reaches full amplitude at the start of the useful part of the burst. When the burst rise delay is positive, the burst reaches full amplitude RiseDelay symbols after the start of the useful part. When the burst rise delay is negative, the burst reaches full amplitude RiseDelay symbols before the start of the useful part.

Data Types: double

FallTime — Burst fall time
2 (default) | positive scalar

Burst fall time in symbols, specified as a positive scalar in the range [1/SamplesPerSymbol, 29], where the increment resolution is 1/SamplesPerSymbol. The total ramp-up and ramp-down duration (RiseTime - RiseDelay + FallTime + FallDelay) must be less than 9.25 symbols. The characteristic shape of the falling edge of the burst is sinusoidal.

For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-801.

Data Types: double

FallDelay — Burst fall delay
0 (default) | positive scalar

Burst fall delay in symbols, specified as a positive scalar in the range [–10, 10], where the increment resolution is 1/SamplesPerSymbol. The total ramp-up and ramp-down duration (RiseTime - RiseDelay + FallTime + FallDelay) must be less than 9.25 symbols. The burst fall delay is measured with respect to the end of the useful part of the burst. For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-801.

When the burst fall delay is 0, the burst begins decreasing from full amplitude at the end of the useful part of the burst. When the burst fall delay is positive, the burst begins decreasing from full amplitude FallDelay symbols after the end of the useful part. When the burst fall delay is negative, the burst begins decreasing from full amplitude FallDelay symbols before the end of the useful part.

Data Types: double

Examples

Create GSM Downlink Waveform

Create a GSM downlink TDMA frame configuration object with default settings, and then create a GSM waveform containing one TDMA frame. The GSM TDMA frame has eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM downlink TDMA frame configuration object with default settings.

```
cfggsmdl = gsmDownlinkConfig
```

```
cfggsmdl = 
  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmDownlinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

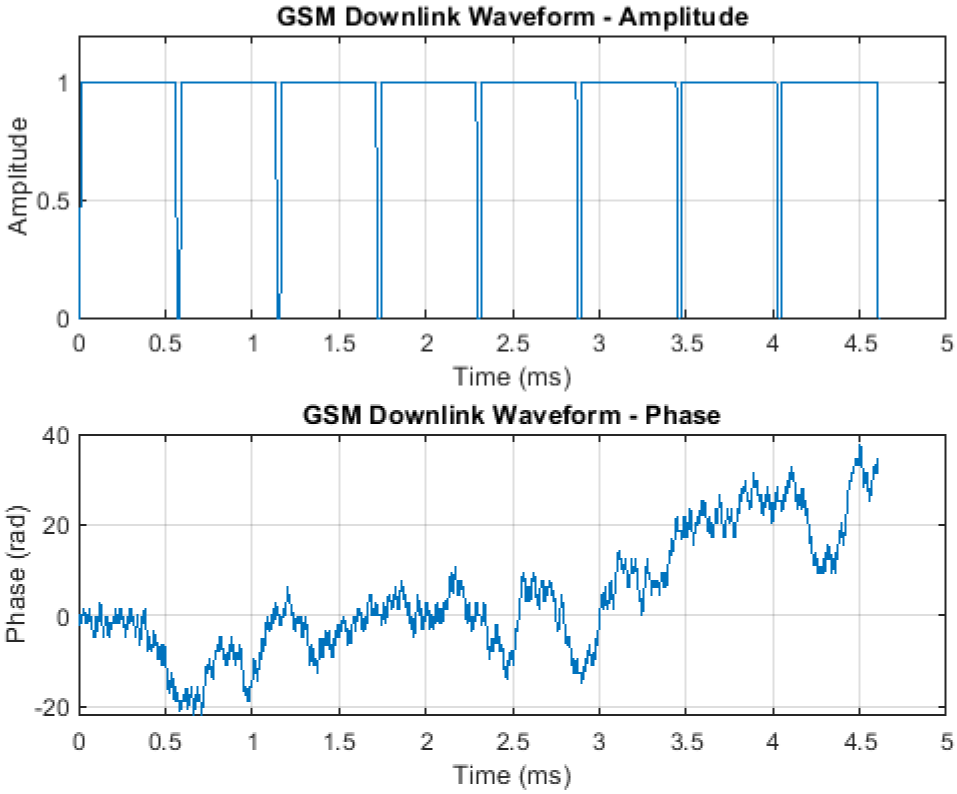
```
wfInfo = gsmInfo(cfggsmdl)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmdl);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Downlink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Downlink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Create GSM Downlink Waveform with Specified Samples per Symbol

Create a GSM downlink TDMA frame configuration object that specifies 8 samples per symbol, and then create a GSM waveform containing one GSM downlink TDMA frame. The GSM TDMA frame are eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms separates each time slot. Plot the GSM waveform.

Create a GSM downlink TDMA frame configuration object, specifying 8 samples per symbols.

```
sps = 8;
cfggsmdl = gsmDownlinkConfig(sps)
```

```
cfggsmdl = 
  gsmDownlinkConfig with properties:

          BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 8
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmDownlinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

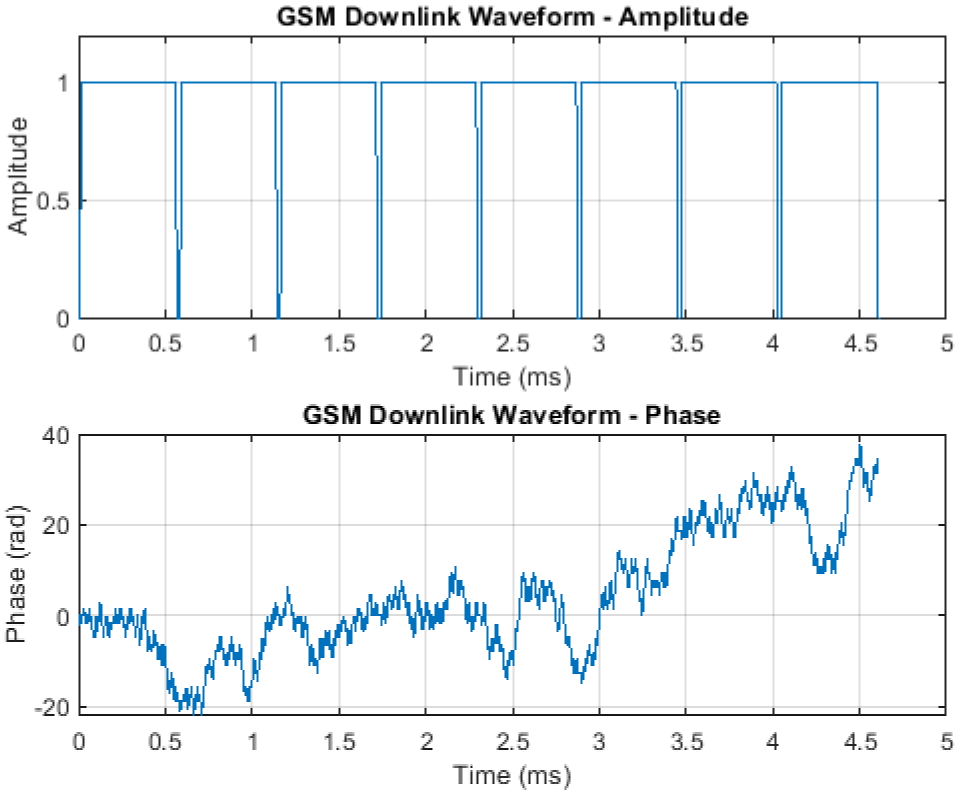
```
wfInfo = gsmInfo(cfggsmdl)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 2.1667e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 1250
    FrameLengthInSamples: 10000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmdl);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Downlink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Downlink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Create GSM Downlink Waveform with Specified Attenuation

Create two GSM downlink TDMA frame configuration objects. Specify default settings for the first `gsmDownlinkConfig` object and adjust the signal power per time slot for the second. Generate GSM waveforms for both configurations. Plot the waveforms to show the signal attenuation per time slot in the second waveform.

Create a GSM downlink TDMA frame configuration object with default settings.

```
cfggsmdl = gsmDownlinkConfig
```

```
cfggsmdl = 
  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Create another GSM downlink TDMA frame configuration object, adjusting the signal attenuation settings per time slot.

```
cfggsmdl2 = gsmDownlinkConfig('Attenuation',[1 0 3 4 5 6 4 2])
```

```
cfggsmdl2 = 
  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [1 0 3 4 5 6 4 2]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmDownlinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

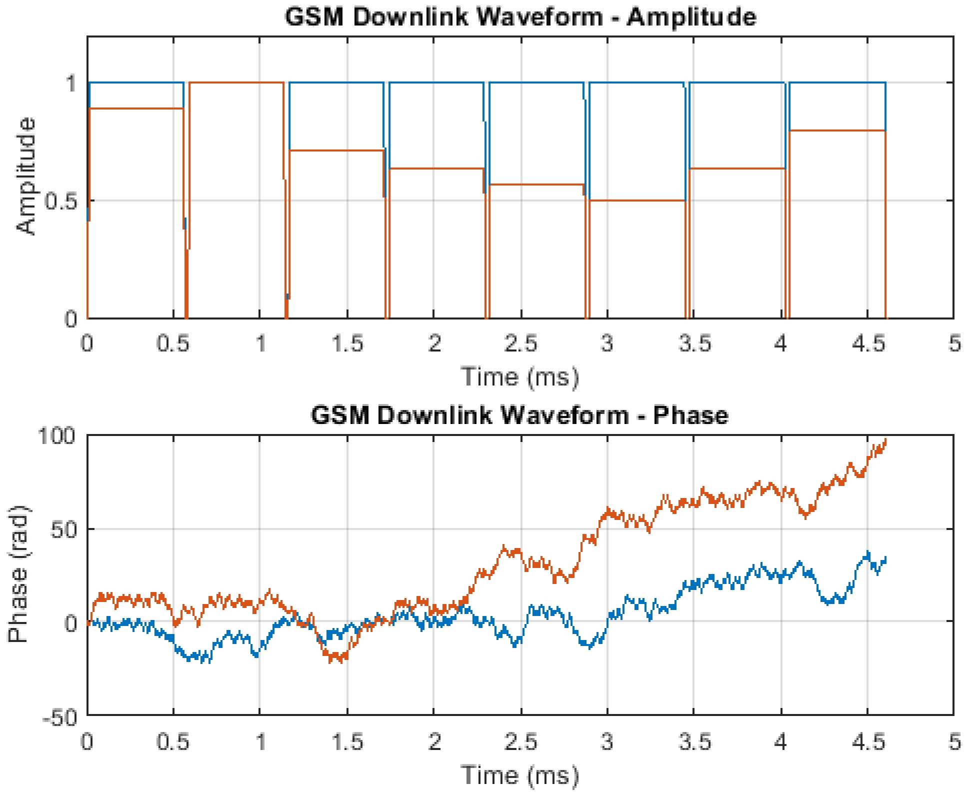
```
wfInfo = gsmInfo(cfggsmdl)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveforms, containing one TDMA frame, by using the `gsmFrame` function. GSM TDMA frames have are eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms separates each time slot. Plot each GSM waveform.

```
waveform = gsmFrame(cfggsmdl);
waveform2 = gsmFrame(cfggsmdl2);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,[abs(waveform),abs(waveform2)])
grid on
axis([0 5 0 1.2])
title('GSM Downlink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,[unwrap(angle(waveform)),unwrap(angle(waveform2))])
grid on
title('GSM Downlink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



MEX Generation for GSM Downlink Waveform

Generate and run a GSM waveform MEX function from the helper function `createDownlinkWaveform`. The `createDownlinkWaveform` helper function creates a GSM downlink waveform.

Write MATLAB Function

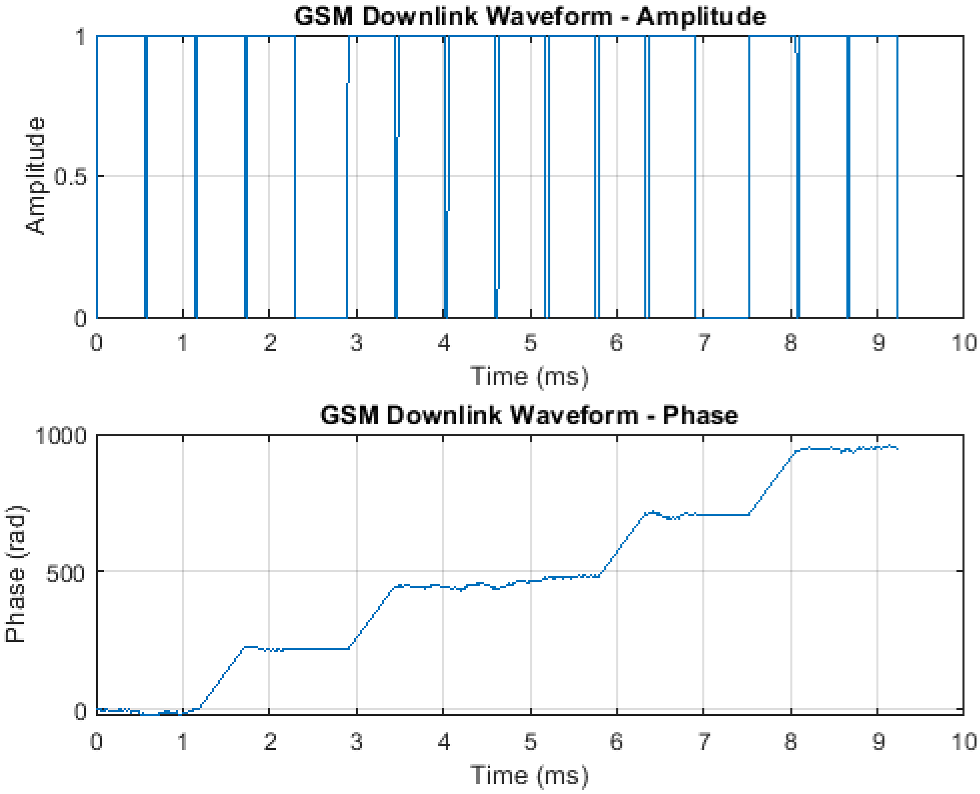
Open createDownlinkWaveform.m to see the code. The `createDownlinkWaveform` helper function generates a GSM downlink waveform by using the `gsmDownlinkConfig` object and the `gsmInfo` and `gsmFrame` functions.

Generate GSM Waveform

Use the `createDownlinkWaveform` helper function to create a GSM waveform containing two TDMA frames, and then plot the waveform.

```
[x,t] = createDownlinkWaveform(2);

figure
subplot(2,1,1);
plot(t,abs(x));
grid on;
title('GSM Downlink Waveform - Amplitude');
```

```
xlabel('Time (ms)');
ylabel('Amplitude')
subplot(2,1,2);
plot(t,unwrap(angle(x)));
grid on;
title('GSM Downlink Waveform - Phase');
xlabel('Time (ms)');
ylabel('Phase (rad)')
```



Generate MEX Function

Code generation defaults to MEX code generation when you do not specify a build target. By default, `codegen` names the generated MEX function `createDownlinkWaveform_mex`. Generate a MEX function from the `createDownlinkWaveform` helper function, and then run the MEX function to create two TDMA frames.

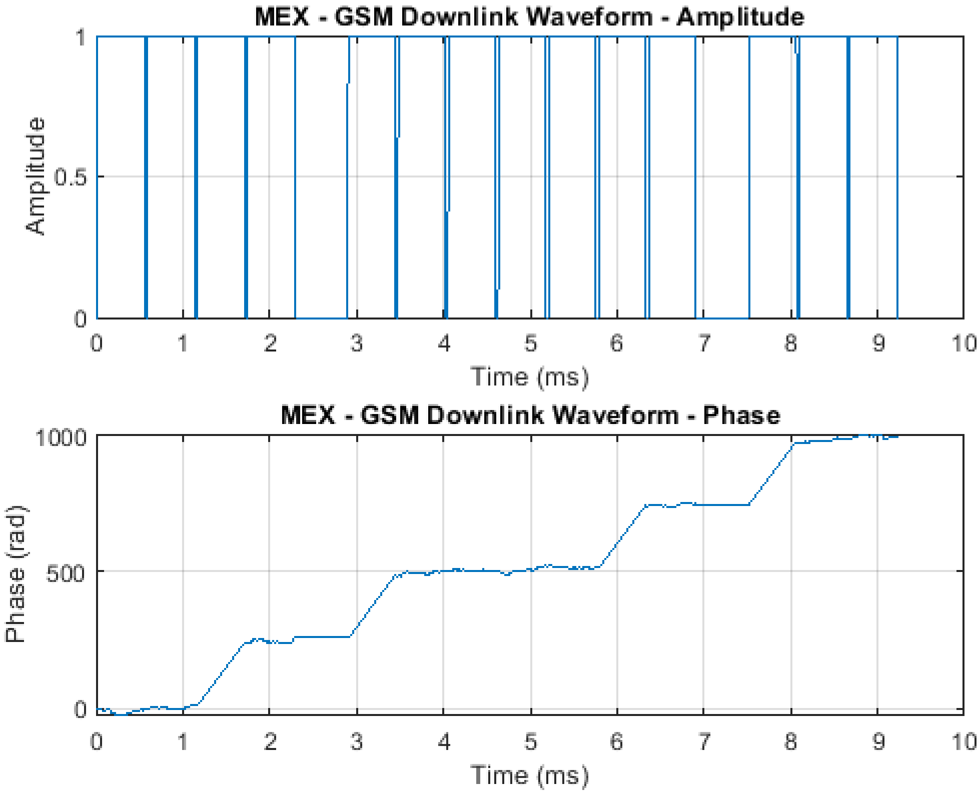
```
codegen createDownlinkWaveform -args 3
```

Generate Waveform Using MEX Function

Run the MEX function and plot the results. Since the waveform is created using random data, the phase plot changes each time you run the `generateDownlinkFrame` helper function or `createDownlinkWaveform_mex` function.

```
[x,t] = createDownlinkWaveform_mex(2);

figure
```

```
subplot(2,1,1);
plot(t,abs(x));
grid on;
title('MEX - GSM Downlink Waveform - Amplitude');
xlabel('Time (ms)');
ylabel('Amplitude')
subplot(2,1,2);
plot(t,unwrap(angle(x)));
grid on;
title('MEX - GSM Downlink Waveform - Phase');
xlabel('Time (ms)');
ylabel('Phase (rad)')
```

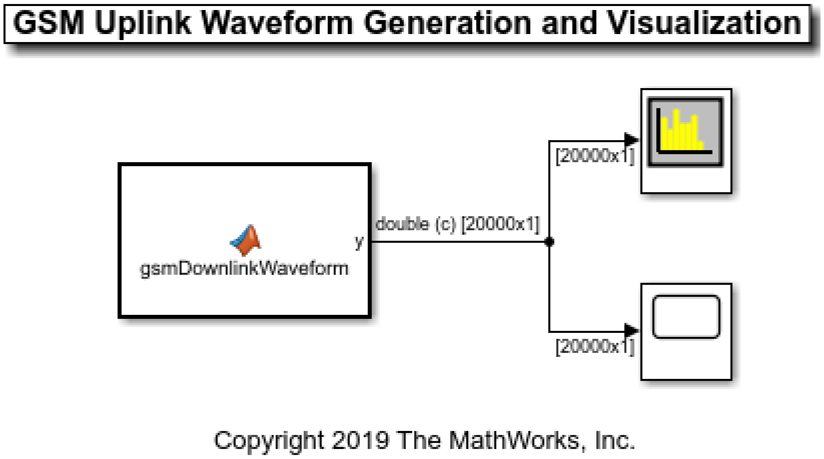


GSM Downlink Waveform Generation in Simulink

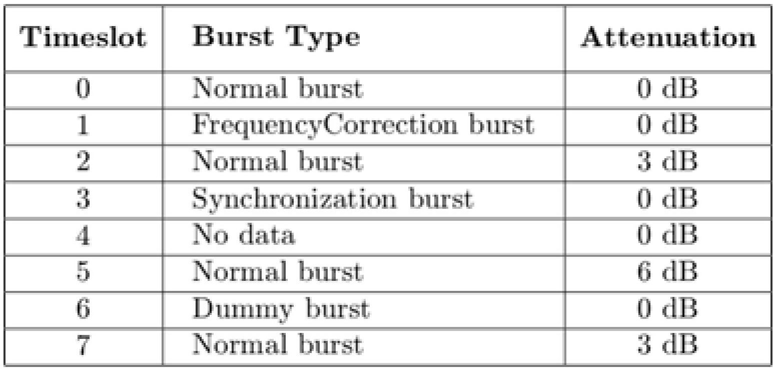
Model a GSM® waveform generator in Simulink® by using the MATLAB Function block and Communications Toolbox™ functions.

GSM Downlink Waveform Generation

The MATLAB Function (Simulink) block contains the `gsmDownlinkWaveform` function code. The code in the MATLAB Function block creates a GSM waveform by using the `gsmDownlinkConfig` object and the `gsmFrame` function.


GSM Uplink Waveform Generation and Visualization
gsmDownlinkWaveform
y
double (c) [20000x1]
[20000x1]
[20000x1]
Copyright 2019 The MathWorks, Inc.

The `gsmDownlinkConfig` object specifies 16 samples per symbol and the time slot configuration for the GSM downlink TDMA frame shown is this table.



Timeslot	Burst Type	Attenuation
0	Normal burst	0 dB
1	FrequencyCorrection burst	0 dB
2	Normal burst	3 dB
3	Synchronization burst	0 dB
4	No data	0 dB
5	Normal burst	6 dB
6	Dummy burst	0 dB
7	Normal burst	3 dB

The output waveform has 16 samples for each GMSK symbol. The `gsmFrame` function generates the samples of the waveform.

Explore the Model

In compliance with GSM standards 3GPP TS 45.001 and 3GPP TS 45.002, the sample time of the MATLAB Function block that contains the `gsmDownlinkWaveform` function code is set to the GSM symbol rate of 1625e3/6 symbols per second. Display the current `gsmDownlinkConfig` object settings by using the `gsmInfo` function.

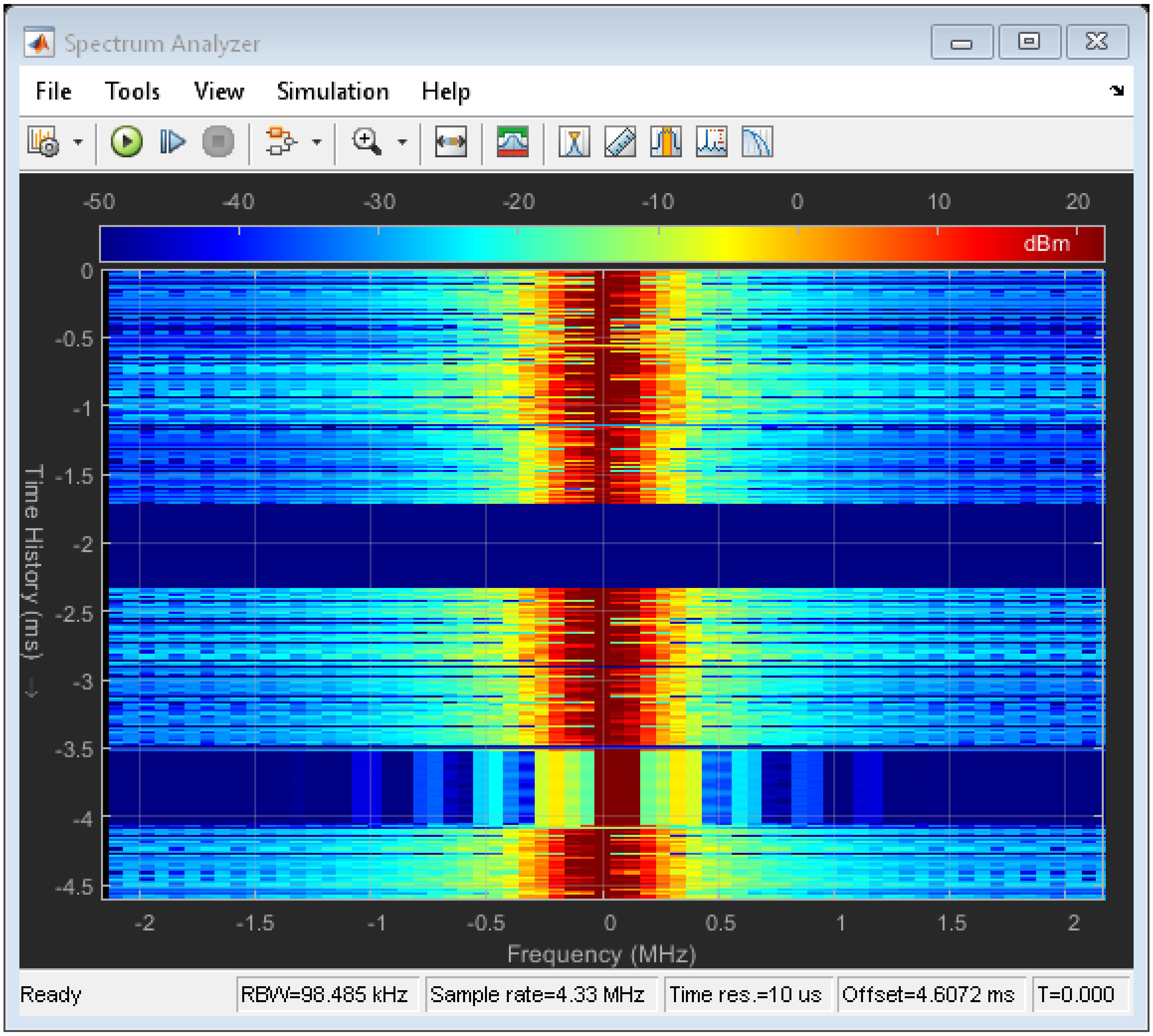
```
wfInfo = 

  struct with fields:

              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

The model sample time of the MATLAB Function (Simulink) block is set to `wfInfo.FrameLengthInSamples/wfInfo.SampleRate`. To view the **Sample time** parameter, open the Block Parameters dialog box by right-clicking the MATLAB Function block and selecting **Block Parameters (Subsystem)**.

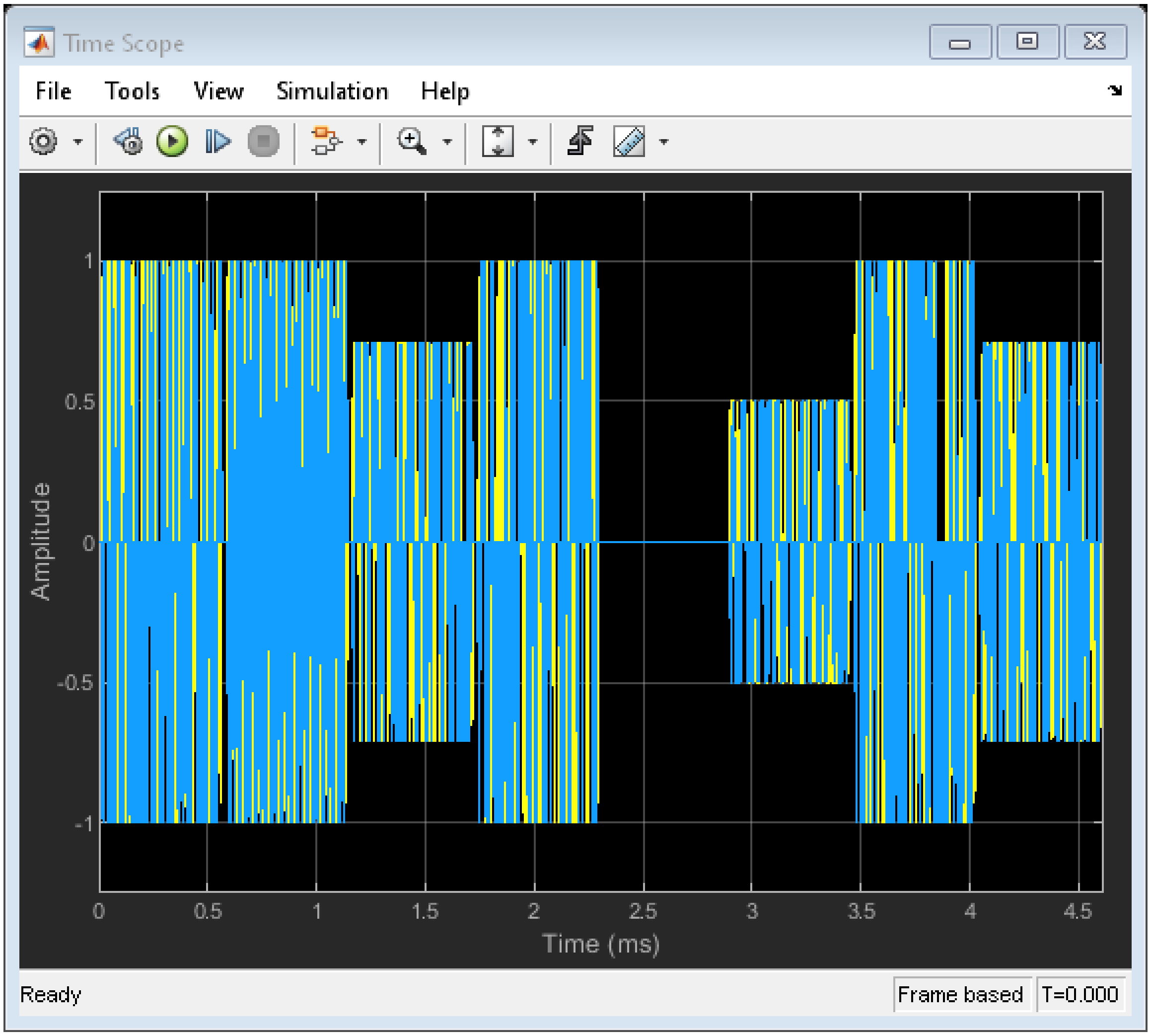
Before the simulation runs, you must configure the sample rate of the MATLAB Function block. The `PreLoadFcn` and `InitFcn` callback functions configure the MATLAB Function block by creating a `gsmDownlinkConfig` object and `wfInfo` structure. To view the callback functions, on the **Modeling** tab, in the **Setup** section, select **Model Settings** > **Model Properties**. Then, on the **Callbacks** tab, select the `PreLoadFcn` or `InitFcn` callback function in the **Model callbacks** pane.

Results

Display the time domain signal and the spectrogram by running the simulation.



RBW=98.485 kHz | Sample rate=4.33 MHz | Time res.=10 us | Offset=4.6072 ms | T=0.000

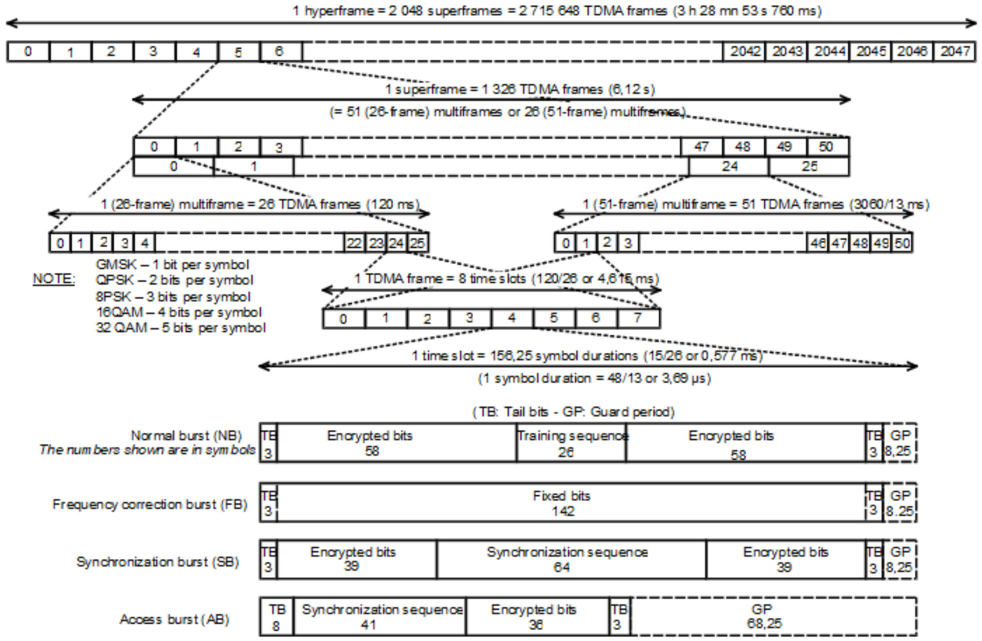


More About

GSM Frames, Time Slots, and Bursts

In GSM, transmissions consist of TDMA frames. Each GSM TDMA frame consists of eight time slots. The transmission data content of a time slot is called a burst. As described in Section 5.2 of 3GPP TS 45.011, a GSM time slot has a 156.25-symbol duration when using the normal symbol period, which is a time interval of 15/26 ms or about 576.9 microseconds. A guard period of 8.25 symbols or about 30.46 microseconds separates each time slot. The GSM standards describes a symbol as one bit period. Since GSM uses GMSK modulation, there is one bit per bit period. The transmission timing of a burst within a time slot is defined in terms of the bit number (BN). The BN refers to a particular bit period within a time slot. The bit with the lowest BN is transmitted first. BN0 is the first bit period, and BN156 is the last quarter-bit period.

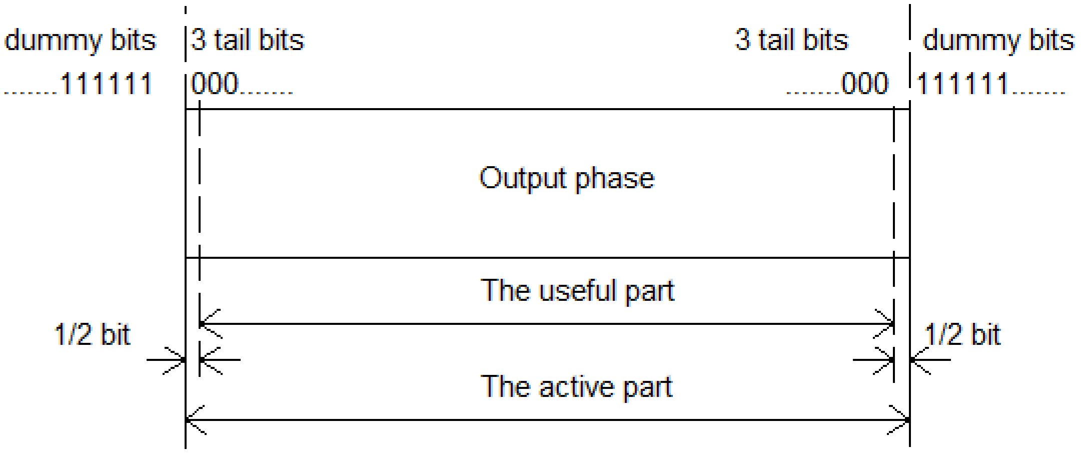
This image from 3GPP TS 45.011 shows the relationship between different frame types and the relationship between different burst types.


1 hyperframe = 2 048 superframes = 2 715 648 TDMA frames (3 h 28 mn 53 s 760 ms)
0
1
2
3
4
5
6
2042
2043
2044
2045
2046
2047
1 superframe = 1 326 TDMA frames (6,12 s)
(= 51 (26-frame) multiframes or 26 (51-frame) multiframes)
0
1
2
3
47
48
49
50
0
1
24
25
1 (26-frame) multiframe = 26 TDMA frames (120 ms)
0
1
2
3
4
22
23
24
25
1 (51-frame) multiframe = 51 TDMA frames (3060/13 ms)
0
1
2
3
46
47
48
49
50
NOTE:
GMSK – 1 bit per symbol
QPSK – 2 bits per symbol
8PSK – 3 bits per symbol
16QAM – 4 bits per symbol
32 QAM – 5 bits per symbol
1 TDMA frame = 8 time slots (120/26 or 4,615 ms)
0
1
2
3
4
5
6
7
1 time slot = 156,25 symbol durations (15/26 or 0,577 ms)
(1 symbol duration = 48/13 or 3,69 µs)
(TB: Tail bits - GP: Guard period)
Normal burst (NB)
The numbers shown are in symbols
TB 3
Encrypted bits 58
Training sequence 26
Encrypted bits 58
TB 3
GP 8,25
Frequency correction burst (FB)
TB 3
Fixed bits 142
TB 3
GP 8,25
Synchronization burst (SB)
TB 3
Encrypted bits 39
Synchronization sequence 64
Encrypted bits 39
TB 3
GP 8,25
Access burst (AB)
TB 8
Synchronization sequence 41
Encrypted bits 36
TB 3
GP 68,25

This table shows the supported burst types and their characteristics.

Burst Type	Description	Link Direction	Useful Duration
NB	Normal burst	Uplink/Downlink	147
FB	Frequency correction burst	Downlink	147
SB	Synchronization burst	Downlink	147
Dummy	Dummy burst	Downlink	147
AB	Access burst	Uplink	87
Off	No burst sent	Uplink/Downlink	0

Useful duration, described in Section 5.2.2 of 3GPP TS 45.002, is a characteristic of GSM bursts. The useful duration, or useful part, of a burst is defined as beginning halfway through BN0 and ending half a bit period before the start of the guard period. The guard period is the period between bursts in successive time slots. This figure, from Section 2.2 of 3GPP TS 45.004, shows the leading and trailing ½ bit difference between the useful and active parts of the burst.



For more information, see “GSM TDMA Frame Parameterization for Waveform Generation”.

Training Sequence Code (TSC)

Normal bursts include a training sequence bits field assigned a bit pattern based on the specified TSC. For GSM, you can select one of these eight training sequences for normal burst type time slots.

Training Sequence Code (TSC)	Training Sequence Bits (BN61, BN62, …, BN86)
0	(0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,1,1)
1	(0,0,1,0,1,1,0,1,1,1,0,1,1,1,1,0,0,0,1,0,1,1,0,1,1,1)
2	(0,1,0,0,0,0,1,1,1,0,1,1,1,0,1,0,0,1,0,0,0,0,1,1,1,0)
3	(0,1,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,1,1,1,0)
4	(0,0,0,1,1,0,1,0,1,1,1,0,0,1,0,0,0,0,0,1,1,0,1,0,1,1)
5	(0,1,0,0,1,1,1,0,1,0,1,1,0,0,0,0,0,1,0,0,1,1,1,0,1,0)
6	(1,0,1,0,0,1,1,1,1,1,0,1,1,0,0,0,1,0,1,0,0,1,1,1,1,1)
7	(1,1,1,0,1,1,1,1,0,0,0,1,0,0,1,0,1,1,1,0,1,1,1,1,0,0)

For more information, see Section 5.2.3 in 3GPP TS 45.002.

References

[1] 3GPP TS 45.001. "GSM/EDGE Physical layer on the radio path. General description." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] 3GPP TS 45.002. "GSM/EDGE Multiplexing and multiple access on the radio path." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[3] 3GPP TS 45.004. "GSM/EDGE Modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `SamplesPerSymbol`, `RiseTime`, `RiseDelay`, `FallTime`, and `FallDelay` properties must be set when creating the object, and their settings are static in the generated code.
- The `BurstType` [property must be set using the enumeration type instead of the string representation. Use these `gsmDownlinkBurstType` enumerations: `gsmDownlinkBurstType.NB`, `gsmDownlinkBurstType.FB`, `gsmDownlinkBurstType.SB`, `gsmDownlinkBurstType.Dummy`, and `gsmUplinkBurstType.Off`. For example, this code assigns a frequency correction burst in time slot 2 and 5.

```
cfg = gsmDownlinkConfig
```

```
cfg = 

  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

```
cfg.BurstType([2 5] +1) = gsmDownlinkBurstType.FB
```

```
cfg = 

  gsmDownlinkConfig with properties:

           BurstType: [NB    NB    FB    NB    NB    FB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

See Also

Objects

`gsmUplinkConfig`

Functions

`gsmCheckTimeMask` | `gsmFrame` | `gsmInfo`

Topics

"GSM TDMA Frame Parameterization for Waveform Generation"

Introduced in R2019b

gsmUplinkConfig

Create GSM uplink TDMA frame configuration object

Description

The `gsmUplinkConfig` object is a GSM uplink TDMA frame configuration object. Use `gsmUplinkConfig` objects to create GSM uplink waveforms.

Creation

Syntax

```
cfggsmul = gsmUplinkConfig
cfggsmul = gsmUplinkConfig(sps)
cfggsmul = gsmUplinkConfig(___,Name,Value)
```

Description

`cfggsmul = gsmUplinkConfig` creates a GSM uplink TDMA frame configuration object.

`cfggsmul = gsmUplinkConfig(sps)` sets the `SamplesPerSymbol` property to `sps`.

`cfggsmul = gsmUplinkConfig(___,Name,Value)` sets one or more name-value pair arguments using any of the previous syntaxes. For example, `'RiseTime',4` sets the burst rise time to `4` symbols. Enclose each property in quotes. Specify name-value pair arguments after all other input arguments.

Properties

`SamplesPerSymbol` — Samples per symbol
`16` (default) | positive integer multiple of 4

Samples per symbol, specified as a positive integer multiple of 4.

Data Types: `double`

`BurstType` — Burst types
`["NB" "NB" "NB" "NB" "NB" "NB" "NB" "NB"]` (default) | string row vector with 8 elements | `"NB"` | `"AB"` | `"Off"`

Burst types for time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector where each value is `"NB"`, `"AB"`, or `"Off"` — Each element specifies the burst type for the corresponding time slot.
- `"NB"` — Transmit data using a normal burst for every time slot.
- `"AB"` — Transmit data using an access burst for every time slot.

- "Off" — All eight time slots contain no data.

For more information, see "GSM Frames, Time Slots, and Bursts" on page 3-819

Note The BurstType property is an enumeration. To perform code generation, see "Code Generation" on page 3-822 and the "MEX Generation for GSM Uplink Waveform" on page 3-814 example.

Example: ["NB" "AB" "AB" "NB" "Off" "NB" "AB" "Off"] configures the frame to use normal bursts in time slots 0, 3, and 5, use access bursts in time slots 1, 2, and 6, and transmit no data in time slots 4 and 7.

TSC — Training sequence code

[0 1 2 3 4 5 6 7] (default) | eight-element row vector | integer in the range [0, 7]

Training sequence code (TSC) for normal bursts in time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector of integers in the range [0, 7] — Each element specifies the TSC value for the corresponding normal burst time slot.
- Integer in the range [0, 7] — Specifies the TSC value for every normal burst time slot.

For more information, see "Training Sequence Code (TSC)" on page 3-821.

Example: [5 7 0 0 0 0 0 0] configures the frame to use training sequence 5 in time slot 0, training sequence 7 in time slot 1, and training sequence 0 in time slots 2 through 7.

Dependencies

To enable this property for a time slot, set the corresponding element of BurstType to "NB".

Data Types: double

Attenuation — Power attenuation

[0 0 0 0 0 0 0 0] (default) | eight-element row vector | nonnegative integer

Power attenuation in dB for time slots 0–7 in the TDMA frame, specified as one of these options:

- Eight-element row vector of nonnegative integers — Each element specifies the attenuation power value for the corresponding time slot.
- Nonnegative integer — Specifies the power attenuation value for every time slot.

Example: [0 0 0 0 0 0 0 3] configures the frame to apply 0 dB of attenuation to the burst signal power in time slot 0 through 6 and 3 dB of attenuation to the burst signal power in time slot 7.

Data Types: double

RiseTime — Burst rise time

2 (default) | positive scalar

Burst rise time in symbols, specified as a positive scalar in the range [1/SamplesPerSymbol, 29], where the increment resolution is 1/SamplesPerSymbol. The total ramp-up and ramp-down duration (RiseTime - RiseDelay + FallTime + FallDelay) must be less than 9.25 symbols. The characteristic shape of the rising edge of the burst is sinusoidal.

For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-819.

Data Types: `double`

`RiseDelay` — Burst rise delay

`0` (default) | positive scalar

Burst rise delay in symbols, specified as a positive scalar in the range [–10, 10], where the increment resolution is 1/`SamplesPerSymbol`. The total ramp-up and ramp-down duration (`RiseTime` - `RiseDelay` + `FallTime` + `FallDelay`) must be less than 9.25 symbols. The burst rise delay is measured with respect to the start of the useful part of the burst. For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-819.

When the burst rise delay is `0`, the burst reaches full amplitude at the start of the useful part of the burst. When the burst rise delay is positive, the burst reaches full amplitude `RiseDelay` symbols after the start of the useful part. When the burst rise delay is negative, the burst reaches full amplitude `RiseDelay` symbols before the start of the useful part.

Data Types: `double`

`FallTime` — Burst fall time

`2` (default) | positive scalar

Burst fall time in symbols, specified as a positive scalar in the range [1/`SamplesPerSymbol`, 29], where the increment resolution is 1/`SamplesPerSymbol`. The total ramp-up and ramp-down duration (`RiseTime` - `RiseDelay` + `FallTime` + `FallDelay`) must be less than 9.25 symbols. The characteristic shape of the falling edge of the burst is sinusoidal.

For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-819.

Data Types: `double`

`FallDelay` — Burst fall delay

`0` (default) | positive scalar

Burst fall delay in symbols, specified as a positive scalar in the range [–10, 10], where the increment resolution is 1/`SamplesPerSymbol`. The total ramp-up and ramp-down duration (`RiseTime` - `RiseDelay` + `FallTime` + `FallDelay`) must be less than 9.25 symbols. The burst fall delay is measured with respect to the end of the useful part of the burst. For more information, see “GSM Frames, Time Slots, and Bursts” on page 3-819.

When the burst fall delay is `0`, the burst begins decreasing from full amplitude at the end of the useful part of the burst. When the burst fall delay is positive, the burst begins decreasing from full amplitude `FallDelay` symbols after the end of the useful part. When the burst fall delay is negative, the burst begins decreasing from full amplitude `FallDelay` symbols before the end of the useful part.

Data Types: `double`

Examples

Create GSM Uplink Waveform

Create a GSM uplink TDMA frame configuration object with default settings, and then create a GSM waveform containing one TDMA frame. GSM TDMA frames have eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot the GSM waveform.

Create a GSM uplink TDMA frame configuration object with default settings.

```
cfggsmul = gsmUplinkConfig
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

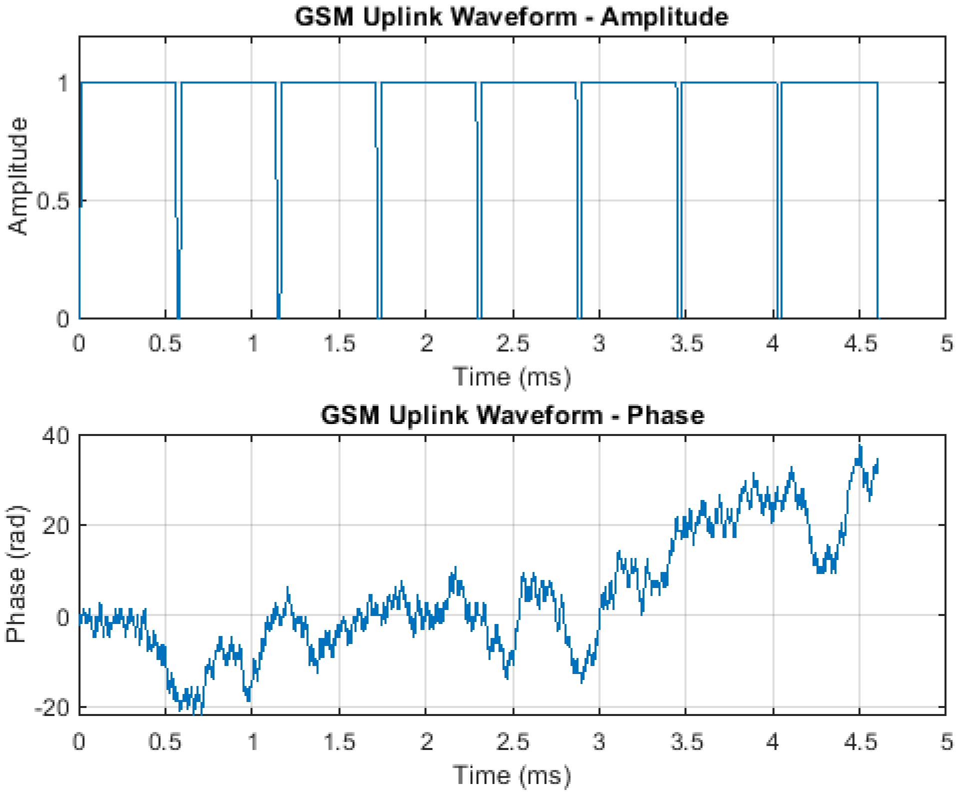
```
wfInfo = gsmInfo(cfggsmul)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmul);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Uplink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Create GSM Uplink Waveform with Specified Samples per Symbol

Create a GSM uplink TDMA frame configuration object that specifies 4 samples per symbol, and then create a GSM waveform containing one GSM downlink TDMA frame. The GSM TDMA frame are eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms separates each time slot. Plot the GSM waveform.

Create a GSM uplink TDMA frame configuration object, specifying 4 samples per symbols.

```
sps = 4;
cfggsmul = gsmUplinkConfig(sps)
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
      SamplesPerSymbol: 4
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

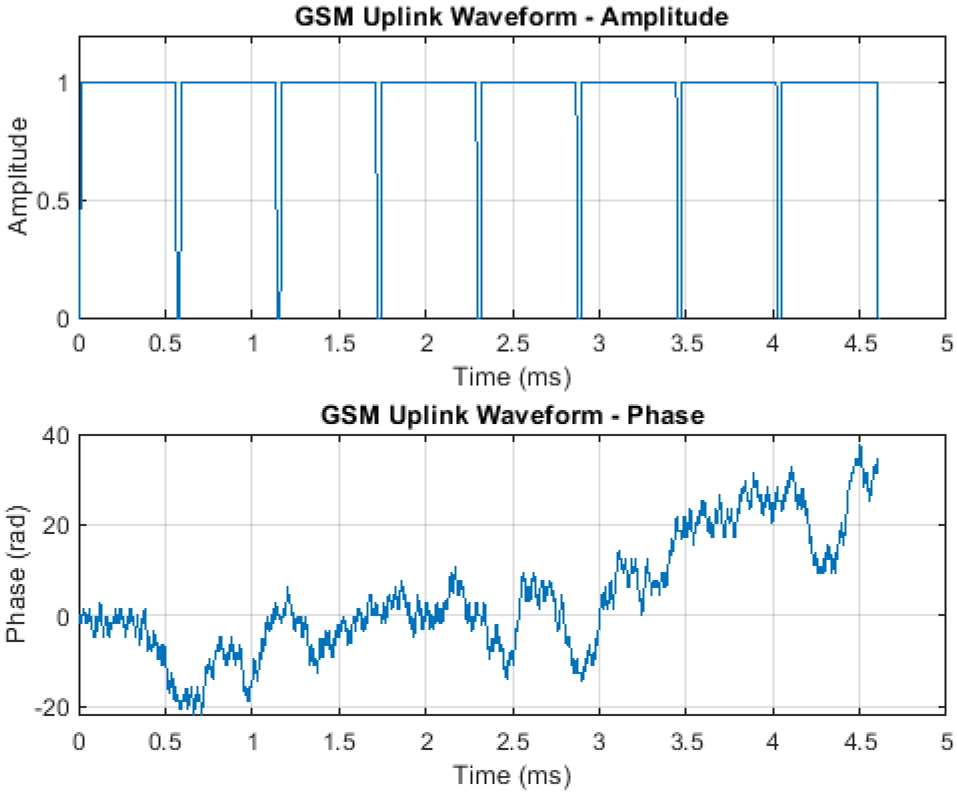
```
wfInfo = gsmInfo(cfggsmul)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 1.0833e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 625
    FrameLengthInSamples: 5000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveform by using the `gsmFrame` function, and then plot the GSM waveform.

```
waveform = gsmFrame(cfggsmul);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,abs(waveform))
grid on
axis([0 5 0 1.2])
title('GSM Uplink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,unwrap(angle(waveform)))
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



Create GSM Uplink Waveform with Specified Attenuation

Create two GSM uplink TDMA frame configuration objects. Specify default settings for the first `gsmUplinkConfig` object and adjust the signal power per time slot for the second. Generate GSM waveforms for both configurations. Plot the waveforms to show the signal attenuation per time slot in the second waveform.

Create a GSM uplink TDMA frame configuration object with default settings.

```
cfggsmul = gsmUplinkConfig
```

```
cfggsmul = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Create another GSM uplink TDMA frame configuration object, adjusting the signal attenuation settings per time slot.

```
cfggsmul2 = gsmUplinkConfig('Attenuation',[1 2 3 4 5 4 3 2])
```

```
cfggsmul2 = 
  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [1 2 3 4 5 4 3 2]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

Display information about the configured `gsmUplinkConfig` object by using the `gsmInfo` function. Assign the sample rate to a variable, `Rs`, for use in computing the plot timescale.

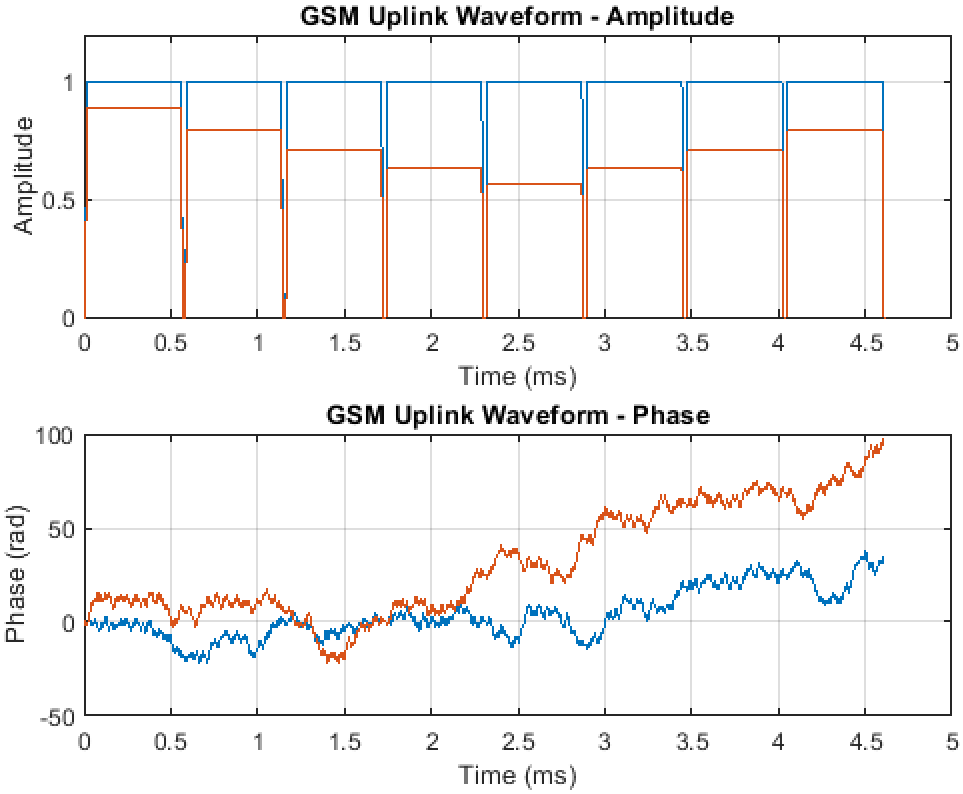
```
wfInfo = gsmInfo(cfggsmul)
```

```
wfInfo = struct with fields:
              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

```
Rs = wfInfo.SampleRate;
```

Create the GSM waveforms, containing one TDMA frame, by using the `gsmFrame` function. GSM TDMA frames are eight time slots, each separated by a guard period of 8.25 symbols or about 30.46x10e-3 ms. Plot each GSM waveform.

```
waveform = gsmFrame(cfggsmul);
waveform2 = gsmFrame(cfggsmul2);

t = (0:length(waveform)-1)/Rs*1e3;
subplot(2,1,1)
plot(t,[abs(waveform),abs(waveform2)])
grid on
axis([0 5 0 1.2])
title('GSM Uplink Waveform - Amplitude')
xlabel('Time (ms)')
ylabel('Amplitude')
subplot(2,1,2)
plot(t,[unwrap(angle(waveform)),unwrap(angle(waveform2))])
grid on
title('GSM Uplink Waveform - Phase')
xlabel('Time (ms)')
ylabel('Phase (rad)')
```



MEX Generation for GSM Uplink Waveform

Generate and run a GSM waveform MEX function from the helper function `createUplinkWaveform`. The `createUplinkWaveform` helper function creates a GSM uplink waveform.

Write MATLAB Function

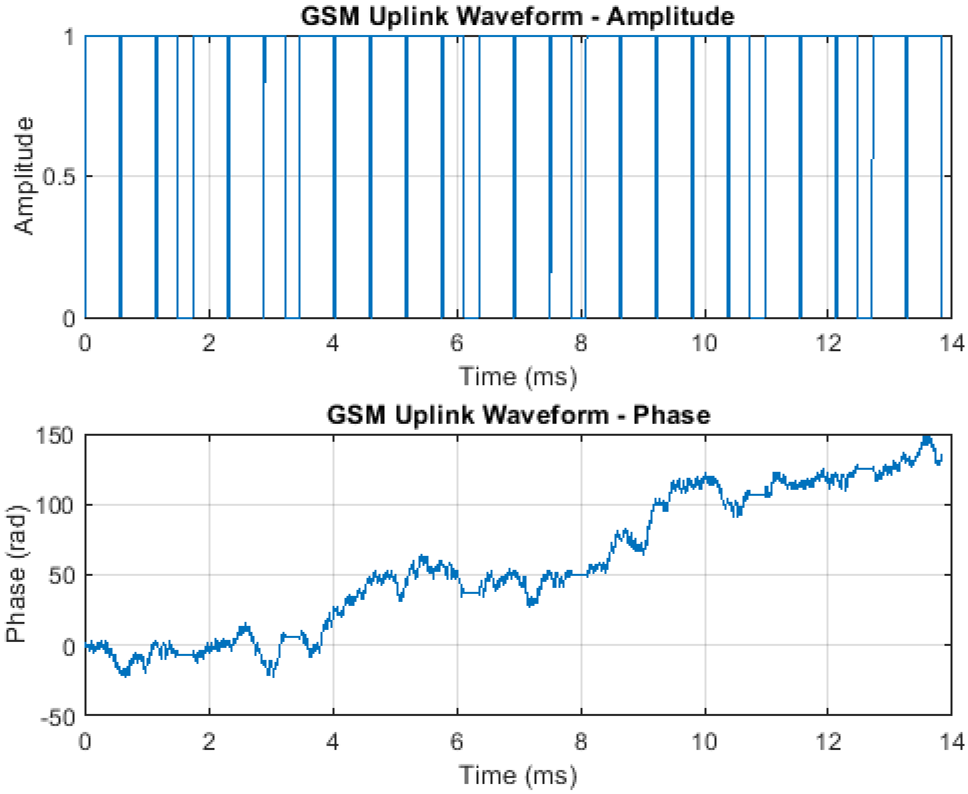
Open createUplinkWaveform.m to see the code. The `createUplinkWaveform` helper function generates a GSM uplink waveform by using the `gsmUplinkConfig` object and the `gsmInfo` and `gsmFrame` functions.

Generate GSM Waveform

Use the `createUplinkWaveform` helper function to create a GSM waveform containing three TDMA frames, and then plot the waveform.

```
[x,t] = createUplinkWaveform(3);

figure
subplot(2,1,1);
plot(t,abs(x));
grid on;
title('GSM Uplink Waveform - Amplitude');
```

```
xlabel('Time (ms)');
ylabel('Amplitude')
subplot(2,1,2);
plot(t,unwrap(angle(x)));
grid on;
title('GSM Uplink Waveform - Phase');
xlabel('Time (ms)');
ylabel('Phase (rad)')
```



Generate MEX Function

Code generation defaults to MEX code generation when you do not specify a build target. By default, `codegen` names the generated MEX function `createUplinkWaveform_mex`. Generate a MEX function from the `createUplinkWaveform` helper function, and then run the MEX function to create three TDMA frames.

```
codegen createUplinkWaveform -args 3
```

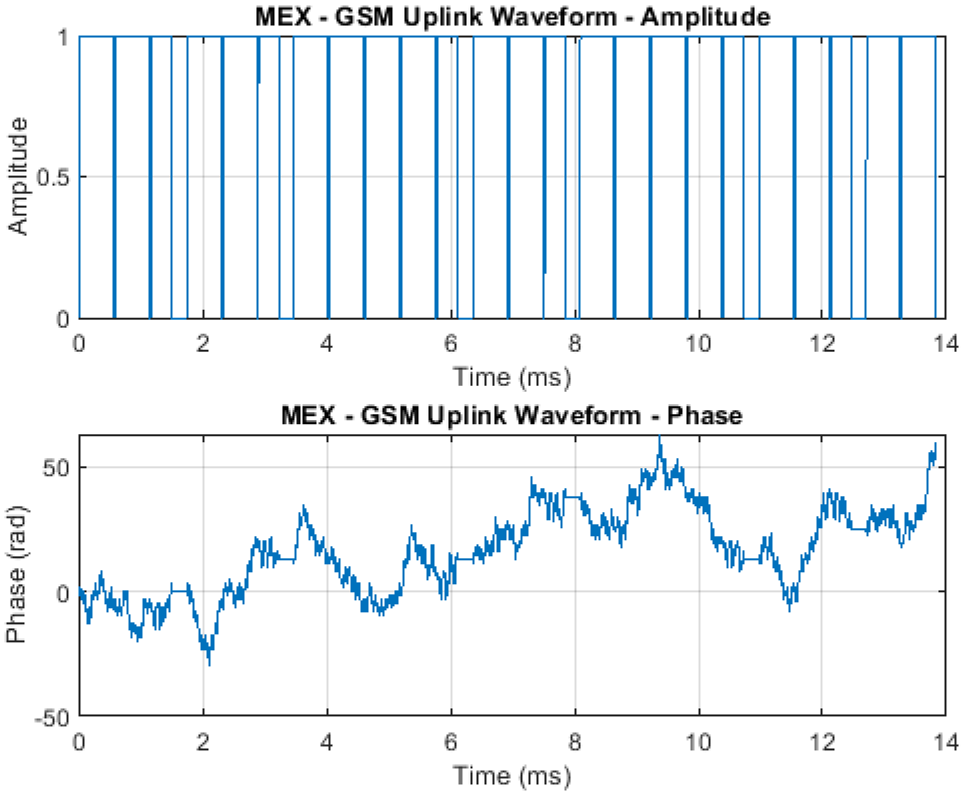
Generate Waveform Using MEX Function

Run the MEX function and plot the results. Since the waveform is created using random data, the phase plot changes each time you run the `generateUplinkFrame` helper function or `createUplinkWaveform_mex` function.

```
[x,t] = createUplinkWaveform_mex(3);
```

```
figure
```

```
subplot(2,1,1);
plot(t,abs(x));
grid on;
title('MEX - GSM Uplink Waveform - Amplitude');
xlabel('Time (ms)');
ylabel('Amplitude')
subplot(2,1,2);
plot(t,unwrap(angle(x)));
grid on;
title('MEX - GSM Uplink Waveform - Phase');
xlabel('Time (ms)');
ylabel('Phase (rad)')
```

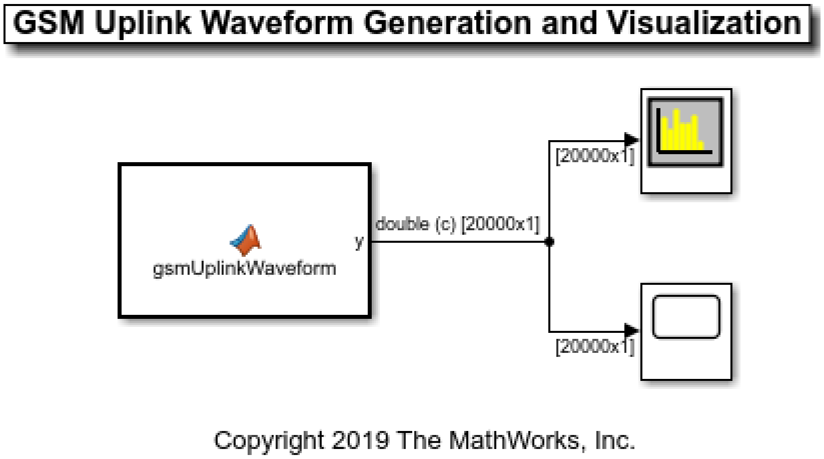


GSM Uplink Waveform Generation in Simulink

Model a GSM® waveform generator in Simulink® by using the MATLAB Function block and Communications Toolbox™ functions.

GSM Uplink Waveform Generation

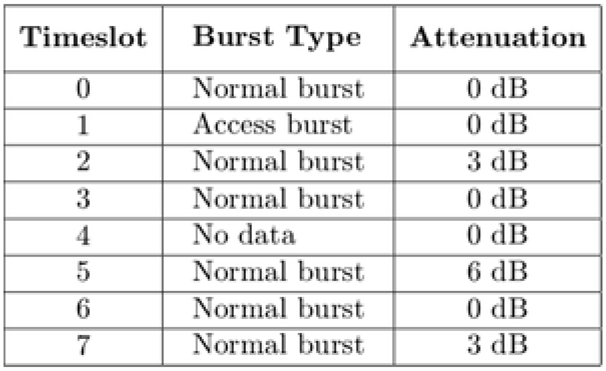
The MATLAB Function (Simulink) block contains the `gsmUplinkWaveform` function code. The code in the MATLAB Function block creates a GSM waveform by using the `gsmUplinkConfig` object and the `gsmFrame` function.



GSM Uplink Waveform Generation and Visualization

Copyright 2019 The MathWorks, Inc.

The `gsmUplinkConfig` object specifies 16 samples per symbol and the time slot configuration for the GSM uplink TDMA frame shown is this table.



Timeslot	Burst Type	Attenuation
0	Normal burst	0 dB
1	Access burst	0 dB
2	Normal burst	3 dB
3	Normal burst	0 dB
4	No data	0 dB
5	Normal burst	6 dB
6	Normal burst	0 dB
7	Normal burst	3 dB

The output waveform has 16 samples for each GMSK symbol. The `gsmFrame` function generates the samples of the waveform.

Explore the Model

In compliance with GSM standards 3GPP TS 45.001 and 3GPP TS 45.002, the sample time of the MATLAB Function block that contains the `gsmUplinkWaveform` function code is set to the GSM symbol rate of 1625e3/6 symbols per second. Display the current `gsmUplinkConfig` object settings by using the `gsmInfo` function.

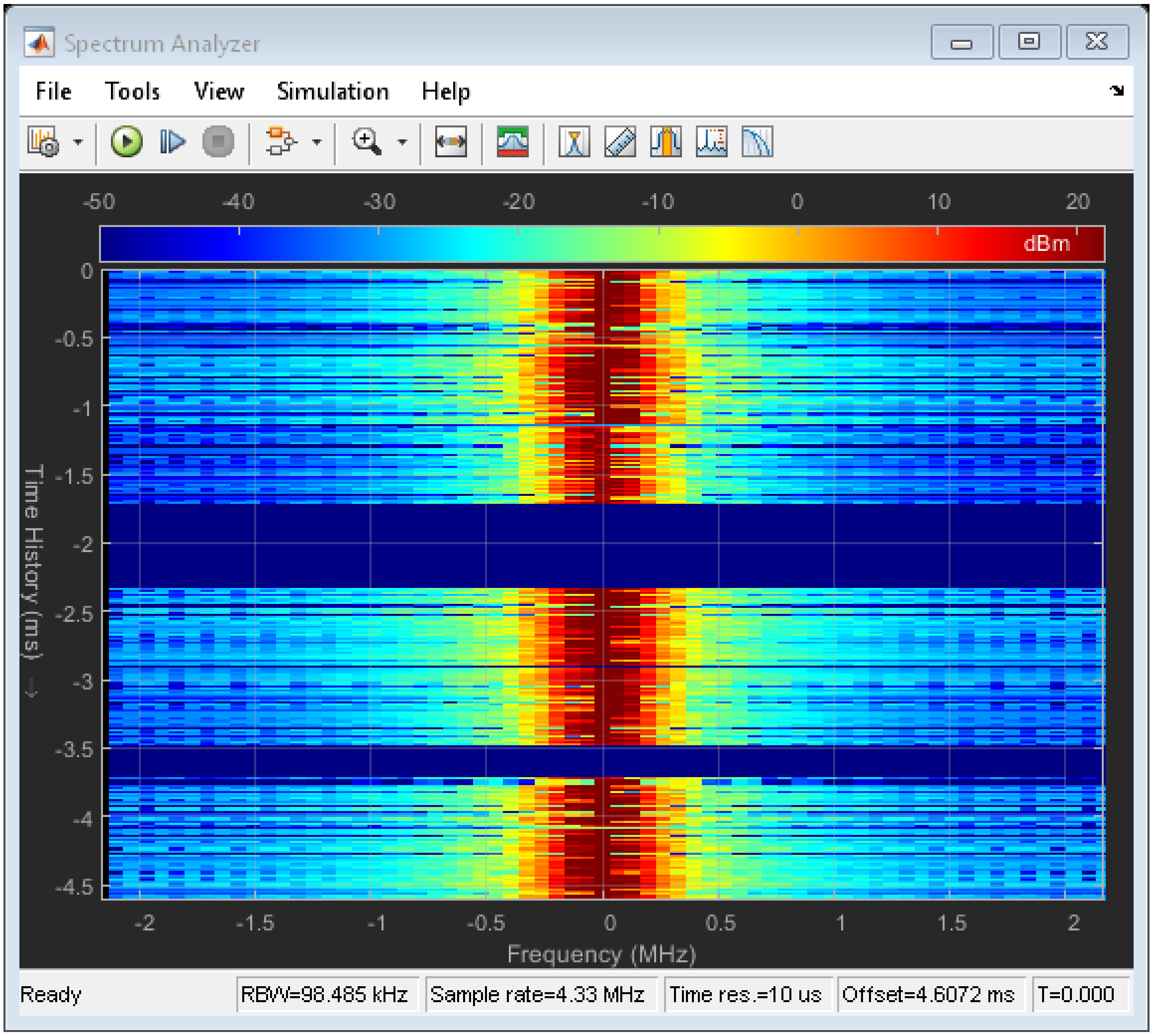
```
wfInfo = 

  struct with fields:

              SymbolRate: 2.7083e+05
              SampleRate: 4.3333e+06
    BandwidthTimeProduct: 0.3000
    BurstLengthInSymbols: 156.2500
       NumBurstsPerFrame: 8
    BurstLengthInSamples: 2500
    FrameLengthInSamples: 20000
```

The model sample time of the MATLAB Function (Simulink) block is set to `wfInfo.FrameLengthInSamples/wfInfo.SampleRate`. To view the **Sample time** parameter, open the Block Parameters dialog box by right-clicking the MATLAB Function block and selecting **Block Parameters (Subsystem)**.

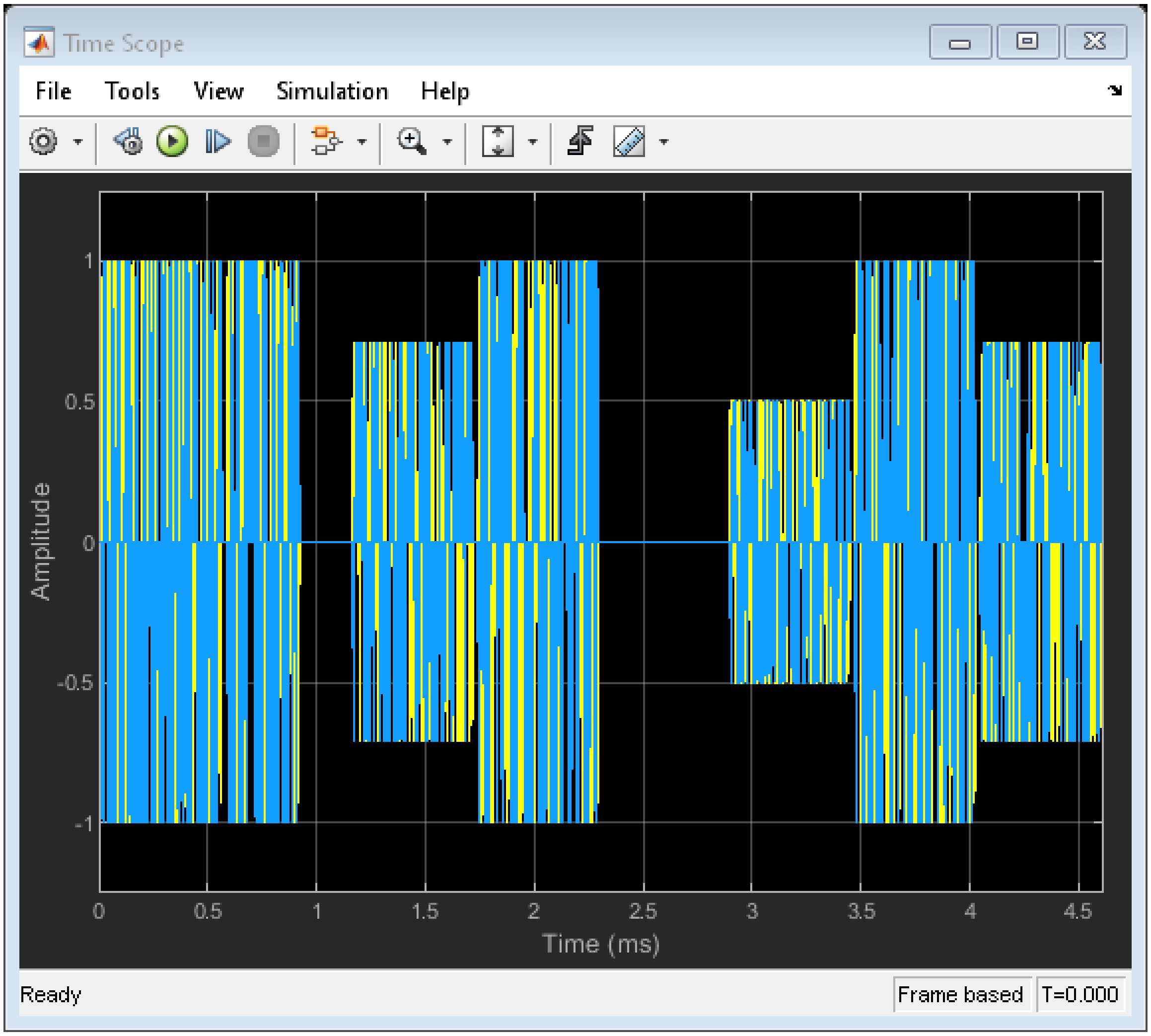
Before the simulation runs, you must configure the sample rate of the MATLAB Function block. The `PreLoadFcn` and `InitFcn` callback functions configure the MATLAB Function block by creating a `gsmUplinkConfig` object and `wfInfo` structure. To view the callback functions, on the **Modeling** tab, in the **Setup** section, select **Model Settings** > **Model Properties**. Then, on the **Callbacks** tab, select the `PreLoadFcn` or `InitFcn` callback function in the **Model callbacks** pane.

Results

Displays the time domain signal and the spectrogram by running the simulation.



Ready | RBW=98.485 kHz | Sample rate=4.33 MHz | Time res.=10 us | Offset=4.6072 ms | T=0.000

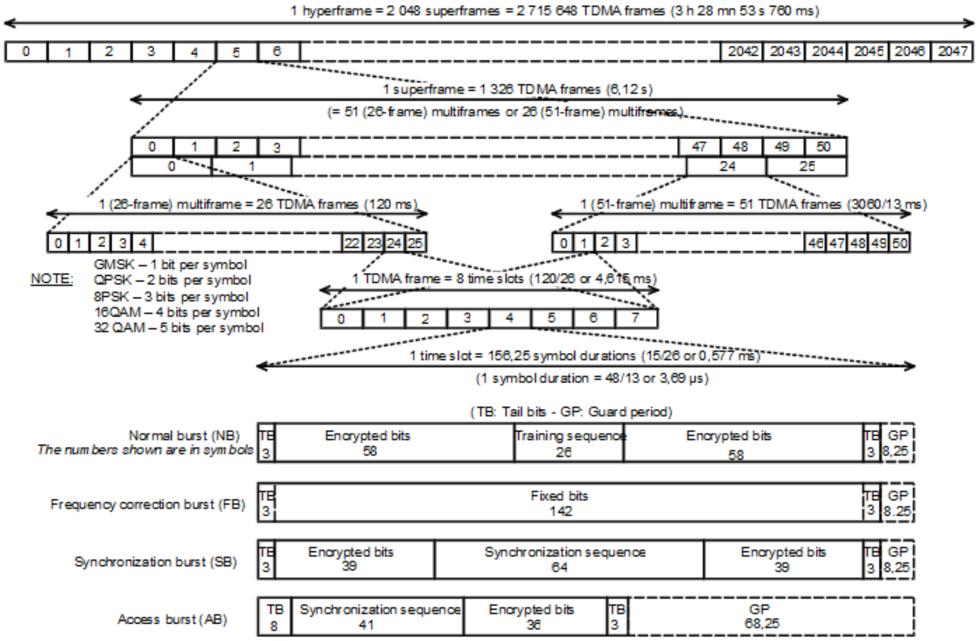


More About

GSM Frames, Time Slots, and Bursts

In GSM, transmissions consist of TDMA frames. Each GSM TDMA frame consists of eight time slots. The transmission data content of a time slot is called a burst. As described in Section 5.2 of 3GPP TS 45.011, a GSM time slot has a 156.25-symbol duration when using the normal symbol period, which is a time interval of 15/26 ms or about 576.9 microseconds. A guard period of 8.25 symbols or about 30.46 microseconds separates each time slot. The GSM standards describes a symbol as one bit period. Since GSM uses GMSK modulation, there is one bit per bit period. The transmission timing of a burst within a time slot is defined in terms of the bit number (BN). The BN refers to a particular bit period within a time slot. The bit with the lowest BN is transmitted first. BN0 is the first bit period, and BN156 is the last quarter-bit period.

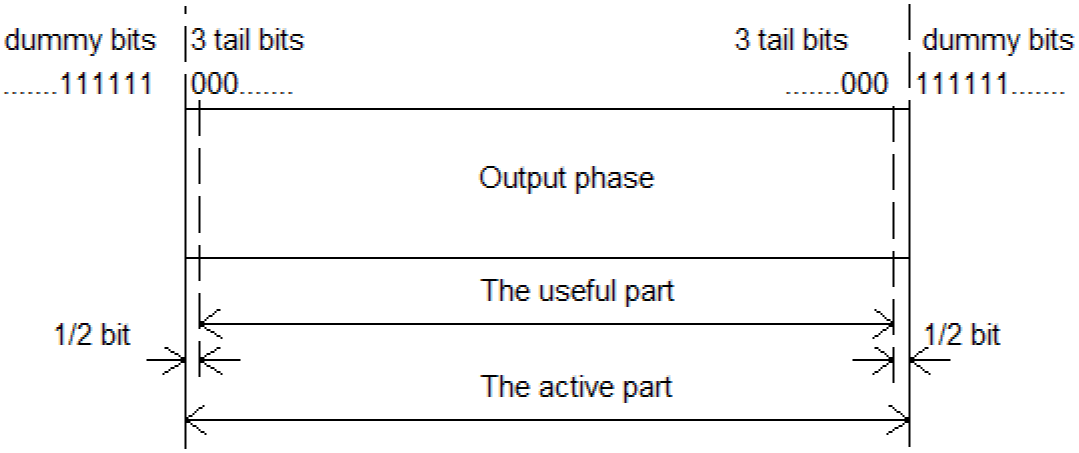
This image from 3GPP TS 45.011 shows the relationship between different frame types and the relationship between different burst types.


1 hyperframe = 2 048 superframes = 2 715 648 TDMA frames (3 h 28 mn 53 s 760 ms)
0
1
2
3
4
5
6
2042
2043
2044
2045
2046
2047
1 superframe = 1 326 TDMA frames (6,12 s)
(= 51 (26-frame) multiframes or 26 (51-frame) multiframes)
0
1
2
3
47
48
49
50
0
1
24
25
1 (26-frame) multiframe = 26 TDMA frames (120 ms)
0
1
2
3
4
22
23
24
25
1 (51-frame) multiframe = 51 TDMA frames (3060/13 ms)
0
1
2
3
46
47
48
49
50
NOTE:
GMSK – 1 bit per symbol
QPSK – 2 bits per symbol
8PSK – 3 bits per symbol
16QAM – 4 bits per symbol
32 QAM – 5 bits per symbol
1 TDMA frame = 8 time slots (120/26 or 4,615 ms)
0
1
2
3
4
5
6
7
1 time slot = 156,25 symbol durations (15/26 or 0,577 ms)
(1 symbol duration = 48/13 or 3,69 μs)
(TB: Tail bits - GP: Guard period)
Normal burst (NB)
The numbers shown are in symbols
TB 3
Encrypted bits 58
Training sequence 26
Encrypted bits 58
TB 3
GP 8,25
Frequency correction burst (FB)
TB 3
Fixed bits 142
TB 3
GP 8.25
Synchronization burst (SB)
TB 3
Encrypted bits 39
Synchronization sequence 64
Encrypted bits 39
TB 3
GP 8,25
Access burst (AB)
TB 8
Synchronization sequence 41
Encrypted bits 36
TB 3
GP 68,25

This table shows the supported burst types and their characteristics.

Burst Type	Description	Link Direction	Useful Duration
NB	Normal burst	Uplink/Downlink	147
FB	Frequency correction burst	Downlink	147
SB	Synchronization burst	Downlink	147
Dummy	Dummy burst	Downlink	147
AB	Access burst	Uplink	87
Off	No burst sent	Uplink/Downlink	0

Useful duration, described in Section 5.2.2 of 3GPP TS 45.002, is a characteristic of GSM bursts. The useful duration, or useful part, of a burst is defined as beginning halfway through BN0 and ending half a bit period before the start of the guard period. The guard period is the period between bursts in successive time slots. This figure, from Section 2.2 of 3GPP TS 45.004, shows the leading and trailing ½ bit difference between the useful and active parts of the burst.



For more information, see “GSM TDMA Frame Parameterization for Waveform Generation”.

Training Sequence Code (TSC)

Normal bursts include a training sequence bits field assigned a bit pattern based on the specified TSC. For GSM, you can select one of these eight training sequences for normal burst type time slots.

Training Sequence Code (TSC)	Training Sequence Bits (BN61, BN62, ..., BN86)
0	(0,0,1,0,0,1,0,1,1,1,0,0,0,0,1,0,0,0,1,0,0,1,0,1,1,1)
1	(0,0,1,0,1,1,0,1,1,1,0,1,1,1,1,0,0,0,1,0,1,1,0,1,1,1)
2	(0,1,0,0,0,0,1,1,1,0,1,1,1,0,1,0,0,1,0,0,0,0,1,1,1,0)
3	(0,1,0,0,0,1,1,1,1,0,1,1,0,1,0,0,0,1,0,0,0,1,1,1,1,0)
4	(0,0,0,1,1,0,1,0,1,1,1,0,0,1,0,0,0,0,0,1,1,0,1,0,1,1)
5	(0,1,0,0,1,1,1,0,1,0,1,1,0,0,0,0,0,1,0,0,1,1,1,0,1,0)
6	(1,0,1,0,0,1,1,1,1,1,0,1,1,0,0,0,1,0,1,0,0,1,1,1,1,1)
7	(1,1,1,0,1,1,1,1,0,0,0,1,0,0,1,0,1,1,1,0,1,1,1,1,0,0)

For more information, see Section 5.2.3 in 3GPP TS 45.002.

References

[1] 3GPP TS 45.001. "GSM/EDGE Physical layer on the radio path. General description." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[2] 3GPP TS 45.002. "GSM/EDGE Multiplexing and multiple access on the radio path." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

[3] 3GPP TS 45.004. "GSM/EDGE Modulation." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The SamplesPerSymbol, RiseTime, RiseDelay, FallTime, and FallDelay properties must be set when creating the object, and their settings are static in the generated code.
- The BurstType property must be set using the enumeration type instead of the string representation. Use these gsmDownlinkBurstType enumerations: gsmDownlinkBurstType.NB, gsmDownlinkBurstType.AB, and gsmUplinkBurstType.Off. For example, this code assigns an access burst in time slot 2 and 5.

```
cfg = gsmUplinkConfig

cfg = 

  gsmUplinkConfig with properties:

           BurstType: [NB    NB    NB    NB    NB    NB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0

cfg.BurstType([2 5] +1) = gsmUplinkBurstType.AB

cfg = 

  gsmUplinkConfig with properties:

           BurstType: [NB    NB    AB    NB    NB    AB    NB    NB]
    SamplesPerSymbol: 16
                 TSC: [0 1 2 3 4 5 6 7]
         Attenuation: [0 0 0 0 0 0 0 0]
            RiseTime: 2
           RiseDelay: 0
            FallTime: 2
           FallDelay: 0
```

See Also

Objects

gsmDownlinkConfig

Functions

gsmCheckTimeMask | gsmFrame | gsmInfo

Topics

"GSM TDMA Frame Parameterization for Waveform Generation"

Introduced in R2019b

comm.HadamardCode

Package: `comm`

Generate Hadamard code

Description

The `HadamardCode` object generates a Hadamard code from a Hadamard matrix, whose rows form an orthogonal set of codes. You can use orthogonal codes for spreading in communication systems in which the receiver is perfectly synchronized with the transmitter. In these systems, the despreading operation is ideal, because the codes decorrelate completely.

To generate a Hadamard code:

1. Define and set up your Hadamard code object. See “Construction” on page 3-824.
2. Call `step` to generate a Hadamard according to the properties of `comm.HadamardCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`H = comm.HadamardCode` creates a Hadamard code generator System object, `H`. This object generates Hadamard codes from a set of orthogonal codes.

`H = comm.HadamardCode(Name,Value)` creates a Hadamard code generator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Length`

Length of generated code

Specify the length of the generated code as a numeric, integer scalar value with a power of two. The default is `64`.

`Index`

Row index of Hadamard matrix

Specify the row index of the Hadamard matrix as a numeric, integer scalar value in the range `[0, 1, ... , N-1]`. *N* is the value of the `Length` on page 3-0 property. The default is `60`. An *N*×*N* Hadamard matrix, denoted as *P*(*N*), is defined recursively as follows: *P*(1) = [1] *P*(2*N*) = [*P*(*N*) *P*(*N*); *P*(*N*) –*P*(*N*)] The *N*x*N* Hadamard matrix has the property that *P*(*N*)×*P*(*N*)' = *N*×`eye`(*N*). The `step` method outputs code samples from the row of the Hadamard matrix that you specify in this property.

When you set this property to an integer *k*, the output code has exactly *k* zero crossings, for *k* = 0, 1, ... , *N*–1.

`SamplesPerFrame`

Number of output samples per frame

Specify the number of Hadamard code samples that the `step` method outputs as a numeric, positive, integer scalar value. The default is `1`.

When you set this property to a value of *M*, the `step` method outputs *M* samples of a Hadamard code of length *N*. *N* equals the length of the code that you specify in the `Length` on page 3-0 property.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `int8`. The default is `double`.

Methods

reset	Reset states of Hadamard code generator object
step	Generate Hadamard code

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Hadamard Code Sequence

Generate 10 samples of a Hadamard code sequence having a length of 128.

```
hadamard = comm.HadamardCode('Length',128,'SamplesPerFrame',10)
```

```
hadamard = 
  comm.HadamardCode with properties:

             Length: 128
              Index: 60
    SamplesPerFrame: 10
     OutputDataType: 'double'

```

```
seq = hadamard()
```

```
seq = 10×1

     1
     1
     1
     1
    -1
    -1
    -1
    -1
    -1
    -1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Hadamard Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.OVSFCode` | `comm.WalshCode`

Introduced in R2012a

reset

System object: `comm.HadamardCode`
Package: `comm`

Reset states of Hadamard code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `HadamardCode` object, `H`.

step

System object: `comm.HadamardCode`
Package: `comm`

Generate Hadamard code

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(H)` outputs a frame of the Hadamard code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Hadamard code corresponds to one of the rows of an *N*x*N* Hadamard matrix, where *N* is a nonnegative power of 2, which you specify in the `Length` property. Use the `Index` property to choose the row of the Hadamard matrix. The `step` method outputs the code in a bi-polar format with `0` and `1` mapped to `1` and `-1`, respectively.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.HDLCRCDetector

Package: comm

Detect errors in input data using CRC

Description

This HDL-optimized cyclic redundancy code (CRC) detector System object computes a checksum on the input data and compares the result against the input checksum. Instead of frame processing, the HDLCRCDetector System object processes streaming data. The object has frame synchronization control signals for both input and output data streams.

To compute and compare checksums:

1. Create the comm.HDLCRCDetector object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
CRCDet = comm.HDLCRCDetector
CRCDet = comm.HDLCRCDetector(Name,Value)
CRCDet = comm.HDLCRCDetector(poly,Name,Value)
```

Description

CRCDet = comm.HDLCRCDetector creates an HDL-optimized CRC detector System object, CRCDet, that detects errors in the input data according to a specified generator polynomial.

CRCDet = comm.HDLCRCDetector(Name,Value)sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
CRCDet = comm.HDLCRCDetector('Polynomial',[1 0 0 0 1 0 0 0 0], ...
'FinalXORValue',[1 1 0 0 0 0 0 0]);
```

specifies a CRC8 polynomial and an 8-bit value to XOR with the final checksum.

CRCDet = comm.HDLCRCDetector(poly,Name,Value) creates an HDL-optimized CRC detector System object, CRCDet, with the Polynomial property set to poly, and the other specified property names set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Polynomial` — Generator polynomial
`[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]` (default) | binary vector

Generator polynomial, specified as a binary vector, with coefficients in descending order of powers. The vector length must be equal to the degree of the polynomial plus 1.

`InitialState` — Initial conditions of shift register
`0` (default) | binary scalar | binary vector

Initial conditions of the shift register, specified as a binary, double-precision or single-precision scalar or vector. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

`DirectMethod` — Method of calculating checksum
`false` (default) | `true`

Method of calculating checksum, specified as a logical scalar. When this property is `true`, the object uses the direct algorithm for CRC checksum calculations.

To learn about direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

`ReflectInput` — Input byte order
`false` (default) | `true`

Input byte order, specified as a logical scalar. When this property is `true`, the object flips the input data on a bytewise basis before it enters the shift register.

`ReflectCRCChecksum` — Checksum byte order
`false` (default) | `true`

Checksum byte order, specified as a logical scalar. When this property is `true`, the object flips the output CRC checksum around its center.

`FinalXORValue` — Checksum mask
`0` (default) | binary scalar | binary vector

Checksum mask, specified as a binary, double- or single-precision data type scalar or vector. The object XORs the checksum with this value before appending the checksum to the input data. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

Usage

Syntax

```
[Y,startOut,endOut,validOut,err] = CRCn(X,startIn,endIn,validIn)
```

Description

`[Y,startOut,endOut,validOut,err] = CRCn(X,startIn,endIn,validIn)` computes CRC checksums for an input message `X` based on the control signals and compares the computed checksum with input checksum. If the two checksums are not equal, the output `err` is set to 1 `true`).

Input Arguments

X — Input message and appended checksum
binary column vector | scalar integer

Input message and appended checksum, specified as a binary vector or a scalar integer representing several bits. For example, vector input `[0,0,0,1,0,0,1,1]` is equivalent to `uint8` input `19`.

If the input is a vector, the data type can be double or logical. If the input is a scalar, the data type can be unsigned integer or unsigned fixed-point with 0 fractional bits (`fi([],0,N,0)`).

`X` can be part or all of the message to be checked.

The length of `X` must be less than or equal to the CRC length, and the CRC length must be divisible by the length of `X`.

The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical` | `unsigned fi`

`startIn` — Start of input message
logical scalar

Start of the input message, specified as a logical scalar.

`endIn` — End of input message
logical scalar

End of the input message, specified as a logical scalar.

`validIn` — Validity of input data
logical scalar

Validity of input data, specified as a logical scalar. When `validIn` is 1 (`true`), the object computes the CRC checksum for input `X`.

Output Arguments

Y — Message with checksum removed
binary column vector | scalar integer

Message with checksum removed, returned as a scalar integer or binary column vector with the same width and data type as input `X`.

`startOut` — Start of input message
logical scalar

Start of the input message, returned as a logical scalar.

`endOut` — End of input message
logical scalar

End of the input message, returned as a logical scalar.

`validOut` — Validity of input data
logical scalar

Validity of input data, returned as a logical scalar. When `validOut` is 1 (`true`), the output data `Y` is valid.

`err` — Checksum mismatch
logical scalar

Checksum mismatch, returned as a logical scalar. `err` is 1 (`true`) when the input checksum does not match the calculated checksum.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

CRC Encode and Decode for HDL

Encode and decode a signal using the HDL-optimized CRC generator and detector System objects. This example shows how to include each object in a function for HDL code generation.

Create a 32-bit message to be encoded, in two 16-bit columns.

```
msg = randi([0 1],16,2);
```

Run for 12 steps to accommodate the latency of both objects. Assign control signals for all steps. The first two samples are the valid data, and the remainder are processing latency.

```
numSteps = 12;
startIn = logical([1 0 0 0 0 0 0 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0 0 0 0 0 0 0]);
validIn = logical([1 1 0 0 0 0 0 0 0 0 0 0]);
```

Pass random input to the `HDLCRCGenerator` System object™ while it is processing the input message. The random data is not encoded because the input valid signal is `0` for steps 3 to 10.

```
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];
```

Write a function that creates and calls each System object™. You can generate HDL from these functions. The generator and detector objects both have a CRC length of 16 and use the default polynomial.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [dataOut,startOut,endOut,validOut] = HDLCRC16Gen(dataIn,startIn,endIn,validIn)
%HDLCRC16Gen
% Generates CRC checksum using the comm.HDLCRCGenerator System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcg16;
  if isempty(crcg16)
    crcg16 = comm.HDLCRCGenerator()
  end
  [dataOut,startOut,endOut,validOut] = crcg16(dataIn,startIn,endIn,validIn);
end
```

```
function [dataOut,startOut,endOut,validOut,err] = HDLCRC16Det(dataIn,startIn,endIn,validIn)
%HDLCRC16Det
% Checks CRC checksum using the comm.HDLCRCDetector System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcd16;
  if isempty(crcd16)
    crcd16 = comm.HDLCRCDetector()
  end
  [dataOut,startOut,endOut,validOut,err] = crcd16(dataIn,startIn,endIn,validIn);
end
```

Call the CRC generator function. The encoded message is the original message plus a 16 bit checksum.

```
 for i =  1:numSteps
 [dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)] = ...
     HDLCRC16Gen(logical(dataIn(:,i)),startIn(i),endIn(i),validIn(i));
 end
```

```
crcg16 = 

  comm.HDLCRCGenerator with properties:

           Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
         InitialState: 0
         DirectMethod: false
```

```
          ReflectInput: false
    ReflectCRCChecksum: false
         FinalXORValue: 0
```

Add noise by flipping a bit in the message.

```
dataOutNoise = dataOutGen;
dataOutNoise(2,4) = ~dataOutNoise(2,4);
```

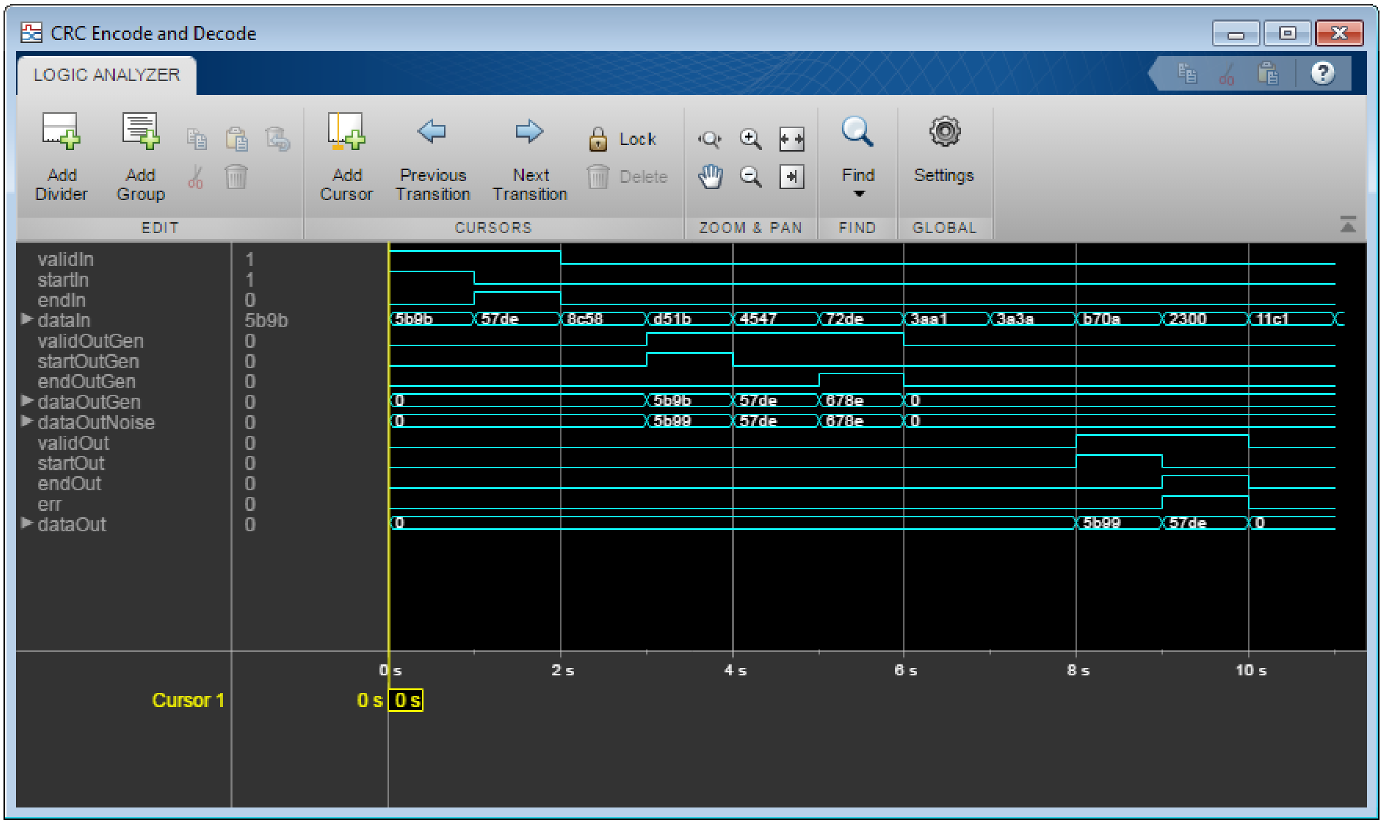
Call the CRC detector function. The output of the detector is the input message with the checksum removed. If the input checksum was not correct, the `err` flag is set with the last word of the output.

```
for i = 1:numSteps
[dataOut(:,i),startOut(i),endOut(i),validOut(i),err(i)] = ...
    HDLCRC16Det(logical(dataOutNoise(:,i)),startOutGen(i),endOutGen(i),validOutGen(i));
end
```

```
crcd16 =

  comm.HDLCRCDetector with properties:

            Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
          InitialState: 0
          DirectMethod: false
          ReflectInput: false
    ReflectCRCChecksum: false
         FinalXORValue: 0
```

Use the **Logic Analyzer** to view the input and output signals.

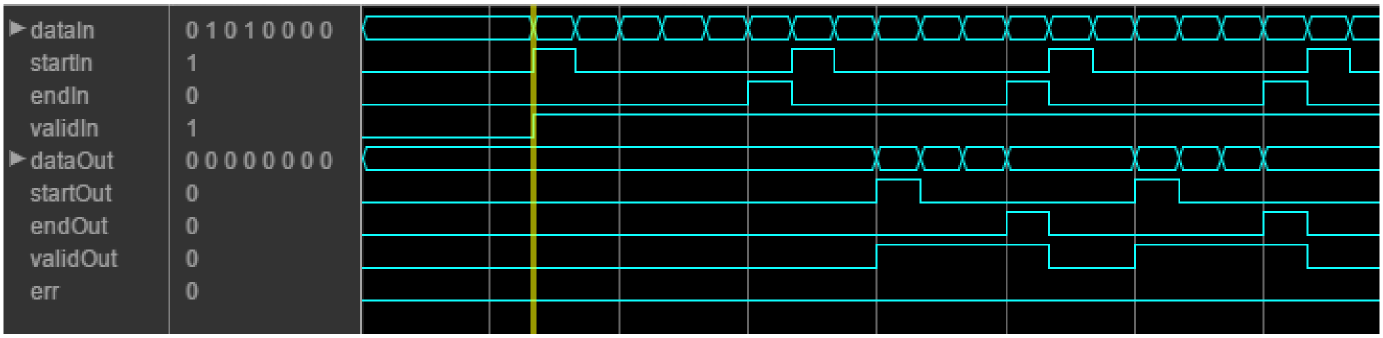
```
channels = {'validIn','startIn','endIn',...
    {'dataIn','Radix','Hexadecimal'},...
    'validOutGen','startOutGen','endOutGen',...
    {'dataOutGen','Radix','Hexadecimal'},...
    {'dataOutNoise','Radix','Hexadecimal'},...
    'validOut','startOut','endOut','err',...
    {'dataOut','Radix','Hexadecimal'}};
la = dsp.LogicAnalyzer('Name','CRC Encode and Decode','NumInputPorts',length(channels),...
                      'BackgroundColor','Black','DisplayChannelHeight',8);

 for ii = 1:length(channels)
    if iscell(channels{ii})
        % Display data signals as hexadecimal integers
        c = channels{ii};
        modifyDisplayChannel(la,ii,'Name',c{1},c{2},c{3})
        % Convert binary column vector to integer
        dat2 = uint16(bi2de(eval(c{1})'));
        chanData{ii} = squeeze(dat2);
    else
        modifyDisplayChannel(la,ii,'Name',channels{ii})
        chanData{ii} = squeeze(eval(channels{ii})');
    end
 end
la(chanData{:})
```



Algorithms

Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. The input frames are contiguous. The output frames include space between them because the detector block removes the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported.

Initial Delay

The `HDLCRCDetector` System object introduces a latency on the output. You can compute the latency as follows, assuming the input data is continuous:

```
initialDelay =  3 * (CRCLength/inputDataWidth) + 2
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.CRCDetector` | `comm.HDLCRCGenerator`

Blocks

General CRC Syndrome Detector HDL Optimized

Introduced in R2012b

comm.HDLCRCGenerator

Package: `comm`

Generate CRC code bits and append to input data

Description

This HDL-optimized cyclic redundancy code (CRC) generator System object generates cyclic redundancy code (CRC) bits. Instead of frame processing, the `HDLCRCGenerator` System object processes streaming data. The object has frame synchronization control signals for both input and output data streams.

To generate cyclic redundancy code bits:

1. Create the `comm.HDLCRCGenerator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
CRCGen = comm.HDLCRCGenerator
CRCGen = comm.HDLCRCGenerator(Name,Value)
CRCGen = comm.HDLCRCGenerator(poly,Name,Value)
```

Description

`CRCGen = comm.HDLCRCGenerator` creates an HDL-optimized CRC generator System object, `CRCGen`. This object generates CRC bits according to a specified generator polynomial and appends them to the input data.

`CRCGen = comm.HDLCRCGenerator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
CRCGen = comm.HDLCRCGenerator('Polynomial',[1 0 0 0 1 0 0 0 0], ...
'FinalXORValue',[1 1 0 0 0 0 0 0]);
```

specifies a CRC8 polynomial and an 8-bit value to XOR with the final checksum.

`CRCGen = comm.HDLCRCGenerator(poly,Name,Value)` sets the `Polynomial` property to `poly`, and the other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Polynomial — Generator polynomial

[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1] (default) | binary vector

Generator polynomial, specified as a binary vector, with coefficients in descending order of powers. The vector length must be equal to the degree of the polynomial plus 1.

InitialState — Initial conditions of shift register

0 (default) | binary scalar | binary vector

Initial conditions of the shift register, specified as a binary, double-precision or single-precision scalar or vector. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

DirectMethod — Method of calculating checksum

false (default) | true

Method of calculating checksum, specified as a logical scalar. When this property is `true`, the object uses the direct algorithm for CRC checksum calculations.

To learn about direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

ReflectInput — Input byte order

false (default) | true

Input byte order, specified as a logical scalar. When this property is `true`, the object flips the input data on a bytewise basis before it enters the shift register.

ReflectCRCChecksum — Checksum byte order

false (default) | true

Checksum byte order, specified as a logical scalar. When this property is `true`, the object flips the output CRC checksum around its center.

FinalXORValue — Checksum mask

0 (default) | binary scalar | binary vector

Checksum mask, specified as a binary, double- or single-precision data type scalar or vector. The object XORs the checksum with this value before appending the checksum to the input data. If you specify this property as a vector, the vector length is the degree of the generator polynomial that you specify in the `Polynomial` property. If you specify this property as a scalar, the object expands the value to a vector of length equal to the degree of the generator polynomial.

Usage

Syntax

```
[Y,startOut,endOut,validOut] = CRCn(X,startIn,endIn, validIn)
```

Description

`[Y,startOut,endOut,validOut] = CRCn(X,startIn,endIn, validIn)` generates CRC checksums for input message X based on control signals and appends the checksums to X.

Input Arguments

X — Input message

binary column vector | scalar integer

Input message, specified as a binary vector or a scalar integer representing several bits. For example, vector input `[0,0,0,1,0,0,1,1]` is equivalent to `uint8` input `19`.

If the input is a vector, the data type can be double or logical. If the input is a scalar, the data type can be unsigned integer or unsigned fixed-point with 0 fractional bits (`fi([],0,N,0)`).

X can be part or all of the message to be encoded.

The length of X must be less than or equal to the CRC length, and the CRC length must be divisible by the length of X.

The CRC length is the order of the polynomial that you specify in the `Polynomial` property.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `logical` | `unsigned fi`

`startIn` — Start of input message

logical scalar

Start of the input message, specified as a logical scalar.

`endIn` — End of input message

logical scalar

End of the input message, specified as a logical scalar.

`validIn` — Validity of input data

logical scalar

Validity of input data, specified as a logical scalar. When `validIn` is 1 (`true`), the object computes the CRC checksum for input X.

Output Arguments

Y — Output message with appended checksum

binary column vector | scalar integer

Output message, consisting of X with appended checksum, returned as a scalar integer or binary column vector with the same width and data type as input X.

`startOut` — Start of input message

logical scalar

Start of the input message, returned as a logical scalar.

`endOut` — End of input message

logical scalar

End of the input message, returned as a logical scalar.

validOut — Validity of input data
logical scalar

Validity of input data, returned as a logical scalar. When `validOut` is 1 (`true`), the output data `Y` is valid.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

CRC Encode and Decode for HDL

Encode and decode a signal using the HDL-optimized CRC generator and detector System objects. This example shows how to include each object in a function for HDL code generation.

Create a 32-bit message to be encoded, in two 16-bit columns.

```
msg = randi([0 1],16,2);
```

Run for 12 steps to accommodate the latency of both objects. Assign control signals for all steps. The first two samples are the valid data, and the remainder are processing latency.

```
numSteps = 12;
startIn = logical([1 0 0 0 0 0 0 0 0 0 0 0]);
endIn   = logical([0 1 0 0 0 0 0 0 0 0 0 0]);
validIn = logical([1 1 0 0 0 0 0 0 0 0 0 0]);
```

Pass random input to the `HDLCRCGenerator` System object™ while it is processing the input message. The random data is not encoded because the input valid signal is `0` for steps 3 to 10.

```
randIn = randi([0, 1],16,numSteps-2);
dataIn = [msg randIn];
```

Write a function that creates and calls each System object™. You can generate HDL from these functions. The generator and detector objects both have a CRC length of 16 and use the default polynomial.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function [dataOut,startOut,endOut,validOut] = HDLCRC16Gen(dataIn,startIn,endIn,validIn)
%HDLCRC16Gen
% Generates CRC checksum using the comm.HDLCRCGenerator System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcg16;
  if isempty(crcg16)
    crcg16 = comm.HDLCRCGenerator()
  end
  [dataOut,startOut,endOut,validOut] = crcg16(dataIn,startIn,endIn,validIn);
end
```

```
function [dataOut,startOut,endOut,validOut,err] = HDLCRC16Det(dataIn,startIn,endIn,validIn)
%HDLCRC16Det
% Checks CRC checksum using the comm.HDLCRCDetector System object(TM)
% dataIn is a binary column vector.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent crcd16;
  if isempty(crcd16)
    crcd16 = comm.HDLCRCDetector()
  end
  [dataOut,startOut,endOut,validOut,err] = crcd16(dataIn,startIn,endIn,validIn);
end
```

Call the CRC generator function. The encoded message is the original message plus a 16 bit checksum.

```
 for i =  1:numSteps
 [dataOutGen(:,i),startOutGen(i),endOutGen(i),validOutGen(i)] = ...
     HDLCRC16Gen(logical(dataIn(:,i)),startIn(i),endIn(i),validIn(i));
 end
```

```
crcg16 = 

  comm.HDLCRCGenerator with properties:

            Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
          InitialState: 0
          DirectMethod: false
          ReflectInput: false
    ReflectCRCChecksum: false
         FinalXORValue: 0
```

Add noise by flipping a bit in the message.

```
dataOutNoise = dataOutGen;
dataOutNoise(2,4) = ~dataOutNoise(2,4);
```

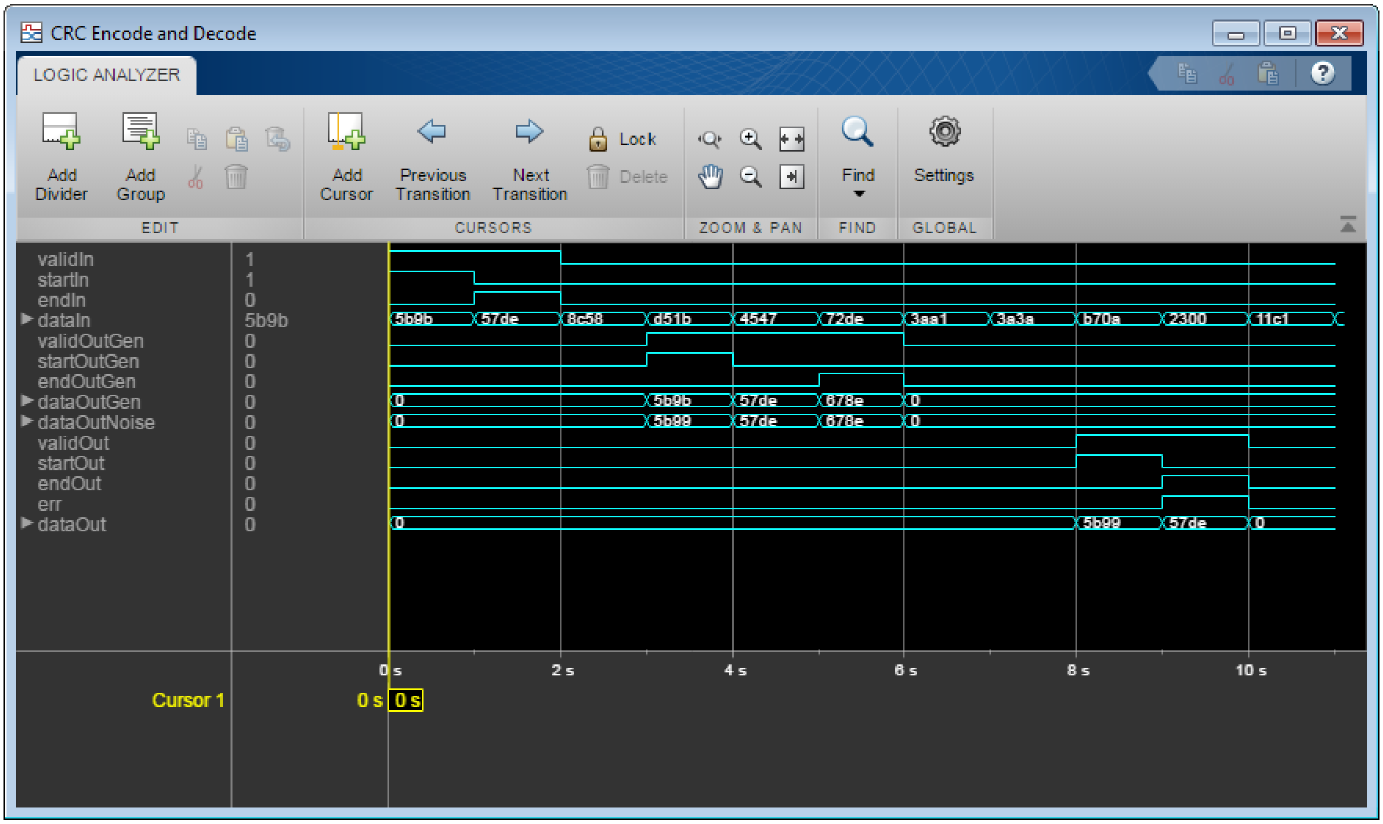
Call the CRC detector function. The output of the detector is the input message with the checksum removed. If the input checksum was not correct, the `err` flag is set with the last word of the output.

```
for i = 1:numSteps
[dataOut(:,i),startOut(i),endOut(i),validOut(i),err(i)] = ...
    HDLCRC16Det(logical(dataOutNoise(:,i)),startOutGen(i),endOutGen(i),validOutGen(i));
end
```

```
crcd16 = 

  comm.HDLCRCDetector with properties:

            Polynomial: [1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]
          InitialState: 0
          DirectMethod: false
          ReflectInput: false
    ReflectCRCChecksum: false
         FinalXORValue: 0
```

Use the **Logic Analyzer** to view the input and output signals.

```
channels = {'validIn','startIn','endIn',...
    {'dataIn','Radix','Hexadecimal'},...
    'validOutGen','startOutGen','endOutGen',...
    {'dataOutGen','Radix','Hexadecimal'},...
    {'dataOutNoise','Radix','Hexadecimal'},...
    'validOut','startOut','endOut','err',...
    {'dataOut','Radix','Hexadecimal'}};
la = dsp.LogicAnalyzer('Name','CRC Encode and Decode','NumInputPorts',length(channels),...
                      'BackgroundColor','Black','DisplayChannelHeight',8);

 for ii = 1:length(channels)
    if iscell(channels{ii})
        % Display data signals as hexadecimal integers
        c = channels{ii};
        modifyDisplayChannel(la,ii,'Name',c{1},c{2},c{3})
        % Convert binary column vector to integer
        dat2 = uint16(bi2de(eval(c{1})'));
        chanData{ii} = squeeze(dat2);
    else
        modifyDisplayChannel(la,ii,'Name',channels{ii})
        chanData{ii} = squeeze(eval(channels{ii})');
    end
 end
la(chanData{:})
```

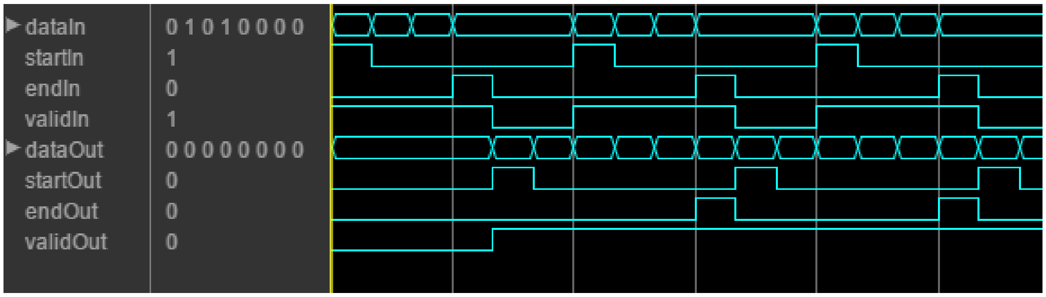


Algorithms

Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with an 8-bit binary vector input. To insert the checksum word, input frames must have enough space between the.

This waveform diagram shows continuous input data. The block also supports noncontinuous data. The output valid signal matches the input valid pattern.



Initial Delay

The `HDLCRCGenerator` System object introduces a latency on the output. You can compute the latency as follows, assuming the input data is continuous:

```
initialDelay = (CRCLength/inputDataWidth) + 2
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.CRCGenerator` | `comm.HDLCRCDetector`

Blocks

General CRC Generator HDL Optimized

Introduced in R2012a

comm.HDLRSDecoder

Package: `comm`

Decode message using Reed-Solomon decoder

Description

The HDL-optimized `HDLRSDecoder` System object recovers a message vector from a Reed-Solomon (RS) codeword vector. For proper decoding, the code and polynomial property values for this object must match those values in the corresponding encoder.

To recover a message vector from a Reed-Solomon codeword vector:

1. Create the `comm.HDLRSDecoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Troubleshooting

- Each input frame must contain more than `(N-K)*2` symbols and fewer than or exactly `N` symbols. The object infers a shortened code when the number of valid data samples between `startIn` and `endIn` is less than `N`. A shortened code still requires `N` cycles to perform the Chien search. If the input message is less than `N` symbols, leave a guard interval of at least `N - size` inactive cycles before starting the next frame, where `size`is the message length.
- The decoder can operate on up to four messages at a time. If the object receives the start of a fifth message before completely decoding the first message, the object drops data samples from the first message. To avoid this issue, increase the number of inactive cycles between input messages.
- The generator polynomial is not specified explicitly. However, it is defined by the codeword length, the message length, and the B value for the starting exponent of the roots. To get the value of B from a generator polynomial, use the `genpoly2b` function.

Creation

Syntax

```
RSDec = comm.HDLRSDecoder
RSDec = comm.HDLRSDecoder(Name,Value)
RSDec = comm.HDLRSDecoder(N,K,Name,Value)
```

Description

`RSDec = comm.HDLRSDecoder` creates an HDL-optimized RS decoder System object, `RSDec`, that performs Reed-Solomon decoding.

`RSDec = comm.HDLRSDecoder(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
comm.HDLRSDecoder('BSource','Property','B',2)
```

sets a starting power of 2 for the roots of the primitive polynomial.

`RSDec = comm.HDLRSDecoder(N,K,Name,Value)` sets the `CodewordLength` property to `N`, the `MessageLength` property to `K`, and other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

B — Starting power for roots of primitive polynomial
1 (default) | positive integer

Starting power for roots of the primitive polynomial, specified as a positive integer.

Dependencies

The object uses this value when you set `BSource` to `'Property'`.

`BSource` — Source of starting power for roots of primitive polynomial
`'Auto'` (default) | `'Property'`

Source of the starting power for roots of the primitive polynomial, specified as either `'Property'` or `'Auto'`. When you select `'Auto'`, the object uses `B = 1`.

`CodewordLength` — Number of symbols, N, in RS codeword
7 (default) | positive integer

Number of symbols, N, in the RS codeword, specified as a positive integer. This value is rounded up to 2^M–1. M is the degree of the primitive polynomial, as specified by the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. The difference of `CodewordLength` – `MessageLength` must be an even integer.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

If you set `PrimitivePolynomialSource` to `'Auto'`, then `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16} – 1.

If you set `PrimitivePolynomialSource` to `'Property'`, then `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M– 1. M must be in the range $3 \leq M \leq 16$.

`MessageLength` — Message length, K
3 (default) | positive integer

Message length, K, specified as a positive integer. The difference of `CodewordLength` – `MessageLength` must be an even integer.

`NumErrorsOutputPort` — Enable number of errors output argument
`false` (default) | `true`

When you set this property to `true`, the object returns the number of corrected errors. The number of corrected errors is not valid when `errOut` is `true`, since there were more errors than could be corrected.

`PrimitivePolynomialSource` — Source of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the primitive polynomial, specified as either `'Property'` or `'Auto'`.

- When you set this property to `'Auto'`, the object uses a primitive polynomial of degree *M* = `ceil`(**log2**(`CodewordLength`+1)), which is the result of `fliplr`(`de2bi`(`primpoly`(*M*))).
- When you set this property to `'Property'`, you must specify a polynomial using the `PrimitivePolynomial` property.

`PrimitivePolynomial` — Primitive polynomial

`[1 0 1 1]` (default) | binary row vector

Primitive polynomial, specified as a binary row vector that represents a primitive polynomial over `gf`(2) of degree *M*, in descending order of powers. The polynomial defines the finite field `gf`(2^M) corresponding to the integers that form messages and codewords.

Dependencies

The object uses this value when you set `PrimitivePolynomialSource` to `'Property'`.

Usage

Syntax

```
[Y,startOut,endOut,validOut,errOut] = RSDec(X,startIn,endIn,validIn)
[Y,startOut,endOut,validOut,errOut,numErrors] = RSDec(X,startIn,endIn,
validIn)
```

Description

`[Y,startOut,endOut,validOut,errOut] = RSDec(X,startIn,endIn,validIn)` decodes one encoded message symbol, `X`, and returns the decoded symbol `Y`. The `start` and `end` signals indicate the message frame boundaries. If `errOut` is `1` (`true`), then the object detected uncorrectable errors in the input frame.

`[Y,startOut,endOut,validOut,errOut,numErrors] = RSDec(X,startIn,endIn, validIn)` decodes the input data, and also returns the number of errors detected and corrected. To use this syntax, set the `NumErrorsOutputPort` property to `true`. If `errOut` is `1` (`true`), then the object detected uncorrectable errors in the output frame, and `numErrors` is invalid.

Input Arguments

`X` — Input message data or parity symbols

integer

Input message data and parity symbols, one symbol at a time, specified as an unsigned integer or `fi()` with any binary point scaling.

`double` type is allowed for simulation but not supported for HDL code generation.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startIn` — Start of input data frame
logical scalar

Start of input data frame, specified as a logical scalar.

Data Types: `logical`

`endIn` — End of input data frame
logical scalar

End of input data frame, specified as a logical scalar.

Data Types: `logical`

`validIn` — Validity of input data
logical scalar

Validity of input data, specified as a logical scalar.

Data Types: `logical`

Output Arguments

Y — Message data symbols
integer

Message data symbols, returned one symbol at a time, as an integer with the same data type as the input message, X.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startOut` — Start of output data frame
logical scalar

Start of output data frame, returned as a logical scalar.

Data Types: `logical`

`endOut` — End of output data frame
logical scalar

End of output data frame, returned as a logical scalar.

Data Types: `logical`

`validOut` — Validity of output data
logical scalar

Validity of output data, returned as a logical scalar.

Data Types: `logical`

`errOut` — Uncorrectable error in output data frame
logical scalar

Uncorrectable error in output data frame, returned as a logical scalar. This signal is 1 (`true`) when the message frame contains uncorrectable errors. In this case, the output data symbols are corrupted. This value is valid when `endOut` is 1 (`true`).

Data Types: `logical`

`numErrors` — Number of errors detected and corrected
integer

Number of errors detected and corrected, returned as an integer. This value is valid when `endOut` is 1 (`true`) and `errOut` is 0 (`false`).

Data Types: `uint8`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Reed-Solomon Coding and Error Detection for HDL

Encode and decode a signal using Reed Solomon encoder and decoder System objects. This example shows how to include each object in a function for HDL code generation.

Create a random message to encode. This message is smaller than the codeword length to show how the objects support shortened codes. Pad the message with zeros to accommodate the latency of the decoder, including the Chien search.

```
messageLength = 188;
dataIn = [randi([0,255],1,messageLength,'uint8') zeros(1,1024-messageLength)];
```

Write a function that creates and calls a `HDLRSEncoder` System object™ with an RS(255,239) code. This code is used in the IEEE® 802.16 Broadband Wireless Access standard. `B` is the starting power of the roots of the primitive polynomial. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function  [dataOut,startOut,endOut,validOut] = HDLRSEnc80216(dataIn,startIn,endIn,validIn)
%HDLRSEnc80216
% Processes one sample of data using the comm.HDLRSEncoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsEnc80216;
  if isempty(rsEnc80216)
    rsEnc80216 = comm.HDLRSEncoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut] = rsEnc80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to encode the message.

```
for ii = 1:1024
    messageStart = (ii==1);
    messageEnd = (ii==messageLength);
    validIn = (ii<=messageLength);
    [encOut(ii),startOut(ii),endOut(ii),validOut(ii)] = ...
        HDLRSEnc80216(dataIn(ii),messageStart,messageEnd,validIn);
end
```

```
rsEnc80216 = 

  comm.HDLRSEncoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
        PuncturePatternSource: 'None'
                      BSource: 'Property'
                            B: 0
```

Inject errors at random locations in the encoded message. Reed-Solomon can correct up to (*N* – *K*)/2 errors in each *N* symbols. So, in this example, the error correction capability is (255 – 239)/2=8 symbols.

```
numErrors = 8;
loc = randperm(messageLength,numErrors);
% encOut is qualified by validOut, use an offset for injecting errors
vi = find(validOut==true,1);
for i = 1:numErrors
   idx = loc(i)+vi;
   symbol = encOut(idx);
   encOut(idx) = randi([0 255],'uint8');
   fprintf('Symbol(%d): was 0x%x, now 0x%x\n',loc(i),symbol,encOut(idx))
end
```

```
Symbol(147): was 0x1f, now 0x82
Symbol(16): was 0x6b, now 0x82
Symbol(173): was 0x3, now 0xd1
Symbol(144): was 0x66, now 0xcb
Symbol(90): was 0x13, now 0xa4
Symbol(80): was 0x5a, now 0x60
Symbol(82): was 0x95, now 0xcf
Symbol(56): was 0xf5, now 0x88
```

Write a function that creates and calls a `HDLRSDecoder` System object™. This object must have the same code and polynomial as the encoder. You can generate HDL from this function.

```
function  [dataOut,startOut,endOut,validOut,err] = HDLRSDec80216(dataIn,startIn,endIn,validIn)
%HDLRSDec80216
% Processes one sample of data using the comm.HDLRSDecoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsDec80216;
  if isempty(rsDec80216)
    rsDec80216 = comm.HDLRSDecoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut,err] = rsDec80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to detect errors in the encoded message.

```
for ii = 1:1024
 [decOut(ii),decStartOut(ii),decEndOut(ii),decValidOut(ii),decErrOut(ii)] = ...
     HDLRSDec80216(encOut(ii),startOut(ii),endOut(ii),validOut(ii));
end
```

```
rsDec80216 = 

  comm.HDLRSDecoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
                      BSource: 'Property'
                            B: 0
          NumErrorsOutputPort: false
```

Select the valid decoder output and compare the decoded symbols to the original message.

```
decOut = decOut(decValidOut==1);
originalMessage = dataIn(1:messageLength);
if all(originalMessage==decOut)
    fprintf('All %d message symbols were correctly decoded.\n',messageLength)
else
   for jj = 1:messageLength
      if dataIn(jj)~=decOut(jj)
        fprintf('Error in decoded symbol(%d). Original 0x%x, Decoded 0x%x.\n',jj,dataIn(jj),decOu
      end
   end
end
```

```
All 188 message symbols were correctly decoded.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

For C/C++ code generation, these usage notes and limitations apply:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.HDLRSEncoder` | `comm.RSDecoder`

Blocks

Integer-Output RS Decoder HDL Optimized

Introduced in R2012b

comm.HDLRSEncoder

Package: `comm`

Encode message using Reed-Solomon encoder

Description

The HDL-optimized `HDLRSEncoder` System object creates a Reed-Solomon (RS) code with message and codeword lengths that you specify.

To encode a message using a Reed-Solomon code:

1. Create the `comm.HDLRSEncoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
RSEnc = comm.HDLRSEncoder
RSEnc = comm.HDLRSEncoder(Name,Value)
RSEnc = comm.HDLRSEncoder(N,K,Name,Value)
```

Description

`RSEnc = comm.HDLRSEncoder` creates an HDL-optimized block encoder System object, `RSEnc`, that performs Reed-Solomon encoding in a streaming fashion for HDL.

`RSEnc = comm.HDLRSEncoder(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes. For example,

```
comm.HDLRSEncoder('BSource','Property','B',2)
```

sets a starting power of 2 for the roots of the primitive polynomial.

`RSEnc = comm.HDLRSEncoder(N,K,Name,Value)` sets the `CodewordLength` property to `N`, the `MessageLength` property to `K`, and other specified property names to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

B — Starting power for roots of primitive polynomial

1 (default) | positive integer

Starting power for roots of the primitive polynomial, specified as a positive integer.

Dependencies

The object uses this value when you set `BSource` to `'Property'`.

BSource — Source of starting power for roots of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the starting power for roots of the primitive polynomial, specified as either `'Property'` or `'Auto'`. When you select `'Auto'`, the object uses `B = 1`.

CodewordLength — Number of symbols, N, in RS codeword

7 (default) | positive integer

Number of symbols, *N*, in the RS codeword, specified as a positive integer. This value is rounded up to 2^M–1. *M* is the degree of the primitive polynomial, as specified by the `PrimitivePolynomialSource` and `PrimitivePolynomial` properties. The difference of `CodewordLength` – `MessageLength` must be an even integer.

If the value of this property is less than 2^M–1, the object assumes a shortened RS code.

If you set `PrimitivePolynomialSource` to `'Auto'`, then `CodewordLength` must be in the range 3 < `CodewordLength` ≤ 2^{16} – 1.

If you set `PrimitivePolynomialSource` to `'Property'`, then `CodewordLength` must be in the range 3 ≤ `CodewordLength` ≤ 2^M– 1. *M* must be in the range 3 ≤ *M* ≤ 16.

MessageLength — Message length, K

3 (default) | positive integer

Message length, *K*, specified as a positive integer. The difference of `CodewordLength` – `MessageLength` must be an even integer.

PrimitivePolynomialSource — Source of primitive polynomial

`'Auto'` (default) | `'Property'`

Source of the primitive polynomial, specified as either `'Property'` or `'Auto'`.

- When you set this property to `'Auto'`, the object uses a primitive polynomial of degree *M* = `ceil`(**log2**(`CodewordLength`+1)), which is the result of `fliplr`(`de2bi`(`primpoly`(*M*))).
- When you set this property to `'Property'`, you must specify a polynomial using the `PrimitivePolynomial` property.

PrimitivePolynomial — Primitive polynomial

`[1 0 1 1]` (default) | binary row vector

Primitive polynomial, specified as a binary row vector that represents a primitive polynomial over `gf`(2) of degree *M*, in descending order of powers. The polynomial defines the finite field `gf`(2^M) corresponding to the integers that form messages and codewords.

Dependencies

The object uses this value when you set PrimitivePolynomialSource to 'Property'.

PuncturePatternSource — Source of puncture pattern

'None' (default) | 'Property'

Source of the puncture pattern, specified as 'None' or 'Property'. If you set this property to 'None', then the object does not apply puncturing to the code. If you set this property to 'Property', then the object punctures the code based on a puncture pattern vector specified in the PuncturePattern property.

PuncturePattern — Pattern used to puncture encoded data

[ones(2,1); zeros(2,1)] (default) | binary column vector

Pattern used to puncture the encoded data, specified as a double-precision, binary column vector with a length of CodewordLength – MessageLength. The default is [ones(2,1); zeros(2,1)]. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword.

Dependencies

This property applies when you set the PuncturePatternSource property to 'Property'.

Usage

Syntax

```
[Y,startOut,endOut,validOut] = RSEnc(X,startIn,endIn,validIn)
```

Description

[Y,startOut,endOut,validOut] = RSEnc(X,startIn,endIn,validIn) encodes one input message symbol, X, and returns one symbol of encoded data, Y. The start and end signals indicate the message frame boundaries. The object returns associated parity symbols at the end of each message frame.

Input Arguments

X — Input message symbol

integer

Input message data, one symbol at a time, specified as an unsigned integer or fi() with any binary point scaling. The word length of each symbol must be ceil($\log_2$(CodewordLength+1)).

double type is allowed for simulation but not supported for HDL code generation.

Data Types: double | uint8 | uint16 | uint32 | fi

startIn — Start of input data frame

logical scalar

Start of input data frame, specified as a logical scalar.

Data Types: logical

`endIn` — End of input data frame

logical scalar

End of input data frame, specified as a logical scalar.

Data Types: `logical`

`validIn` — Validity of input data

logical scalar

Validity of input data, specified as a logical scalar.

Data Types: `logical`

Output Arguments

Y — Output message data and parity symbols

integer

Message data and parity symbols, returned one symbol at a time, as an integer with the same data type as the input message, X.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fi`

`startOut` — Start of output data frame

logical scalar

Start of output data frame, returned as a logical scalar.

Data Types: `logical`

`endOut` — End of output data frame

logical scalar

End of output data frame, returned as a logical scalar.

Data Types: `logical`

`validOut` — Validity of output data

logical scalar

Validity of output data, returned as a logical scalar.

Data Types: `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics

reset	Reset internal states of System object

Examples

Reed-Solomon Coding and Error Detection for HDL

Encode and decode a signal using Reed Solomon encoder and decoder System objects. This example shows how to include each object in a function for HDL code generation.

Create a random message to encode. This message is smaller than the codeword length to show how the objects support shortened codes. Pad the message with zeros to accommodate the latency of the decoder, including the Chien search.

```
messageLength = 188;
dataIn = [randi([0,255],1,messageLength,'uint8') zeros(1,1024-messageLength)];
```

Write a function that creates and calls a `HDLRSEncoder` System object™ with an RS(255,239) code. This code is used in the IEEE® 802.16 Broadband Wireless Access standard. `B` is the starting power of the roots of the primitive polynomial. You can generate HDL from this function.

Note: This object syntax runs only in R2016b or later. If you are using an earlier release, replace each call of an object with the equivalent `step` syntax. For example, replace `myObject(x)` with `step(myObject,x)`.

```
function  [dataOut,startOut,endOut,validOut] = HDLRSEnc80216(dataIn,startIn,endIn,validIn)
%HDLRSEnc80216
% Processes one sample of data using the comm.HDLRSEncoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsEnc80216;
  if isempty(rsEnc80216)
    rsEnc80216 = comm.HDLRSEncoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut] = rsEnc80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to encode the message.

```
for ii = 1:1024
    messageStart = (ii==1);
    messageEnd = (ii==messageLength);
    validIn = (ii<=messageLength);
    [encOut(ii),startOut(ii),endOut(ii),validOut(ii)] = ...
        HDLRSEnc80216(dataIn(ii),messageStart,messageEnd,validIn);
end
```

```
rsEnc80216 =

  comm.HDLRSEncoder with properties:
```

```
               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
        PuncturePatternSource: 'None'
                      BSource: 'Property'
                            B: 0
```

Inject errors at random locations in the encoded message. Reed-Solomon can correct up to (*N* – *K*)/2 errors in each *N* symbols. So, in this example, the error correction capability is (255 – 239)/2=8 symbols.

```
numErrors = 8;
loc = randperm(messageLength,numErrors);
% encOut is qualified by validOut, use an offset for injecting errors
vi = find(validOut==true,1);
for i = 1:numErrors
   idx = loc(i)+vi;
   symbol = encOut(idx);
   encOut(idx) = randi([0 255],'uint8');
   fprintf('Symbol(%d): was 0x%x, now 0x%x\n',loc(i),symbol,encOut(idx))
end
```

```
Symbol(147): was 0x1f, now 0x82
Symbol(16): was 0x6b, now 0x82
Symbol(173): was 0x3, now 0xd1
Symbol(144): was 0x66, now 0xcb
Symbol(90): was 0x13, now 0xa4
Symbol(80): was 0x5a, now 0x60
Symbol(82): was 0x95, now 0xcf
Symbol(56): was 0xf5, now 0x88
```

Write a function that creates and calls a `HDLRSDecoder` System object™. This object must have the same code and polynomial as the encoder. You can generate HDL from this function.

```
function  [dataOut,startOut,endOut,validOut,err] = HDLRSDec80216(dataIn,startIn,endIn,validIn)
%HDLRSDec80216
% Processes one sample of data using the comm.HDLRSDecoder System object(TM)
% dataIn is a uint8 scalar, representing 8 bits of binary data.
% startIn, endIn, and validIn are logical scalar values.
% You can generate HDL code from this function.

  persistent rsDec80216;
  if isempty(rsDec80216)
    rsDec80216 = comm.HDLRSDecoder(255,239,'BSource','Property','B',0)
  end
  [dataOut,startOut,endOut,validOut,err] = rsDec80216(dataIn,startIn,endIn,validIn);
end
```

Call the function to detect errors in the encoded message.

```
for ii = 1:1024
 [decOut(ii),decStartOut(ii),decEndOut(ii),decValidOut(ii),decErrOut(ii)] = ...
     HDLRSDec80216(encOut(ii),startOut(ii),endOut(ii),validOut(ii));
end
```

```
rsDec80216 = 

  comm.HDLRSDecoder with properties:

               CodewordLength: 255
                MessageLength: 239
    PrimitivePolynomialSource: 'Auto'
                      BSource: 'Property'
                            B: 0
          NumErrorsOutputPort: false
```

Select the valid decoder output and compare the decoded symbols to the original message.

```
decOut = decOut(decValidOut==1);
originalMessage = dataIn(1:messageLength);
if all(originalMessage==decOut)
    fprintf('All %d message symbols were correctly decoded.\n',messageLength)
else
   for jj = 1:messageLength
      if dataIn(jj)~=decOut(jj)
        fprintf('Error in decoded symbol(%d). Original 0x%x, Decoded 0x%x.\n',jj,dataIn(jj),decOu
      end
   end
end
```

```
All 188 message symbols were correctly decoded.
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.HDLRSDecoder` | `comm.RSEncoder`

Blocks

Integer-Input RS Encoder HDL Optimized

Introduced in R2012b

comm.HelicalDeinterleaver

Package: comm

Restore ordering of symbols using helical array

Description

The HelicalDeinterleaver object permutes the symbols in the input signal by placing them in a row-by-row array and then selecting groups helically to send to the output port.

To helically deinterleave input symbols:

1. Define and set up your helical deinterleaver object. See "Construction" on page 3-860.
2. Call step to deinterleave input symbols according to the properties of comm.HelicalDeinterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.HelicalDeinterleaver creates a helical deinterleaver System object, H. This object restores the original ordering of a sequence that was interleaved using the helical interleaver System object.

H = comm.HelicalDeinterleaver(Name,Value) creates a helical deinterleaver object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

NumColumns

Number of columns in helical array

Specify the number of columns in the helical array as a positive integer scalar value. The default is 6.

GroupSize

Size of each group of input symbols

Specify the size of each group of input symbols as a positive integer scalar value. The default is 4.

StepSize

Helical array step size

Specify number of rows of separation between consecutive input groups in their respective columns of the helical array. This property requires a positive integer scalar value. The default is `1`.

`InitialConditions`

Initial conditions of helical array

Specify the value that is initially stored in the helical array as a numeric scalar value. The default is `0`.

Methods

reset	Reset states of the helical deinterleaver object
step	Restore ordering of symbols using a helical array

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Helical Interleaving and Deinterleaving

Create helical interleaver and deinterleaver objects.

```
interleaver = comm.HelicalInterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
deinterleaver = comm.HelicalDeinterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
```

Generate random data. Interleave and then deinterleave the data.

```
[dataIn,dataOut] = deal([]);

for k = 1:10
    data = randi(7,6,1);
    intData = interleaver(data);
    deIntData = deinterleaver(intData);

    dataIn = cat(1,dataIn,data);
    dataOut = cat(1,dataOut,deIntData);
end
```

Determine the delay through the interleaver and deinterleaver pair.

```
intlvDelay = finddelay(dataIn,dataOut)
```

```
intlvDelay = 6
```

After taking the interleaving delay into account, confirm that the original and deinterleaved data are identical.

```
isequal(dataIn(1:end-intlvDelay),dataOut(1+intlvDelay:end))
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Helical Deinterleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HelicalInterleaver` | `comm.MultiplexedDeinterleaver`

Introduced in R2012a

reset

System object: `comm.HelicalDeinterleaver`
Package: `comm`

Reset states of the helical deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `HelicalDeinterleaver` object, `H`.

step

System object: `comm.HelicalDeinterleaver`
Package: `comm`

Restore ordering of symbols using a helical array

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a helical interleaver and returns `Y`. The input `X` must be a column vector. The data type must be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The helical deinterleaver object uses an array for its computations. If you set the NumColumns property of the object to *C*, then the array has *C* columns and unlimited rows. If you set the GroupSize property to *N*, then the object accepts an input of length *C*×*N* and inserts it into the next *N* rows of the array. The object also places the value of the InitialConditions property into certain positions in the top few rows of the array. This accommodates the helical pattern and also preserves the vector indices of symbols that pass through the `HelicalInterleaver` and `HelicalDeinterleaver` objects. The output consists of consecutive groups of *N* symbols. The object selects the *k*-th output group in the array from column *k* mod *C*. This selection is of type helical because of the reduction modulo *C* and because the first symbol in the *k*-th group is in row 1+(*k*-1)×*s*, where *s* is the value for the StepSize property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.HelicalInterleaver

Package: comm

Permute input symbols using helical array

Description

The HelicalInterleaver object permutes the symbols in the input signal by placing them in an array in a helical arrangement and then sending rows of the array to the output port.

To helically interleave input symbols:

1. Define and set up your helical interleaver object. See "Construction" on page 3-865.
2. Call step to interleave input symbols according to the properties of comm.HelicalInterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.HelicalInterleaver creates a helical interleaver System object, H. This object permutes the input symbols in the input signal by placing them in an array in a helical arrangement.

H = comm.HelicalInterleaver(Name,Value) creates a helical interleaver object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

NumColumns

Number of columns in helical array

Specify the number of columns in the helical array as a positive integer scalar value. The default is 6.

GroupSize

Size of each group of input symbols

Specify the size of each group of input symbols as a positive integer scalar value. The default is 4.

StepSize

Helical array step size

Specify the number of rows of separation between consecutive input groups in their respective columns of the helical array. This property requires as a positive integer scalar value . The default is 1.

`InitialConditions`

Initial conditions of helical array

Specify the value that is initially stored in the helical array as a numeric scalar value. The default is `0`.

Methods

reset	Reset states of the helical interleaver object
step	Permute input symbols using a helical array

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Helical Interleaving and Deinterleaving

Create helical interleaver and deinterleaver objects.

```
interleaver = comm.HelicalInterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
deinterleaver = comm.HelicalDeinterleaver('GroupSize',2,'NumColumns',3, ...
    'InitialConditions',-1);
```

Generate random data. Interleave and then deinterleave the data.

```
[dataIn,dataOut] = deal([]);

for k = 1:10
    data = randi(7,6,1);
    intData = interleaver(data);
    deIntData = deinterleaver(intData);

    dataIn = cat(1,dataIn,data);
    dataOut = cat(1,dataOut,deIntData);
end
```

Determine the delay through the interleaver and deinterleaver pair.

```
intlvDelay = finddelay(dataIn,dataOut)
```

```
intlvDelay = 6
```

After taking the interleaving delay into account, confirm that the original and deinterleaved data are identical.

```
isequal(dataIn(1:end-intlvDelay),dataOut(1+intlvDelay:end))
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Helical Interleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HelicalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: `comm.HelicalInterleaver`
Package: `comm`

Reset states of the helical interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `HelicalInterleaver` object, `H`.

step

System object: `comm.HelicalInterleaver`
Package: `comm`

Permute input symbols using a helical array

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector. The data type must be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The helical interleaver object places the elements of `X` in an array in a helical fashion. If you set the NumColumns property of the object to *C*, then the array has *C* columns and unlimited rows. If you set the GroupSize property to *N*, then the object accepts an input of length *C*×*N* and partitions the input into consecutive groups of *N* symbols. The object places the *k*-th group in the array along column *k* mod *C*. This placement is of type helical because of the reduction modulo C and because the first symbol in the *k*-th group is in the row 1+(*k*-1)×*s*, where *s* is the value for the StepSize property. Positions in the array that do not contain input symbols have default contents specified by the `InitialConditions` property. The object outputs *C*×*N* symbols from the array by reading the next *N* rows sequentially.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.IntegerToBit

Package: `comm`

(To be removed) Convert vector of integers to vector of bits

Note will be removed in a future release. Use `de2bi` instead. For more information, see "Compatibility Considerations".

Description

The `IntegerToBit` object maps each integer (or fixed-point value) in the input vector to a group of bits in the output vector.

To map integers to bits:

1 Define and set up your integer to bit object. See "Construction" on page 3-870.

2 Call `step` to map integers in the input vector to groups of bits in the output vector according to the properties of `comm.IntegerToBit`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.IntegerToBit` creates an integer-to-bit converter System object, `H`. This object maps a vector of integer-valued or fixed-point inputs to a vector of bits.

`H = comm.IntegerToBit(Name,Value)` creates an integer-to-bit converter object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.IntegerToBit(NUMBITS,Name,Value)` creates an integer-to-bit converter object, `H`. This object has the `BitsPerInteger` property set to `NUMBITS` and the other specified properties set to the specified values.

Properties

BitsPerInteger

Number of bits per integer

Specify the number of bits the System object uses to represent each input integer. You must set this property to a scalar integer between `1` and `32`. The default is `3`.

MSBFirst

Output bit words with first bit as most significant bit

Set this property to true to indicate that the first bit of the output bit words is the most significant bit (MSB). The default is `true`. Set this property to `false` to indicate that the first bit of the output bit words is the least significant bit (LSB).

SignedIntegerInput

Assume inputs are signed integers

Set this property to true if the integer inputs are signed. The default is `false`. Set this property to `false` if the integer inputs are unsigned. If the `SignedIntegerInput` on page 3-0 property is `false`, the input values must be between 0 and (2^N)–1. In this case, *N* is the value you specified in the `BitsPerInteger` on page 3-0 property. When you set this property to `true`, the input values must be between –($2^{(N-1)}$) and ($2^{(N-1)}$)–1.

OutputDataType

Data type of output

Specify output data type as one of `Full precision` | `Smallest unsigned integer` | `Same as input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `logical`. The default is `Full precision`.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When you set this property to `Full precision`, the object determines the output data type based on the input data type. If the input data type is double- or single-precision, the output data has the same data type as the input data. Otherwise, the output data type is determined in the same way as when you set this property to `Smallest unsigned integer`.

When you set this property to `Same as input`, and the input data type is numeric or fixed-point integer (fi object), the output data has the same data type as the input data.

Methods

step	(To be removed) Convert vector of integers to vector of bits

Common to All System Objects	
`release`	Allow System object property value changes

Examples

```
hIntToBit = comm.IntegerToBit(4);
intData = randi([0 2^hIntToBit.BitsPerInteger-1],3,1);
bitData = step(hIntToBit,intData)
```

```
bitData = 12×1

     1
     1
     0
     1
     1
     1
     1
     0
     0
     0
       ⋮
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Integer to Bit Converter block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.IntegerToBit` will be removed in a future release. Use `de2bi` instead.

Warns starting in R2020b

Use `de2bi` instead of comm.IntegerToBit. Data types as supported by comm.IntegerToBit are not inherently supported by functions. This code shows decimal to binary conversion for various data types using this function.

Examples Using de2bi for Various Data Types

```
function compde2bi

% Number of integers
n = randi([1 100]);

% Default
h1 = comm.IntegerToBit;
bpi = h1.BitsPerInteger;
x = randi([0,2^bpi-1],n,1);
y1 = h1(x);
y2 = reshape(de2bi(x,bpi,'left-msb')',[],1);
isequal(y1,y2)

% Right MSB, logical input
h2 = comm.IntegerToBit( ...
    'BitsPerInteger',5, ...
    'MSBFirst',false);
bpi = h2.BitsPerInteger;
x = randi([0,2^bpi-1],n,1);
y1 = h2(x);
y2 = reshape(de2bi(x,bpi,'right-msb')',[],1);
isequal(y1,y2)

% Right MSB, signed input, single input
h3 = comm.IntegerToBit( ...
    'BitsPerInteger',8, ...
    'MSBFirst',false, ...
    'SignedIntegerInput',true);
bpi = h3.BitsPerInteger;
N = 2^bpi;
x = randi([-N/2,N/2-1],n,1);
y1 = h3(x);
y2 = reshape(de2bi(x+(x<0)*N,bpi,'right-msb')',[],1);
isequal(y1,y2)
end
```

See Also

`bi2de` | `de2bi` | `dec2bin`

Introduced in R2012a

step

System object: `comm.IntegerToBit`
Package: `comm`

(To be removed) Convert vector of integers to vector of bits

Note `comm.IntegerToBit` will be removed in a future release. Use `de2bi` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` converts integer input, `X`, to corresponding bits, `Y`. The input must be scalar or a column vector and the data type can be numeric or fixed-point (fi objects). The output is a column vector with length equal to length(`X`)×*N*, where *N* is the value of the `BitsPerInteger` property. If any input value is outside the range of *N*, the object issues an error. If the `SignedIntegerInput` property is `false`, the input values must be between `0` and (2^{N})-1. If you set the `SignedIntegerInput` property to `true`, the input values must be between -($2^{(N-1)}$) and ($2^{(N-1)}$)-1.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.IntegrateAndDumpFilter

Package: `comm`

Integrate discrete-time signal with periodic resets

Description

The `IntegrateAndDumpFilter` object creates a cumulative sum of the discrete-time input signal, while resetting the sum to zero according to a fixed schedule. When the simulation begins, the object discards the number of samples specified in the Offset property. After this initial period, the object sums the input signal along columns and resets the sum to zero every *N*input samples, set by the integration period property. The reset occurs after the object produces output at that time step.

To integrate discrete-time signals with periodic resets:

1. Define and set up your integrate and dump filter object. See “Construction” on page 3-875.
2. Call `step` to integrate discrete-time signals according to the properties of `comm.IntegrateAndDumpFilter`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.IntegrateAndDumpFilter` creates an integrate and dump filter System object, `H`. this object integrates over a number of samples in an integration period, and then resets at the end of that period.

`H = comm.IntegrateAndDumpFilter(Name,Value)` creates an integrate and dump filter object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.IntegrateAndDumpFilter(PERIOD,Name,Value)` creates an integrate and dump filter object, `H`. This object has the `IntegrationPeriod` property set to `PERIOD` and the other specified properties set to the specified values.

Properties

`IntegrationPeriod`

Integration period

Specify the integration period, in samples, as a positive, integer scalar value greater than `1`. The integration period defines the length of the sample blocks that the object integrates between resets. The default is `8`.

`Offset`

Number of offset samples

Specify a nonnegative, integer vector or scalar specifying the number of input samples that the object discards from each column of input data at the beginning of data processing. Discarding begins when you call the `step` method for the first time. The default is `0`.

When you set the `Offset` on page 3-0 property to a nonzero value, the object outputs one or more zeros during the initial period while discarding input samples.

When you specify this property as a vector of length *L*, the *i*-th element of the vector corresponds to the offset for the *i*-th column of the input data matrix, which has *L* columns.

When you specify this property as a scalar value, the object applies the same offset to each column of the input data matrix. The offset creates a transient effect, rather than a persistent delay.

`DecimateOutput`

Decimate output

Specify whether the `step` method returns intermediate cumulative sum results or decimates intermediate results. The default is `true`.

When you set this property to `true`, the `step` method returns one output sample, consisting of the final integration value, for each block of `IntegrationPeriod` on page 3-0 input samples. If the inputs are (*K*×`IntegrationPeriod`)×*L* matrices, then the outputs are *K*×*L* matrices.

When you set this property to `false`, the `step` method returns `IntegrationPeriod` output samples, comprising the intermediate cumulative sum values, for each block of `IntegrationPeriod` input samples. In this case, inputs and outputs have the same dimensions.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects” on page 3-879.

`RoundingMethod`

Rounding of fixed-point numeric values

Specify the rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies only if the object is not in full precision mode.

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`. This property applies only if the object is not in full precision mode.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as one of `Full precision` | `Same as input` | `Custom`. The default is `Full precision`. When you set this property to `Full precision` the object automatically calculates the accumulator output word and fraction lengths. Set this property to `Custom` to specify the accumulator data type using the `CustomAccumulatorDataType` on page 3-0 property. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false.

CustomAccumulatorDataType

Fixed-point data type of accumulator

Specify the accumulator fixed-point type as a scaled `numerictype` object with a signedness of `Auto`. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `AccumulatorDataType` on page 3-0 property to `Custom`.

OutputDataType

Data type of output

Specify the output fixed-point type as one of `Same as accumulator` | `Same as input` | `Custom`. The default is `Same as accumulator`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false.

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a scaled `numerictype` object with a signedness of `Auto`. The default is `numerictype([],32,30)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `OutputDataType` on page 3-0 property to `Custom`.

Methods

step	Integrate discrete-time signal with periodic resets

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Pass Noisy Pulses Through Integrate and Dump Filter

Create an integrate and dump filter having an integration period of 20 samples.

```
intdump = comm.IntegrateAndDumpFilter(20);
```

Generate binary data.

```
d = randi([0 1],50,1);
```

Upsample the data, and pass it through an AWGN channel.

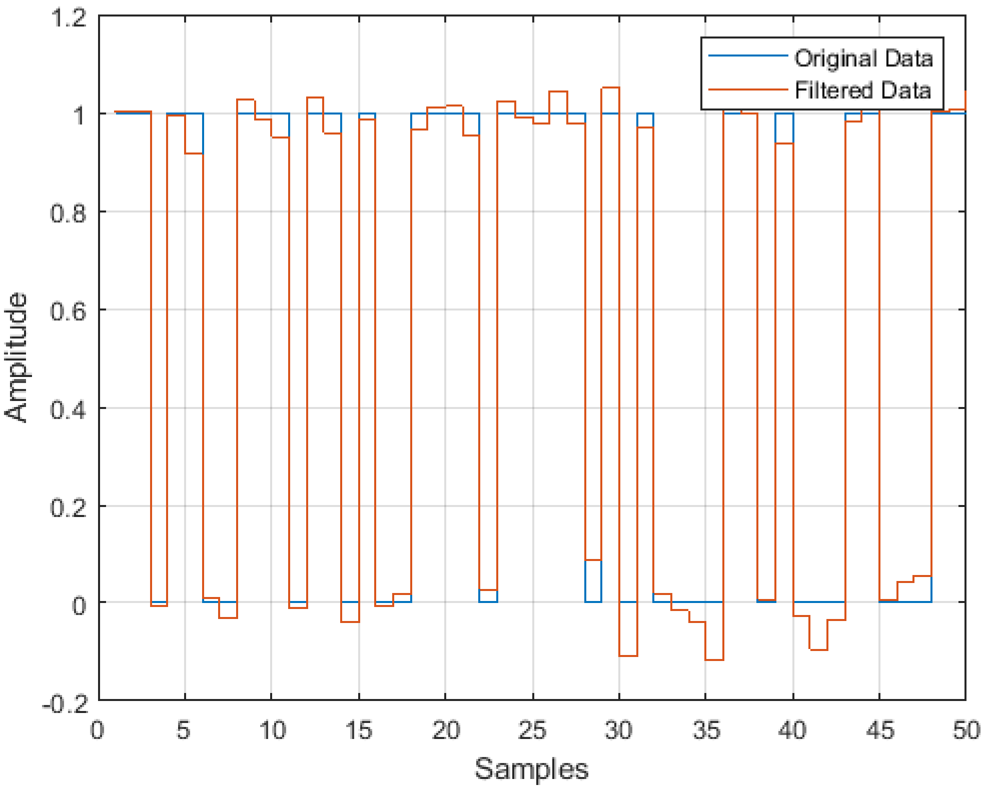
```
x = upsample(d,20);
y = awgn(x,25,'measured');
```

Pass the noisy data through the filter.

```
z = intdump(y);
```

Plot the original and filtered data. The integrate and dump filter removes most of the noise effects.

```
stairs([d z])
legend('Original Data','Filtered Data')
xlabel('Samples')
ylabel('Amplitude')
grid
```



More About

Full Precision for Fixed-Point System Objects

`FullPrecisionOverride` is a convenience property that, when you set to `true`, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input.

For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Full precision for System objects does not optimize coefficient values. When you set the `FullPrecisionOverride` property to `true`, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set `FullPrecisionOverride` to `false`.

Algorithms

This object implements the algorithm, inputs, and outputs described on the Integrate and Dump block reference page. The object properties correspond to the block parameters, except:
The **Output intermediate values** parameter corresponds to the `DecimateOutput` on page 3-0 property.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

Introduced in R2012a

step

System object: `comm.IntegrateAndDumpFilter`
Package: `comm`

Integrate discrete-time signal with periodic resets

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` periodically integrates blocks of N samples from the input data, `X`, and returns the result in `Y`. N is the number of samples that you specify in the `IntegrationPeriod` property. `X` is a column vector or a matrix and the data type is double, single or fixed-point (fi objects). `X` must have K*N rows for some positive integer K, with one or more columns. The object treats each column as an independent channel with integration occurring along every column. The dimensions of output `Y` depend on the value you set for the `DecimateOutput` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.KasamiSequence

Package: `comm`

Generate Kasami sequence

Description

The `KasamiSequence` object generates a sequence from the set of Kasami sequences. The Kasami sequences are a set of sequences that have good cross-correlation properties.

To generate a Kasami sequence:

1. Define and set up your Kasami sequence object. See “Construction” on page 3-881.
2. Call `step` to generate a Kasami sequence according to the properties of `comm.KasamiSequence`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

Construction

`H = comm.KasamiSequence` creates a KasamiSequence System object, `H`. This object generates a Kasami sequence.

`H = comm.KasamiSequence(Name,Value)` creates a Kasami sequence generator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Polynomial`

Generator polynomial

Specify the polynomial that determines the shift register's feedback connections. The default is `'z^6 + z + 1'`.

You can specify the generator polynomial as a character vector or as a binary numeric vector that lists the coefficients of the polynomial in descending order of powers. The first and last elements must equal `1`. Specify the length of this vector as n+1, where n is the degree of the generator polynomial and must be even.

Lastly, you can specify the generator polynomial as a vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`. For example, `[1 0 0 0 0 0 1 0 1]` and `[8 2 0]` represent the same polynomial, $g(z) = z^8 + z^2 + 1$.

InitialConditions

Initial conditions of shift register

Specify the initial values of the shift register as a binary numeric scalar or as binary numeric vector. The default is `[0 0 0 0 0 1]`. Set the vector length equal to the degree of the generator polynomial.

When you set this property to a vector value, each element of the vector corresponds to the initial value of the corresponding cell in the shift register.

When you set this property to a scalar value, that value specifies the initial conditions of all the cells of the shift register. The scalar, or at least one element of the specified vector, requires a nonzero value for the object to generate a nonzero sequence.

Index

Sequence index

Specify the index to select a Kasami sequence of interest from the set of possible sequences. The default is `0`. Kasami sequences have a period equal to $N = 2^n - 1$, where n indicates a nonnegative, even integer equal to the degree of the generator polynomial that you specify in the `Polynomial` on page 3-0 property.

There are two classes of Kasami sequences: those obtained from a small set and those obtained from a large set. You choose a Kasami sequence from the small set by setting this property to a numeric, scalar, integer value in the range $[0 \ldots 2^{n/2}-2]$. You choose a sequence from the large set by setting this property to a numeric 1×2 integer vector $[k\ m]$ for k in $[-2, \ldots, 2^n-2]$, and m in $[-1, \ldots, 2^{n/2}-2]$.

Shift

Sequence offset from initial time

Specify the offset of the Kasami sequence from its starting point as a numeric, integer scalar value that can be positive or negative. The default is `0`. The Kasami sequence has a period of $N = 2^n-1$, where n is the degree of the generator polynomial that you specify in the `Polynomial` on page 3-0 property. The shift value is wrapped with respect to the sequence period.

VariableSizeOutput

Enable variable-size outputs

Set this property to true to enable an additional input to the step method. The default is false. When you set this property to true, the enabled input specifies the output size of the Kasami sequence used for the step. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to false, the `SamplesPerFrame` property specifies the number of output samples.

MaximumOutputSize

Maximum output size

Specify the maximum output size of the Kasami sequence as a positive integer 2-element row vector. The second element of the vector must be 1. The default is [10 1].

This property applies when you set the VariableSizeOutput property to true.

SamplesPerFrame

Number of output samples per frame

Specify the number of Kasami sequence samples that the step method outputs as a numeric, positive, integer scalar value . The default value is 1.

When you set this property to a value of *M*, then the step method outputs *M* samples of a Kasami sequence that has a period of $N = 2^n–1$. The value *n* equals the degree of the generator polynomial that you specify in the Polynomial on page 3-0 property.

ResetInputPort

Enable generator reset input

Set this property to true to enable an additional input to the step method. The default is false. The additional input resets the states of the Kasami sequence generator to the initial conditions that you specify in the InitialConditions on page 3-0 property.

OutputDataType

Data type of output

Specify the output data type as one of double | logical. The default is double.

Methods

reset	Reset states of Kasami sequence generator object
step	Generate a Kasami sequence

Common to All System Objects	
release	Allow System object property value changes

Examples

Spread BPSK Data with a Kasami Sequence

Spread BPSK data with a Kasami sequence of length 255 by using the Kasami sequence System object.

Generate binary data and apply BPSK modulation.

```
data = randi([0 1],10,1);
modData = pskmod(data,2);
```

Create a Kasami sequence object of length 255 using generator polynomial $x^8 + x^7 + x^4 + 1$.

```
kasamiSequence = comm.KasamiSequence('Polynomial',[8 7 4 0], ...
    'InitialConditions',[0 0 0 0 0 0 0 1],'SamplesPerFrame',255);
```

Generate the Kasami sequence and convert it to bipolar form.

```
kasSeq = kasamiSequence();
kasSeq = 2*kasSeq - 1;
```

Apply a gain of $1/\sqrt{255}$ to ensure that the spreading operation does not increase the overall signal power.

```
kasSeq = kasSeq/sqrt(255);
```

Spread the BPSK data using the Kasami sequence.

```
spreadData = modData*kasSeq';
spreadData = spreadData(:);
```

Verify that the spread data sequence is 255 times longer than the input data sequence.

```
spreadingFactor = length(spreadData)/length(data)
```

```
spreadingFactor = 255
```

Verify that the spreading operation did not increase the signal power.

```
spreadSigPwr = sum(abs(spreadData).^2)/length(data)
```

```
spreadSigPwr = 1.0000
```

Change the generator polynomial of the Kasami sequence generator to $x^8 + x^3 + 1$ after first releasing the object. Use the character representation of the polynomial.

```
release(kasamiSequence)
kasamiSequence.Polynomial = 'x^8 + x^3 + 1';
```

Generate a new sequence and convert it to bipolar form.

```
kasSeq = kasamiSequence();
kasSeq = 2*kasSeq - 1;
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Kasami Sequence Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GoldSequence` | `comm.PNSequence`

reset

System object: `comm.KasamiSequence`
Package: `comm`

Reset states of Kasami sequence generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `KasamiSequence` object, `H`.

step

System object: `comm.KasamiSequence`
Package: `comm`

Generate a Kasami sequence

Syntax

```
Y = step(H)
Y = step(H,RESET)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(H)` outputs a frame of the Kasami sequence in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Kasami sequence has a period of $N = 2^n$-1, where n is the degree of the generator polynomial that you specify in the `Polynomial` property.

`Y = step(H,RESET)` uses `RESET` as the reset signal when you set the `ResetInputPort` property to `true`. The data type of the `RESET` input must be double precision or logical. `RESET` can be a scalar value or a column vector with a length equal to the number of samples per frame that you specify in the `SamplesPerFrame` property. When the `RESET` input is a non-zero scalar, the object resets to the initial conditions that you specify in the `InitialConditions` property. It then generates a new output frame. A column vector `RESET` input allows multiple resets within an output frame. A non-zero value at the *i*-th element of the vector causes a reset at the *i*-th output sample time.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.LDPCDecoder

Package: comm

Decode binary low-density parity-check (LDPC) code

Description

The comm.LDPCDecoder System object uses the belief propagation algorithm to decode a binary LDPC code, which is input to the object as the soft-decision output (log-likelihood ratio of received bits) from demodulation. The object decodes generic binary LDPC codes where no patterns in the parity-check matrix are assumed. For more information, see “Belief Propagation Decoding” on page 3-893.

To decode an LDPC-encoded signal:

1 Create the comm.LDPCDecoder object and set its properties.
2 Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
ldpcdecoder = comm.LDPCDecoder
ldpcdecoder = comm.LDPCDecoder(parity)
ldpcdecoder = comm.LDPCDecoder( ___ ,Name,Value)
```

Description

ldpcdecoder = comm.LDPCDecoder creates a binary LDPC decoder System object. This object performs LDPC decoding based on the specified parity-check matrix.

ldpcdecoder = comm.LDPCDecoder(parity) sets the ParityCheckMatrix property to parity and creates an LDPC decoder System object. The parity input must be specified as described by the ParityCheckMatrix property.

ldpcdecoder = comm.LDPCDecoder(___ ,Name,Value) sets properties using one or more name-value pairs, in addition to inputs from any of the prior syntaxes. For example, comm.LDPCDecoder('DecisionMethod','Soft decision') configures an LDPC decoder System object to decode data using the soft-decision method and output log-likelihood ratios of data type double. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`ParityCheckMatrix` — Parity-check matrix
`dvbs2ldpc(1/2)` (default) | sparse binary matrix | nonsparse index matrix

Parity-check matrix, specified as a sparse (*N* – *K*)-by-*N* binary-valued matrix. *N* is the length of the received signal and must be in the range (0, 2^{31}). *K* is the length of the uncoded message and must be less than *N*. The last (*N* – *K*) columns in the parity-check matrix must be an invertible matrix in the Galois field of order 2, `gf`(2).

You can also specify the parity-check matrix as a two-column nonsparse index matrix, `I`, that defines the row and column indices of the `1`s in the parity-check matrix such that `sparse(I(:,1),I(:,2),1)`.

This property accepts numeric data types. When you set this property to a sparse binary matrix, this property also accepts the `logical` data type.

The default value uses the `dvbs2ldpc` function to configure a sparse parity-check matrix for half-rate LDPC coding, as specified in the DVB-S.2 standard.

Example: `dvbs2ldpc(R,'indices')` configures the index matrix for the DVB-S.2 standard, where `R` is the code rate, and `'indices'` specifies the output format of `dvbs2ldpc` as a two-column double-precision matrix that defines the row and column indices of the `1`s in the parity-check matrix.

Data Types: `double` | `logical`

`OutputValue` — Output value format
`'Information part'` (default) | `'Whole codeword'`

Output value format, specified as one of these values:

- `'Information part'` — The object outputs a *K*-by-1 column vector containing only the information-part of the received log-likelihood ratio vector. *K* is the length of the uncoded message.
- `'Whole codeword'` — The object outputs an *N*-by-1 column vector containing the whole log-likelihood ratio vector. *N* is the length of the received signal.

  N and *K* must align with the dimension of the (*N*–*K*)-by-*K* parity-check matrix.

Data Types: `char`

`DecisionMethod` — Decision method
`'Hard decision'` (default) | `'Soft decision'`

Decision method used for decoding, specified as one of these values:

- `'Hard decision'` — The object outputs decoded data of data type `logical`.
- `'Soft decision'` — The object outputs log-likelihood ratios of data type `double`.

Data Types: `char`

`IterationTerminationCondition` — Condition for iteration termination
`'Maximum iteration count'` (default) | `'Parity check satisfied'`

Condition for iteration termination, specified as one of these values:

- `'Maximum iteration count'` — Decoding terminates after the number of iterations specified by the `MaximumIterationCount` property.
- `'Parity check satisfied'` — Decoding terminates after all parity checks are satisfied. If not all parity checks are satisfied, decoding terminates after the number of iterations specified by the `MaximumIterationCount` property.

Data Types: `char`

`MaximumIterationCount` — Maximum number of decoding iterations
`50` (default) | positive integer

Maximum number of decoding iterations, specified as a positive integer.

Data Types: `double`

`NumIterationsOutputPort` — Output number of iterations executed
`false` (default) | `true`

Output number of iterations performed, specified as `false` or `true`. To output the number of iterations executed, set this property to `true`.

Data Types: `logical`

`FinalParityChecksOutputPort` — Output final parity checks
`false` (default) | `true`

Output final parity checks, specified as `false` or `true`. To output the final calculated parity checks, set this property to `true`.

Data Types: `logical`

Usage

Syntax

```
y = ldpcdecoder(x)
[y,numiter] = ldpcdecoder(x)
[y,parity] = ldpcdecoder(x)
[y,numiter,parity] = ldpcdecoder(x)
```

Description

`y = ldpcdecoder(x)` decodes input data using an LDPC code based on the default parity-check matrix.

`[y,numiter] = ldpcdecoder(x)` returns the decoded data, `y`, and number of iterations performed, `numiter`. To use this syntax, set the `NumIterationsOutputPort` property to `true`.

`[y,parity] = ldpcdecoder(x)` returns the decoded data, `y`, and final parity checks, `parity`. To use this syntax, set the `FinalParityChecksOutputPort` property to `true`.

`[y,numiter,parity] = ldpcdecoder(x)` returns the decoded data, number of iterations performed, and final parity checks. To use this syntax, set the `NumIterationsOutputPort` and `FinalParityChecksOutputPort` properties to `true`.

Input Arguments

x — Log-likelihood ratios

column vector

Log-likelihood ratios, specified as an *N*-by-1 column vector containing the soft-decision output from demodulation. *N* is the number of bits in the LDPC codeword before modulation. Each element is the log-likelihood ratio for a received bit. Element values are more likely to be `0` if the log-likelihood ratio is positive. The first *K* elements correspond to the information-part of the input message.

Data Types: `double`

Output Arguments

y — Decoded data

column vector

Decoded data, returned as a column vector. The `DecisionMethod` property specifies whether the object outputs hard decisions or soft decisions (log-likelihood ratios).

- If the `OutputValue` property is set to `'Information part'`, the output includes only the information-part of the received log-likelihood ratio vector.
- If the `OutputValue` property is set to `'Whole codeword'`, the output includes the whole log-likelihood ratio vector.

Data Types: `double` | `logical`

`numiter` — Number of executed decoding iterations

positive integer

Number of executed decoding iterations, returned as a positive integer.

Dependencies

To enable this output, set the `NumIterationsOutputPort` property to `true`.

`parity` — Final parity checks

column vector

Final parity checks after decoding the input LDPC code, returned as an (*N*-*K*)-by-1 column vector. *N* is the number of bits in the LDPC codeword before modulation. *K* is the length of the uncoded message.

Dependencies

To enable this output, set the `FinalParityChecksOutputPort` property to `true`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

LDPC Encode and Decode QPSK-Modulated Signal

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel. Demodulate and decode the received signal. Compute the error statistics for the reception of uncoded and LDPC-coded signals.

Define simulation variables. Create System objects for the LDPC encoder, LDPC decoder, QPSK modulator, and QPSK demodulators.

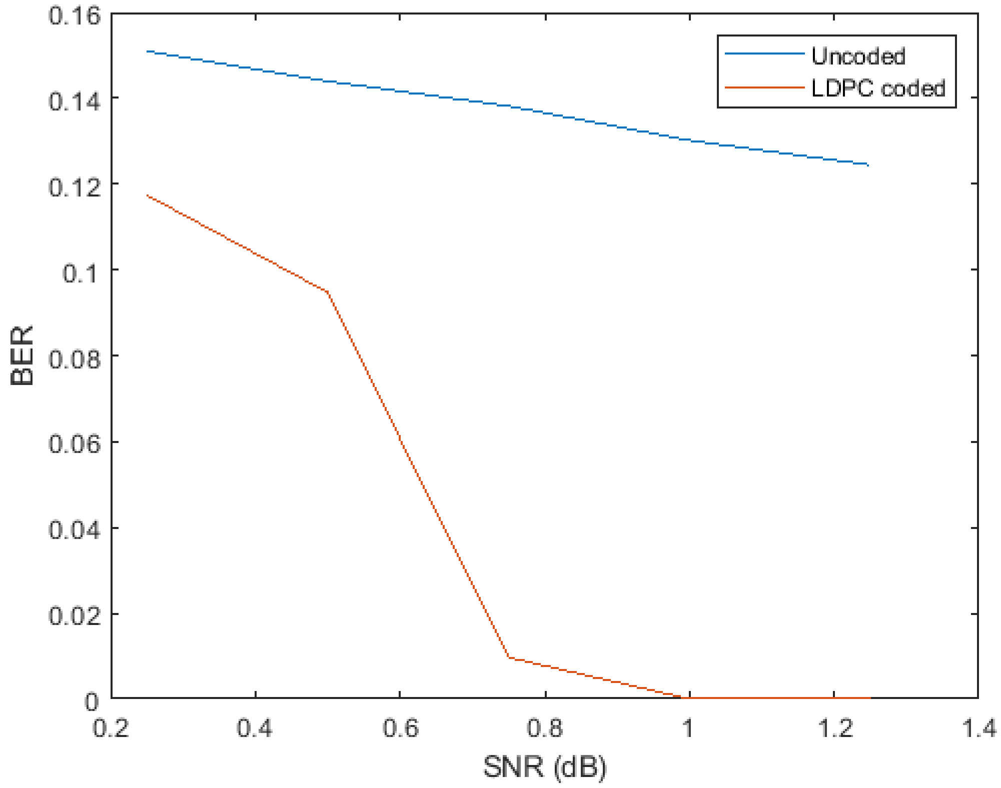
```
M = 4; % Modulation order (QPSK)
snr = [0.25,0.5,0.75,1.0,1.25];
numFrames = 10;
ldpcEncoder = comm.LDPCEncoder;
ldpcDecoder = comm.LDPCDecoder;
pskMod = comm.PSKModulator(M,'BitInput',true);
pskDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Approximate log-likelihood ratio');
pskuDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Hard decision');
errRate = zeros(1,length(snr));
uncErrRate = zeros(1,length(snr));
```

For each SNR setting and all frames, compute the error statistics for uncoded and LDPC-coded signals.

```
for ii = 1:length(snr)
    ttlErr = 0;
    ttlErrUnc = 0;
    pskDemod.Variance = 1/10^(snr(ii)/10); % Set variance using current SNR
    for counter = 1:numFrames
        data = logical(randi([0 1],32400,1));
        % Transmit and receiver uncoded signal data
        mod_uncSig = pskMod(data);
        rx_uncSig = awgn(mod_uncSig,snr(ii),'measured');
        demod_uncSig = pskuDemod(rx_uncSig);
        numErrUnc = biterr(data,demod_uncSig);
        ttlErrUnc = ttlErrUnc + numErrUnc;
        % Transmit and receive LDPC coded signal data
        encData = ldpcEncoder(data);
        modSig = pskMod(encData);
        rxSig = awgn(modSig,snr(ii),'measured');
        demodSig = pskDemod(rxSig);
        rxBits = ldpcDecoder(demodSig);
        numErr = biterr(data,rxBits);
        ttlErr = ttlErr + numErr;
    end
    ttlBits = numFrames*length(rxBits);
    uncErrRate(ii) = ttlErrUnc/ttlBits;
    errRate(ii) = ttlErr/ttlBits;
end
```

Plot the error statistics for uncoded and LDPC-coded data.

```
plot(snr,uncErrRate,snr,errRate)
legend('Uncoded', 'LDPC coded')
xlabel('SNR (dB)')
ylabel('BER')
```

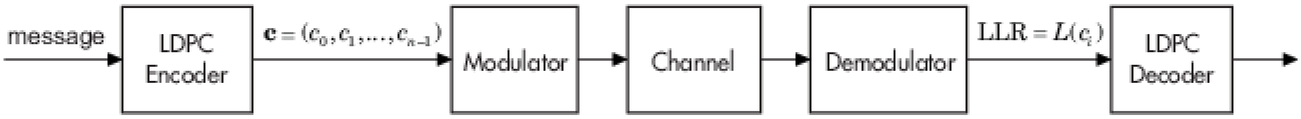


Algorithms

This object performs LDPC decoding using the belief propagation algorithm, also known as a message-passing algorithm.

Belief Propagation Decoding

The implementation of the belief propagation algorithm is based on the decoding algorithm presented by Gallager.



For transmitted LDPC-encoded codeword $c = c_0, c_1, \ldots, c_{n\text{-}1}$, the input to the LDPC decoder is the log-likelihood ratio (LLR) value $L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid \text{channel output for } c_i)}{\Pr(c_i = 1 \mid \text{channel output for } c_i)}\right)$.

In each iteration, the key components of the algorithm are updated based on these equations:

$$L(r_{ji}) = 2\,\text{atanh}\left(\prod_{i' \in V_j \backslash i} \tanh\left(\frac{1}{2} L(q_{i'j})\right)\right),$$

$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \backslash j} L(r_{j'i})$, initialized as $L(q_{ij}) = L(c_i)$ before the first iteration, and

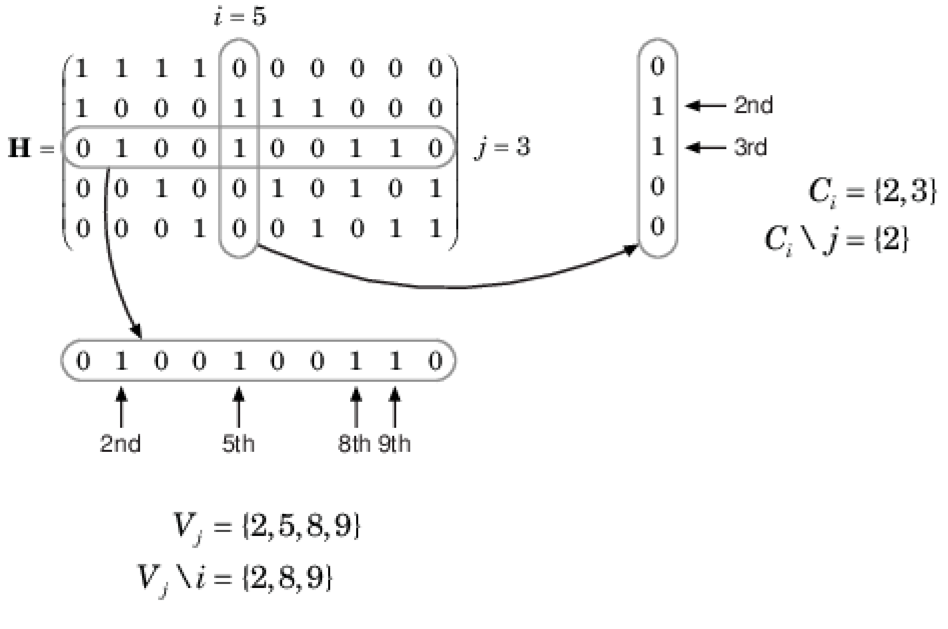
$$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i}).$$

At the end of each iteration, $L(Q_i)$ contains the updated estimate of the LLR value for transmitted bit c_i. The value $L(Q_i)$ is the soft-decision output for c_i. If $L(Q_i) < 0$, the hard-decision output for c_i is 1. Otherwise, the hard-decision output for c_i is 0.

If configured to stop when all parity checks are satisfied, the algorithm verifies the parity-check equation ($H\, c' = 0$) at the end of each iteration. When all parity checks are satisfied, or if the maximum number of iterations is reached, decoding stops.

Index sets $C_i \backslash j$ and $V_j \backslash i$ are based on the parity-check matrix (PCM). Index sets C_i and V_j correspond to all nonzero elements in column i and row j of the PCM, respectively.

This figure highlights the computation of these index sets in a given PCM for $i = 5$ and $j = 3$.



To avoid infinite numbers in the algorithm equations, atanh(1) and atanh(-1) are set to 19.07 and –19.07, respectively. Due to finite precision, MATLAB returns 1 for tanh(19.07) and –1 for tanh(-19.07).

References

[1] Gallager, Robert G. *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Using default settings, `comm.LDPCDecoder` does not support code generation. To generate code, specify the `ParityCheckMatrix` property as a nonsparse index matrix.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.BCHDecoder` | `comm.LDPCEncoder` | `comm.gpu.LDPCDecoder`

Functions
`dvbs2ldpc`

Blocks
LDPC Decoder

Introduced in R2012a

comm.LDPCEncoder

Package: comm

Encode binary low-density parity-check (LDPC) code

Description

The comm.LDPCEncoder System object applies LDPC coding to a binary input message. LDPC codes are linear error control codes with sparse parity-check matrices and long block lengths that can attain performance near the Shannon limit.

To encode a binary LDPC code:

1. Create the comm.LDPCEncoder object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
ldpcencoder = comm.LDPCEncoder
ldpcencoder = comm.LDPCEncoder(parity)
ldpcencoder = comm.LDPCEncoder(___,Name,Value)
```

Description

ldpcencoder = comm.LDPCEncoder creates a binary LDPC encoder System object. This object performs LDPC encoding based on the default parity-check matrix.

ldpcencoder = comm.LDPCEncoder(parity) sets the ParityCheckMatrix property to parity and creates an LDPC encoder System object. The parity input must be specified as described by the ParityCheckMatrix property.

ldpcencoder = comm.LDPCEncoder(___,Name,Value) sets properties using one or more name-value pairs, in addition to inputs from any of the prior syntaxes. For example, comm.LDPCEncoder('ParityCheckMatrix',sparse(I(:,1),I(:,2),1)) configures an LDPC encoder System object to encode data using the parity matrix sparse(I(:,1),I(:,2),1). Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

ParityCheckMatrix — Parity-check matrix
dvbs2ldpc(1/2) (default) | sparse binary matrix | nonsparse index matrix

Parity-check matrix, specified as a sparse (*N* – *K*)-by-*N* binary-valued matrix. *N* is the length of the output codeword vector, and must be in the range (0, 2^{31}). *K* is the length of the uncoded message and must be less than *N*. The last (*N* – *K*) columns in the parity-check matrix must be an invertible matrix in the Galois field of order 2, gf(2).

You can also specify the parity-check matrix as a two-column nonsparse index matrix, I, that defines the row and column indices of the 1s in the parity-check matrix such that sparse(I(:,1),I(:,2),1).

This property accepts numeric data types. When you set this property to a sparse binary matrix, this property also accepts the logical data type.

The default value uses the dvbs2ldpc function to configure a sparse parity-check matrix for half-rate LDPC coding, as specified in the DVB-S.2 standard.

Note

- When the last (*N* – *K*) columns of the parity-check matrix form a triangular matrix, forward or backward substitution is performed to solve the parity-check equation.
- When the last (*N* – *K*) columns of the parity-check matrix do not form a triangular matrix, a matrix inversion is performed to solve the parity-check equation. If a large matrix needs to be inverted, initializations or updates take more time.

Example: dvbs2ldpc(R,'indices') configures the index matrix for the DVB-S.2 standard, where R is the code rate, and 'indices' specifies the output format of dvbs2ldpc as a two-column double-precision matrix that defines the row and column indices of the 1s in the parity-check matrix.

Data Types: double | logical

Usage

Syntax

```
codeword = ldpcencoder(message)
```

Description

codeword = ldpcencoder(message) codes the input message using an LDPC code based on a parity-check matrix. The LDPC codeword output is a solution to the parity-check equation.

Input Arguments

message — Input message
binary column vector

Input message, specified as a *K*-by-1 column vector containing binary-valued elements. *K* is the length of the uncoded message.

Data Types: `double` | `logical`

Output Arguments

`codeword` — LDPC codeword
column vector

LDPC codeword, returned as an *N*-by-1 column vector. *N* is the number of bits in the LDPC codeword. The output signal inherits its data type from the input signal. The LDPC codeword output is a solution to the parity-check equation. The input message comprises the first *K* bits of the LDPC codeword output, and the parity check comprises the remaining (*N* – *K*) bits.

Data Types: `double` | `logical`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

LDPC Encode and Decode QPSK-Modulated Signal

Transmit an LDPC-encoded, QPSK-modulated bit stream through an AWGN channel. Demodulate and decode the received signal. Compute the error statistics for the reception of uncoded and LDPC-coded signals.

Define simulation variables. Create System objects for the LDPC encoder, LDPC decoder, QPSK modulator, and QPSK demodulators.

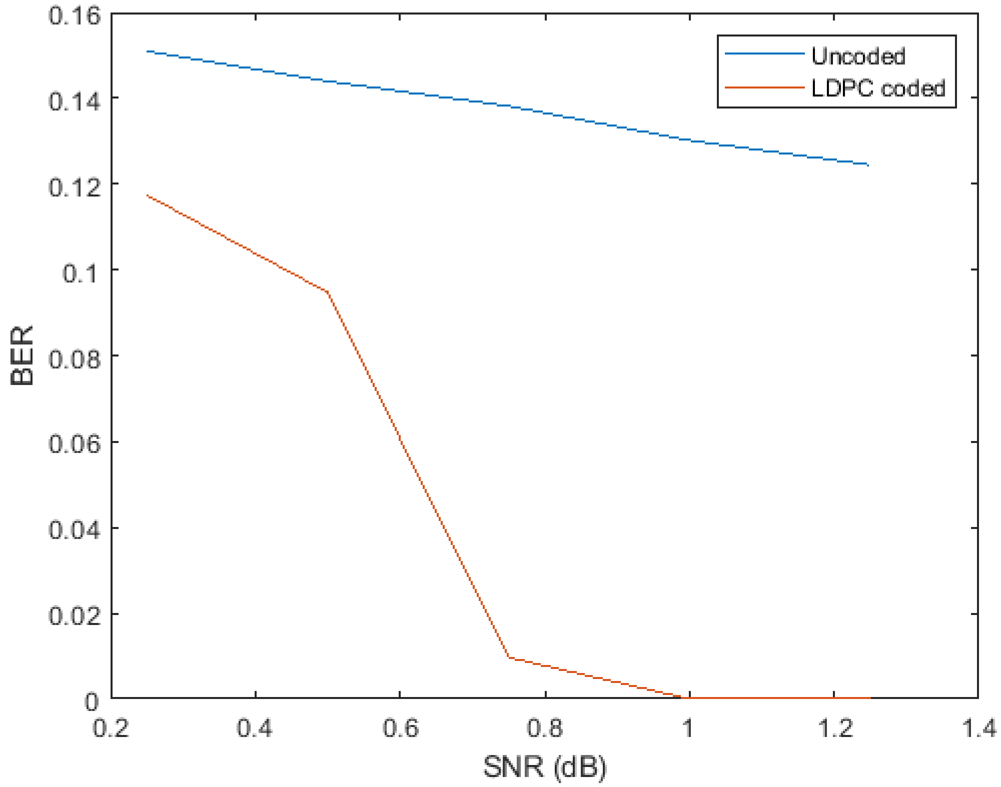
```
M = 4; % Modulation order (QPSK)
snr = [0.25,0.5,0.75,1.0,1.25];
numFrames = 10;
ldpcEncoder = comm.LDPCEncoder;
ldpcDecoder = comm.LDPCDecoder;
pskMod = comm.PSKModulator(M,'BitInput',true);
pskDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Approximate log-likelihood ratio');
pskuDemod = comm.PSKDemodulator(M,'BitOutput',true,...
    'DecisionMethod','Hard decision');
errRate = zeros(1,length(snr));
uncErrRate = zeros(1,length(snr));
```

For each SNR setting and all frames, compute the error statistics for uncoded and LDPC-coded signals.

```
for ii = 1:length(snr)
    ttlErr = 0;
    ttlErrUnc = 0;
    pskDemod.Variance = 1/10^(snr(ii)/10); % Set variance using current SNR
    for counter = 1:numFrames
        data = logical(randi([0 1],32400,1));
        % Transmit and receiver uncoded signal data
        mod_uncSig = pskMod(data);
        rx_uncSig = awgn(mod_uncSig,snr(ii),'measured');
        demod_uncSig = pskuDemod(rx_uncSig);
        numErrUnc = biterr(data,demod_uncSig);
        ttlErrUnc = ttlErrUnc + numErrUnc;
        % Transmit and receive LDPC coded signal data
        encData = ldpcEncoder(data);
        modSig = pskMod(encData);
        rxSig = awgn(modSig,snr(ii),'measured');
        demodSig = pskDemod(rxSig);
        rxBits = ldpcDecoder(demodSig);
        numErr = biterr(data,rxBits);
        ttlErr = ttlErr + numErr;
    end
    ttlBits = numFrames*length(rxBits);
    uncErrRate(ii) = ttlErrUnc/ttlBits;
    errRate(ii) = ttlErr/ttlBits;
end
```

Plot the error statistics for uncoded and LDPC-coded data.

```
plot(snr,uncErrRate,snr,errRate)
legend('Uncoded', 'LDPC coded')
xlabel('SNR (dB)')
ylabel('BER')
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.BCHEncoder` | `comm.LDPCDecoder`

Functions

`dvbs2ldpc`

Blocks

LDPC Decoder

Introduced in R2012a

comm.LTEMIMOChannel

Package: `comm`

(To be removed) Filter input signal through LTE MIMO multipath fading channel

Note `comm.LTEMIMOChannel` will be removed in a future release. Use `comm.MIMOChannel` instead.

Description

The `comm.LTEMIMOChannel` System object filters an input signal through an LTE multiple-input multiple-output (MIMO) multipath fading channel.

A specialization of the `comm.MIMOChannel` System object, the `comm.LTEMIMOChannel` System objects offers pre-set configurations for use with LTE link level simulations. In addition to the `comm.MIMOChannel` System object, the `comm.LTEMIMOChannel` System object also corrects the correlation matrix to be positive semi-definite, after rounding to 4-digit precision. This System object models Rayleigh fading for each of its links.

To filter an input signal using an LTE MIMO multipath fading channel:

1. Define and set up your LTE MIMO multipath fading channel object. See “Construction” on page 3-901.
2. Call `step` to filter the input signal using an LTE MIMO multipath fading channel according to the properties of `comm.LTEMIMOChannel`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.LTEMIMOChannel` creates a 3GPP Long Term Evolution (LTE) Release 10 specified multiple-input multiple-output (MIMO) multipath fading channel System object, `H`. This object filters a real or complex input signal through the multipath LTE MIMO channel to obtain the channel impaired signal.

`H = comm.LTEMIMOChannel(Name,Value)` creates an LTE MIMO multipath fading channel object, `H`, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`SampleRate`

Input signal sample rate (Hertz)

Specify the sample rate of the input signal in hertz as a double-precision, real, positive scalar. The default value of this property is `30.72` MHz, as defined in the LTE specification.

`Profile`

Channel propagation profile

Specify the propagation conditions of the LTE multipath fading channel as one of `EPA 5Hz` | `EVA 5Hz` | `EVA 70Hz` | `ETU 70Hz` | `ETU 300Hz`, which are supported in the LTE specification Release 10. The default value of this property is `EPA 5Hz`.

This property defines the delay profile of the channel to be one of EPA, EVA, and ETU. This property also defines the maximum Doppler shift of the channel to be 5 Hz, 70 Hz, or 300 Hz. The Doppler spectrum always has a Jakes shape in the LTE specification. The EPA profile has seven paths. The EVA and ETU profiles have nine paths.

The following tables list the delay and relative power per path associated with each profile.

Extended Pedestrian A Model (EPA)

Excess tap delay [ns]	Relative power [db]
0	0.0
30	-1.0
70	-2.0
90	-3.0
110	-8.0
190	-17.2
410	-20.8

Extended Vehicular A Model (EVA)

Excess tap delay [ns]	Relative power [db]
0	0.0
30	-1.5
150	-1.4
310	-3.6
370	-0.6
710	-9.1
1090	-7.0
1730	-12.0
2510	-16.9

Extended Typical Urban Model (ETU)

Excess tap delay [ns]	Relative power [db]
0	-1.0

Excess tap delay [ns]	Relative power [db]
50	-1.0
120	-1.0
200	0.0
230	0.0
500	0.0
1600	-3.0
2300	-5.0
5000	-7.0

`AntennaConfiguration`

Antenna configuration

Specify the antenna configuration of the LTE MIMO channel as one of `1x2` | `2x2` | `4x2` | `4x4`. These configurations are supported in the LTE specification Release 10. The default value of this property is `2x2`.

The property value is in the format of N_t-by-N_r. N_t represents the number of transmit antennas and N_r represents the number of receive antennas.

`CorrelationLevel`

Spatial correlation strength

Specify the spatial correlation strength of the LTE MIMO channel as one of `Low` | `Medium` | `High`. The default value of this property is `Low`. When you set this property to `Low`, the MIMO channel is spatially uncorrelated.

The transmit and receive spatial correlation matrices are defined from this property according to the LTE specification Release 10. See the Algorithms section for more information.

`AntennaSelection`

Antenna selection

Specify the antenna selection scheme as one of `Off` | `Tx` | `Rx` | `Tx and Rx`, where `Tx` represents transmit antennas and `Rx` represents receive antennas. When you select `Tx` and/or `Rx`, additional input(s) are required to specify which antennas are selected for signal transmission. The default value of this property is `Off`.

`RandomStream`

Source of random number stream

Specify the source of random number stream as one of `Global stream` | `mt19937ar with seed`. The default value of this property is `Global stream`. When you set this property to `Global stream`, the current global random number stream is used for normally distributed random number generation. In this case, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the object uses the mt19937ar algorithm for normally distributed random number generation. In this case, the `reset` method resets the filters and reinitializes the random number stream to the value of the Seed property.

Seed

Initial seed of mt19937ar random number stream

Specify the initial seed of an mt19937ar random number generator algorithm as a double-precision, real, nonnegative integer scalar. The default value of this property is `73`. This property applies when you set the RandomStream property to `mt19937ar with seed`. The `Seed` reinitializes the mt19937ar random number stream in the `reset` method.

NormalizePathGains

Normalize path gains (logical)

Set this property to `true` to normalize the fading processes so that the total power of the path gains, averaged over time, is `0` dB. The default value of this property is `true`. When you set this property to `false`, there is no normalization for path gains.

NormalizeChannelOutputs

Normalize channel outputs (logical)

Set this property to `true` to normalize the channel outputs by the number of receive antennas. The default value of this property is `true`. When you set this property to `false`, there is no normalization for channel outputs.

PathGainsOutputPort

Enable path gain output (logical)

Set this property to `true` to output the channel path gains of the underlying fading process. The default value of this property is `false`.

Methods

reset	(To be removed) Reset states of the `LTEMIMOChannel` object
step	(To be removed) Filter input signal through LTE MIMO multipath fading channel

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Configure MIMO Channel Object Using LTE MIMO Channel Object

Configure an equivalent `MIMOChannel` System Object using the `LTEMIMOChannel` System Object. Then, verify that the channel output and the path gain output from the two objects are the same.

Create a PSK Modulator System object™ to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
modData = pskModulator(randi([0 pskModulator.ModulationOrder-1],2e3,1));
```

Split modulated data into two spatial streams.

```
channelInput = reshape(modData,[2 1e3]).';
```

Create an `LTEMIMOChannel` System object with a 2-by-2 antenna configuration and a medium correlation level.

```
lteChan = comm.LTEMIMOChannel(...
    'Profile',              'EVA 5Hz',...
    'AntennaConfiguration', '2x2',...
    'CorrelationLevel',     'Medium',...
    'AntennaSelection',     'Off',...
    'RandomStream',         'mt19937ar with seed',...
    'Seed',                 99,...
    'PathGainsOutputPort',  true);
```

```
Warning: COMM.LTEMIMOCHANNEL will be removed in a future release. Use COMM.MIMOCHANNEL or LTEFADI
```

Filter the modulated data using the `LTEMIMOChannel` System object, `lteChan`.

```
[LTEChanOut,LTEPathGains] = lteChan(channelInput);
```

Create an equivalent `MIMOChannel` System object, `mimoChannel`, using the properties of the `LTEMIMOChannel` System object, `lteChan`.

The `KFactor`, `DirectPathDopplerShift` and `DirectPathInitialPhase` properties only exist for the `MIMOChannel` System object. All other `MIMOChannel` System object properties also exist for the `LTEMIMOChannel` System object; however, some properties are hidden and read-only.

```
mimoChannel = comm.MIMOChannel( ...
    'SampleRate',lteChan.SampleRate, ...
    'PathDelays',lteChan.PathDelays, ...
    'AveragePathGains',lteChan.AveragePathGains, ...
    'NormalizePathGains',lteChan.NormalizePathGains, ...
    'FadingDistribution',lteChan.FadingDistribution, ...
    'MaximumDopplerShift',lteChan.MaximumDopplerShift, ...
    'DopplerSpectrum',lteChan.DopplerSpectrum, ...
    'SpatialCorrelationSpecification', ...
         lteChan.SpatialCorrelationSpecification, ...
    'SpatialCorrelationMatrix',lteChan.SpatialCorrelationMatrix, ...
    'AntennaSelection',lteChan.AntennaSelection, ...
    'NormalizeChannelOutputs',lteChan.NormalizeChannelOutputs, ...
    'RandomStream',lteChan.RandomStream, ...
    'Seed',lteChan.Seed, ...
    'PathGainsOutputPort',lteChan.PathGainsOutputPort);
```

Filter the modulated data using the equivalent `mimoChannel` object.

```
[MIMOChanOut, MIMOPathGains] = mimoChannel(channelInput);
```

Verify that the channel output and the path gain output from the two objects are the same.

```
sameChOutput = isequal(LTEChanOut,MIMOChanOut)
```

```
sameChOutput = logical
   1
```

```
samePathGains = isequal(LTEPathGains,MIMOPathGains)
samePathGains = logical
   1
```

You can repeat the preceding process with `AntennaConfiguration` set to `4x2` or `4x4` and `CorrelationLevel` set to `Medium` or `High` for `lteChan`.

Algorithms

This System object is a specialized implementation of the `comm.MIMOChannel` System object. For additional algorithm information, see the `comm.MIMOChannel` System object help page.

Spatial Correlation Matrices

The following table defines the transmitter eNodeB correlation matrix.

	One Antenna	Two Antennas	Four Antennas
eNodeB Correlation	$R_{\mathrm{eNB}} = 1$	$R_{eNB} = \begin{pmatrix} 1 & \alpha \\ \alpha^* & 1 \end{pmatrix}$	$R_{eNB} =$ $\begin{pmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9*} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9*} & \alpha^{1/9*} & 1 & \alpha^{1/9} \\ \alpha^* & \alpha^{4/9*} & \alpha^{1/9*} & 1 \end{pmatrix}$

The following table defines the receiver UE correlation matrix.

	One Antenna	Two Antennas	Four Antennas
UE Correlation	$R_{\mathrm{UE}} = 1$	$R_{UE} = \begin{pmatrix} 1 & \beta \\ \beta^* & 1 \end{pmatrix}$	$R_{UE} =$ $\begin{pmatrix} 1 & \beta^{1/9} & \beta^{4/9} & \beta \\ \beta^{1/9}* & 1 & \beta^{1/9} & \beta^{4/9} \\ \beta^{4/9}* & \beta^{1/9}* & 1 & \beta^{1/9} \\ \beta* & \beta^{4/9}* & \beta^{1/9}* & 1 \end{pmatrix}$

The following table describes the R_{spat} channel spatial correlation matrix between the transmitter and receiver antennas.

Tx-by-Rx Configuration	Correlation Matrix
1-by-2	$R_{spat} = R_{UE} = \begin{bmatrix} 1 & \beta \\ \beta* & 1 \end{bmatrix}$
2-by-2	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha \\ \alpha* & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & \beta \\ \beta* & 1 \end{bmatrix}$ $= \begin{bmatrix} 1 & \beta & \alpha & \alpha\beta \\ \beta* & 1 & \alpha\beta* & \alpha \\ \alpha* & \alpha*\beta & 1 & \beta \\ \alpha*\beta* & \alpha* & \beta* & 1 \end{bmatrix}$
4-by-2	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9*} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9*} & \alpha^{1/9*} & 1 & \alpha^{1/9} \\ \alpha* & \alpha^{4/9*} & \alpha^{1/9*} & 1 \end{bmatrix}$ $\otimes \begin{bmatrix} 1 & \beta \\ \beta* & 1 \end{bmatrix}$

Tx-by-Rx Configuration	Correlation Matrix
4-by-4	$R_{spat} = R_{eNB} \otimes R_{UE} = \begin{bmatrix} 1 & \alpha^{1/9} & \alpha^{4/9} & \alpha \\ \alpha^{1/9*} & 1 & \alpha^{1/9} & \alpha^{4/9} \\ \alpha^{4/9*} & \alpha^{1/9*} & 1 & \alpha^{1/9} \\ \alpha^{*} & \alpha^{4/9*} & \alpha^{1/9*} & 1 \end{bmatrix} \otimes \begin{pmatrix} 1 & \beta^{1/9} & \beta^{4/9} & \beta \\ \beta^{1/9*} & 1 & \beta^{1/9} & \beta^{4/9} \\ \beta^{4/9*} & \beta^{1/9*} & 1 & \beta^{1/9} \\ \beta^{*} & \beta^{4/9*} & \beta^{1/9*} & 1 \end{pmatrix}$

Spatial Correlation Correction

Low Correlation		Medium Correlation		High Correlation	
α	β	α	β	α	β
0	0	0.3	0.9	0.9	0.9

To insure the correlation matrix is positive semi-definite after round-off to 4 digit precision, this System object uses the following equation:

$$R_{high} = \left[R_{spatial} + aI_n\right]/(1 + a)$$

Where

α represents the scaling factor such that the smallest value is used to obtain a positive semi-definite result.

For the 4-by-2 high correlation case, α=0.00010.

For the 4-by-4 high correlation case, α=0.00012.

The object uses the same method to adjust the 4-by-4 medium correlation matrix to insure the correlation matrix is positive semi-definite after rounding to 4 digit precision with α = 0.00012.

Selected Bibliography

[1] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), *Base Station (BS) radio transmission and reception*, Release 10, 2009–2010, 3GPP TS 36.104, Vol. 10.0.0.

[2] 3rd Generation Partnership Project, Technical Specification Group Radio Access Network, Evolved Universal Terrestrial Radio Access (E-UTRA), *User Equipment (UE) radio transmission and reception*, Release 10, 2010, 3GPP TS 36.101, Vol. 10.0.0.

[3] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.

[4] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

[5] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*, Second Edition, New York, Kluwer Academic/Plenum, 2000.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.MIMOChannel`

Introduced in R2012a

reset

System object: `comm.LTEMIMOChannel`
Package: `comm`

(To be removed) Reset states of the `LTEMIMOChannel` object

Note `comm.LTEMIMOChannel` will be removed in a future release. Use `comm.MIMOChannel` instead.

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `LTEMIMOChannel` object, `H`.

If you set the RandomStream property of `H` to `Global stream`, the `reset` method only resets the filters. If you set `RandomStream` to `mt19937ar with seed`, the `reset` method not only resets the filters but also reinitializes the random number stream to the value of the Seed property.

step

System object: `comm.LTEMIMOChannel`
Package: `comm`

(To be removed) Filter input signal through LTE MIMO multipath fading channel

Note `comm.LTEMIMOChannel` will be removed in a future release. Use `comm.MIMOChannel` instead.

Syntax

```
Y = step(H,X)
[Y,PATHGAINS] = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` filters input signal `X` through an LTE MIMO multipath fading channel and returns the result in `Y`. The input `X` can be a double- or single-precision data type scalar, vector, or 2-D matrix with real or complex values. `X` is of size N_s-by-N_t. N_s represents the number of samples and N_t represents the number of transmit antennas that must match the AntennaConfiguration property setting of `H`. `Y` is the output signal of size N_s-by-N_r. N_r represents the number of receive antennas that is specified by the `AntennaConfiguration` property of `H`. `Y` contains complex values with same precision as input signal.

`[Y,PATHGAINS] = step(H,X)` returns the LTE MIMO channel path gains of the underlying fading process in PATHGAINS. This applies when you set the PathGainsOutputPort property to `true`. PATHGAINS is of size N_s-by-N_p-by-N_t-by-N_r. N_p represents the number of discrete paths of the channel implicitly defined by the Profile property of `H`. PATHGAINS contains complex values with same precision as input signal.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MatrixDeinterleaver

Package: `comm`

(To be removed) Deinterleave input symbols using permutation matrix

Compatibility

comm.MatrixDeinterleaver will be removed in a future release. Use `matdeintrlv` instead. For more information, see “Compatibility Considerations” on page 3-913.

Description

The `MatrixDeinterleaver` object performs block deinterleaving by filling a matrix with the input symbols column by column and then sending the matrix contents to the output port row by row. The number of rows and number of columns properties set the dimensions of the matrix that the object uses internally for computations.

To deinterleave input symbols using a permutation vector:

1. Define and set up your matrix deinterleaver object. See “Construction” on page 3-912.
2. Call `step` to deinterleave the input signal according to the properties of `comm.MatrixDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MatrixDeinterleaver` creates a matrix deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the matrix interleaver object.

`H = comm.MatrixDeinterleaver(Name,Value)` creates a matrix deinterleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.MatrixDeinterleaver(N,M)` creates a matrix deinterleaver object, `H`. This object has the `NumRows` property set to `N`, the `NumColumns` property set to `M`.

Properties

NumRows

Number of rows of permutation matrix

Specify the number of permutation matrix rows as a scalar, positive integer. The default is `3`.

`NumColumns`

Number of columns of permutation matrix

Specify the number of permutation matrix columns as a scalar, positive integer. The default is 4.

Methods

step	(To be removed) Deinterleave input symbols using permutation matrix

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Matrix Interleaving and Deinterleaving

Create matrix interleaver and deinterleaver objects.

```
interleaver = comm.MatrixInterleaver('NumRows',2,'NumColumns', 5);
```

```
Warning: COMM.MATRIXINTERLEAVER will be removed in a future release. Use MATINTRLV instead. See
```

```
deinterleaver = comm.MatrixDeinterleaver('NumRows',2,'NumColumns', 5);
```

```
Warning: COMM.MATRIXDEINTERLEAVER will be removed in a future release. Use MATDEINTRLV instead.
```

Generate random data, interleave, and then deinterleave the data.

```
data = randi(7,10,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.MatrixDeinterleaver` will be removed
Not recommended starting in R2019b

comm.MatrixDeinterleaver will be removed in a future release. Use `matdeintrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`deintrlv` | `intrlv` | `matdeintrlv` | `matintrlv`

Introduced in R2012a

step

System object: `comm.MatrixDeinterleaver`
Package: `comm`

(To be removed) Deinterleave input symbols using permutation matrix

Compatibility

step will be removed in a future release. Use `matdeintrlv` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a block interleaver. The object fills a permutation matrix with the input symbols column by column and outputs the matrix contents row by row in the output, `Y`. The input `X` must be a column vector of length equal to NumRows × NumColumns. The data type for `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MatrixInterleaver

Package: comm

(To be removed) Permute input symbols using permutation matrix

Compatibility

comm.MatrixInterleaver will be removed in a future release. Use matintrlv instead. For more information, see “Compatibility Considerations” on page 3-917.

Description

The MatrixInterleaver object performs block interleaving by filling a matrix with the input symbols row by row and then outputs the matrix contents column-by-column.

To perform block interleaving using a permutation matrix:

1. Define and set up your matrix interleaver object. See “Construction” on page 3-916.
2. Call step to interleave the input symbols according to the properties of comm.MatrixInterleaver. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.MatrixInterleaver creates a matrix interleaver System object, H. This object permutes the input by filling a permutation matrix with the input symbols row by row. The object then outputs the matrix contents column by column.

H = comm.MatrixInterleaver(Name,Value) creates a matrix interleaver object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.MatrixInterleaver(N,M) creates a matrix interleaver object, H. This object has the NumRows property set to N, the NumColumns property set to M.

Properties

NumRows

Number of rows of permutation matrix

Specify the number of permutation matrix rows as a scalar, positive integer. The default is 3.

NumColumns

Number of columns of permutation matrix

Specify the number of permutation matrix columns as a scalar, positive integer. The default is 4.

Methods

step (To be removed) Permute input symbols using permutation matrix

Common to All System Objects	
release	Allow System object property value changes

Examples

Matrix Interleaving and Deinterleaving

Create matrix interleaver and deinterleaver objects.

```
interleaver = comm.MatrixInterleaver('NumRows',2,'NumColumns', 5);
```

```
Warning: COMM.MATRIXINTERLEAVER will be removed in a future release. Use MATINTRLV instead. See <
```

```
deinterleaver = comm.MatrixDeinterleaver('NumRows',2,'NumColumns', 5);
```

```
Warning: COMM.MATRIXDEINTERLEAVER will be removed in a future release. Use MATDEINTRLV instead. S
```

Generate random data, interleave, and then deinterleave the data.

```
data = randi(7,10,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Confirm the original and deinterleaved data are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

comm.MatrixInterleaver will be removed
Not recommended starting in R2019b

comm.MatrixInterleaver will be removed in a future release. Use matintrlv instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`deintrlv` | `intrlv` | `matdeintrlv` | `matintrlv`

Introduced in R2012a

step

System object: `comm.MatrixInterleaver`
Package: `comm`

(To be removed) Permute input symbols using permutation matrix

Compatibility

step will be removed in a future release. Use `matintrlv` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The object fills a permutation matrix with the input symbols row by row and outputs the matrix contents column by column. The input `X` must be a column vector of length NumRows × NumColumns and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MatrixHelicalScanDeinterleaver

Package: `comm`

(To be removed) Deinterleave input symbols by filling a matrix along diagonals

Compatibility

comm.MatrixHelicalScanDeinterleaver will be removed in a future release. Use `helscandeintrlv` instead. For more information, see "Compatibility Considerations" on page 3-922.

Description

The `MatrixHelicalScanDeinterleaver` object performs block deinterleaving by filling a matrix with the input symbols helically and then outputs the matrix contents row by row. The number of rows and number of columns properties represent the dimensions of the matrix that the object uses internally for computations.

To deinterleave the input symbols by filling a matrix with the input symbols helically and then outputting the matrix contents row-by-row:

1. Define and set up your matrix helical scan deinterleaver object. See "Construction" on page 3-920.
2. Call `step` to deinterleave the input signal according to the properties of `comm.MatrixHelicalScanDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MatrixHelicalScanDeinterleaver` creates a matrix helical scan deinterleaver object, `H`. This object restores the original ordering of a sequence that was interleaved using the matrix helical scan interleaver System object.

`H = comm.MatrixHelicalScanDeinterleaver(Name,Value)` creates a matrix helical scan deinterleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`NumRows`

Number of rows of permutation matrix

Specify the number of rows in the permutation matrix as a scalar, positive integer. The default is `64`.

NumColumns

Number of columns of permutation matrix

Specify the number of columns in the permutation matrix as a scalar, positive integer. The default is 64.

StepSize

Slope of diagonals

Specify slope as a scalar integer between 0 and the value you specify in the NumRows on page 3-0 property. The default is 1. The slope value indicates the amount by which the row index increases as the column index increases by 1. When you set the value of this property to 0, the object does not interleave and the output matches the input.

Methods

step (To be removed) Deinterleave input symbols by filling a matrix along diagonals

Common to All System Objects	
release	Allow System object property value changes

Examples

Matrix Helical Scan Interleaving and Deinterleaving

Create matrix helical scan interleaver and deinterleaver objects.

```
interleaver = comm.MatrixHelicalScanInterleaver('NumRows',4,'NumColumns', 4);
```

```
Warning: COMM.MATRIXHELICALSCANINTERLEAVER will be removed in a future release. Use HELSCANINTRLV
```

```
deinterleaver = comm.MatrixHelicalScanDeinterleaver('NumRows',4,'NumColumns',4);
```

```
Warning: COMM.MATRIXHELICALSCANDEINTERLEAVER will be removed in a future release. Use HELSCANDEIN
```

Generate random symbols. Pass the data through the interleaver, and then pass that data through the deinterleaver.

```
data = randi(7,16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intData deIntData]
```

```
ans = 16×3

     6     6     6
     7     1     7
     1     2     1
     7     1     7
     5     5     5
     1     7     1
     2     6     2
     4     7     4
     7     7     7
     7     4     7
      ⋮
```

Confirm that the original and deinterleaved sequences are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Helical Scan Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.MatrixHelicalScanDeinterleaver` will be removed

Not recommended starting in R2019b

comm.MatrixHelicalScanDeinterleaver will be removed in a future release. Use `helscandeintrlv` instead.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions

`deintrlv` | `helscandeintrlv` | `helscanintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.MatrixHelicalScanDeinterleaver`
Package: `comm`

(To be removed) Deinterleave input symbols by filling a matrix along diagonals

Compatibility

step will be removed in a future release. Use `helscandeintrlv` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`. The object fills a permutation matrix with the input symbols in a helical fashion and output the contents row by row, and returns `Y`. The input `X` must be a NumRows × NumColumns long column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MatrixHelicalScanInterleaver

Package: `comm`

(To be removed) Permute input symbols by selecting matrix elements along diagonals

Compatibility

comm.MatrixHelicalScanInterleaver will be removed in a future release. Use `helscanintrlv` instead. For more information, see “Compatibility Considerations” on page 3-926.

Description

The `MatrixHelicalScanInterleaver` object performs block interleaving by filling a matrix with the input symbols row by row and then outputs the matrix contents helically. The number of rows and number of columns properties are the dimensions of the matrix that the object uses internally for computations.

To interleave the input signal by filling a matrix row-by-row with the input symbols and then outputting the matrix contents helically:

1. Define and set up your matrix helical scan interleaver object. See “Construction” on page 3-924.
2. Call `step` to interleave the input signal according to the properties of `comm.MatrixHelicalScanInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MatrixHelicalScanInterleaver` creates a matrix helical scan interleaver object, `H`. This object permutes the input by filling a permutation matrix with the input symbols row by row and then outputs the matrix contents helically.

`H = comm.MatrixHelicalScanInterleaver(Name,Value)` creates a matrix helical scan interleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

NumRows

Number of rows of permutation matrix

Specify the number of rows in the permutation matrix as a scalar, positive integer. The default is `64`.

NumColumns

Number of columns of permutation matrix

Specify the number of columns in the permutation matrix as a scalar, positive integer. The default is 64.

StepSize

Slope of diagonals

Specify slope as a scalar integer between 0 and the value you specify in the NumRows on page 3-0 property. The slope value represents the amount by which the row index increases as the column index increases by 1. When you set the value of this property to 0, the object does not interleave and the output matches the input. The default is 1.

Methods

step	(To be removed) Permute input symbols by selecting matrix elements along diagonals

Common to All System Objects	
release	Allow System object property value changes

Examples

Matrix Helical Scan Interleaving and Deinterleaving

Create matrix helical scan interleaver and deinterleaver objects.

```
interleaver = comm.MatrixHelicalScanInterleaver('NumRows',4,'NumColumns', 4);
```

```
Warning: COMM.MATRIXHELICALSCANINTERLEAVER will be removed in a future release. Use HELSCANINTRL
```

```
deinterleaver = comm.MatrixHelicalScanDeinterleaver('NumRows',4,'NumColumns',4);
```

```
Warning: COMM.MATRIXHELICALSCANDEINTERLEAVER will be removed in a future release. Use HELSCANDEI
```

Generate random symbols. Pass the data through the interleaver, and then pass that data through the deinterleaver.

```
data = randi(7,16,1);
intData = interleaver(data);
deIntData = deinterleaver(intData);
```

Display the original sequence, interleaved sequence and restored sequence.

```
[data intData deIntData]
```

```
ans = 16×3

     6     6     6
     7     1     7
     1     2     1
     7     1     7
     5     5     5
     1     7     1
     2     6     2
     4     7     4
     7     7     7
     7     4     7
       ⋮
```

Confirm that the original and deinterleaved sequences are identical.

```
isequal(data,deIntData)
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Matrix Helical Scan Deinterleaver block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.MatrixHelicalScanInterleaver` will be removed
Not recommended starting in R2019b

comm.MatrixHelicalScanInterleaver will be removed in a future release. Use `helscanintrlv` instead.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
`deintrlv` | `helscandeintrlv` | `helscanintrlv` | `intrlv`

Introduced in R2012a

step

System object: `comm.MatrixHelicalScanInterleaver`
Package: `comm`

(To be removed) Permute input symbols by selecting matrix elements along diagonals

Compatibility

step will be removed in a future release. Use `helscanintrlv` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a NumRows × NumColumns long column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MemorylessNonlinearity

Package: `comm`

Apply memoryless nonlinearity to complex baseband signal

Description

The `comm.MemorylessNonlinearity` System object applies memoryless nonlinear impairments to a complex baseband signal. Use this System object to model memoryless nonlinear impairments caused by signal amplification in a radio frequency (RF) transmitter or receiver. For more information, see "Memoryless Nonlinear Impairments" on page 3-943.

Note All values of power assume a nominal impedance of 1 ohm.

To apply memoryless nonlinear impairments to a complex baseband signal:

1. Create the `comm.MemorylessNonlinearity` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
mnl = comm.MemorylessNonlinearity
mnl = comm.MemorylessNonlinearity(Name,Value)
```

Description

`mnl = comm.MemorylessNonlinearity` creates a memoryless nonlinearity System object that models RF nonlinear impairments.

`mnl = comm.MemorylessNonlinearity(Name,Value)` specifies properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `'Method','Saleh model'` sets the modeling method to the Saleh method.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Method` — Nonlinearity modeling method

`'Cubic polynomial'` (default) | `'Hyperbolic tangent'` | `'Saleh model'` | `'Ghorbani model'` | `'Rapp model'` | `'Lookup table'`

Nonlinearity modeling method, specified as `'Cubic polynomial'`, `'Hyperbolic tangent'`, `'Saleh model'`, `'Ghorbani model'`, `'Rapp model'`, or `'Lookup table'`. For more information, see “Memoryless Nonlinear Impairments” on page 3-943.

Data Types: `char` | `string`

`InputScaling` — Input signal scaling factor

`0` (default) | scalar

Input signal scaling factor in decibels, specified as a scalar. This property scales the power gain of the input signal.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property to `'Saleh model'` or `'Ghorbani model'`.

Data Types: `double`

`LinearGain` — Linear gain

`0` (default) | scalar

Linear gain in decibels, specified as a scalar. This property scales the power gain of the output signal.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property to `'Cubic polynomial'`, `'Hyperbolic tangent'`, or `'Rapp model'`.

Data Types: `double`

`IIP3` — Third-order input intercept point

`30` (default) | scalar

Third-order input intercept point in dBm, specified as a scalar.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property to `'Cubic polynomial'` or `'Hyperbolic tangent'`.

Data Types: `double`

`AMPMConversion` — AM/PM conversion factor

`10` (default) | scalar

AM/PM conversion factor in degrees per decibel, specified as a scalar. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 3-945.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property to `'Cubic polynomial'` or `'Hyperbolic tangent'`.

Data Types: `double`

`AMAMParameters` — AM/AM parameters

`[2.1587 1.1517]` | `[8.1081 1.5413 6.5202 -0.0718]` | row vector

AM/AM parameters used to compute the amplitude gain for an input signal, specified as a row vector.

- When the `Method` property is set to `'Saleh model'`, this property must be a two-element vector that specifies alpha and beta values. In this case, the default value is `[2.1587 1.1517]`.
- When the `Method` property is set to `'Ghorbani model'`, this property must be a four-element vector that specifies x_1, x_2, x_3, and x_4 values. In this case, the default value is `[8.1081 1.5413 6.5202 -0.0718]`.

For more information, see “Saleh Model Method” on page 3-945 and “Ghorbani Model Method” on page 3-946.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Saleh model'` or `'Ghorbani model'`.

Data Types: `double`

`AMPMParameters` — AM/PM parameters

`[4.0033 9.1040]` | `[4.6645 2.0965 10.88 -0.003]` | row vector

AM/PM parameters used to compute the phase change for an input signal, specified as a row vector.

- When the `Method` property is set to `'Saleh model'`, this property must be a two-element vector that specifies alpha and beta values. In this case, the default value is `[4.0033 9.1040]`.
- When the `Method` property is set to `'Ghorbani model'`, this property must be a four-element vector that specifies y_1, y_2, y_3, and y_4 values. In this case, the default value is `[4.6645 2.0965 10.88 -0.003]`

For more information, see “Saleh Model Method” on page 3-945 and “Ghorbani Model Method” on page 3-946.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Saleh model'` or `'Ghorbani model'`.

Data Types: `double`

`PowerLowerLimit` — Input power lower limit

`10` (default) | scalar

Input power lower limit in dBm, specified as a scalar less than the `PowerUpperLimit` property value. The AM/PM conversion scales linearly for input power values in the range [`PowerLowerLimit`, `PowerUpperLimit`]. If the input signal power is below the input power lower limit, the phase shift resulting from AM/PM conversion is zero. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 3-945.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Cubic polynomial'` or `'Hyperbolic tangent'`.

Data Types: `double`

`PowerUpperLimit` — Input power upper limit

`inf` (default) | scalar

Input power upper limit in dBm, specified as a scalar greater than `PowerLowerLimit`. The AM/PM conversion scales linearly for input power values in the range [`PowerLowerLimit`, `PowerUpperLimit`]. If the input signal power is above the input power upper limit, the phase shift resulting from AM/PM conversion is constant. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 3-945.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Cubic polynomial'` or `'Hyperbolic tangent'`.

Data Types: `double`

`OutputScaling` — Output signal scaling factor

`0` (default) | scalar

Output signal scaling factor in decibels, specified as a scalar. This property scales the power gain of the output signal.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Saleh model'` or `'Ghorbani model'`.

Data Types: `double`

`Smoothness` — Smoothness factor

`0.5` (default) | scalar

Smoothness factor, specified as a scalar. For more information, see “Rapp Model Method” on page 3-947.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Rapp model'`.

Data Types: `double`

`OutputSaturationLevel` — Output saturation level

1 (default) | scalar

Output saturation level, specified as a scalar. For more information, see "Rapp Model Method" on page 3-947.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Rapp model'`.

Data Types: `double`

`Table` — Amplifier characteristics lookup table

N-by-[P_{in}, P_{out}, $\Delta\Phi$] matrix

Amplifier characteristics lookup table, specified as an N-by-3 matrix of measured power amplifier (PA) characteristics. Each row is of the form [P_{in}, P_{out}, $\Delta\Phi$]. P_{in} specifies the PA input signal in dBm, P_{out} specifies the PA output signal in dBm, and $\Delta\Phi$ specifies the output phase shift in degrees. The default value is `[-25, 5.16, -0.25; -20, 10.11, -0.47; -15, 15.11, -0.68; -10, 20.05, -0.89; -5, 24.79, -1.22; 0, 27.64, 5.59; 5, 28.49, 12.03]`.

The measured PA characteristics defined by this property are used to compute the AM/AM (in dBm/dBm) and AM/PM (in deg/dBm) nonlinear impairment characteristics. The System object distorts the input signal by the computed AM/AM (in dBm/dBm) and AM/PM (in deg/dBm) values.

Note To determine appropriate P_{out} and $\Delta\Phi$ for P_{in} values outside the range of values specified in the `Table` property, the System object applies linear extrapolation from the first two or last two [P_{in}, P_{out}, $\Delta\Phi$] rows of `Table`.

Tunable: Yes

Dependencies

To enable this property, set the `Method` property is set to `'Lookup table'`.

Data Types: `double`

Usage

Syntax

```
outsig = mnl(insig)
```

Description

`outsig = mnl(insig)` applies memoryless nonlinear impairments to the input RF baseband signal.

Input Arguments

`insig` — Input RF baseband signal

scalar | column vector

Input RF baseband signal, specified as a scalar or column vector. Values in this input must be complex.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`outsig` — Output RF baseband signal
scalar | column vector

Output RF baseband signal, returned as a scalar or column vector. The output is of the same data type as the input.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Apply Saleh Model of Power Amplifier Nonlinearity to 16-QAM Signal

Generate 16-QAM data with an average power of 10 mW and pass the data through a nonlinear power amplifier (PA).

```
M = 16;
data = randi([0 (M - 1)]',1000,1);
avgPow = 1e-2;
minD = avgPow2MinD(avgPow,M);
```

Create a memoryless nonlinearity System object, specifying the Saleh model method.

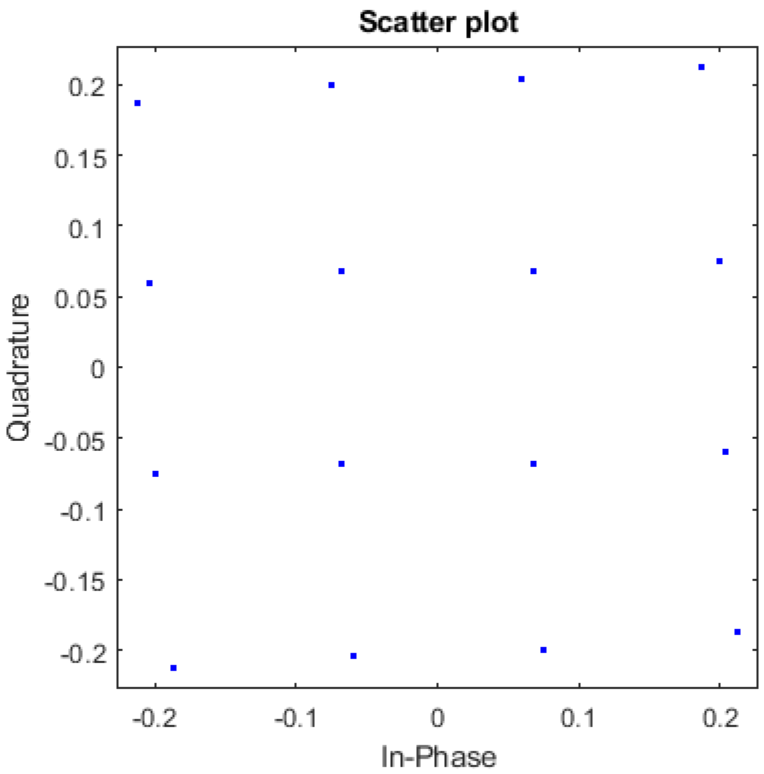
```
saleh = comm.MemorylessNonlinearity('Method','Saleh model');
```

Generate modulated symbols and pass them through the PA nonlinearity model.

```
modData = (minD/2).*qammod(data,M);
y = saleh(modData);
```

Generate a scatter plot of the results.

```
scatterplot(y)
```



Average power normalization of input signal.

```
function minD = avgPow2MinD(avgPow,M)
    % Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits>4)
            sf = ((31*M/32) - 1)/6;
        else
            sf = ((5*M/4) - 1)/6;
        end
    end
    minD = sqrt(avgPow/sf);
end
```

Nonlinear Amplifier Gain Compression

Plot the gain compression of a nonlinear amplifier for a 16-QAM signal.

Specify the modulation order and samples per symbol parameters.

```
M = 16;
sps = 4;
```

Model a nonlinear amplifier, by creating a memoryless nonlinearity System object with a 30 dB third-order input intercept point. Create a raised cosine transmit filter System object.

```
amplifier = comm.MemorylessNonlinearity('IIP3',30);

txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.3, ...
    'FilterSpanInSymbols',6, ...
    'OutputSamplesPerSymbol',sps, ...
    'Gain',sqrt(sps));
```

Specify the input power in dBm. Convert the input power to W and initialize the gain vector.

```
pindBm = -5:25;
pin = 10.^((pindBm-30)/10);
gain = zeros(length(pindBm),1);
```

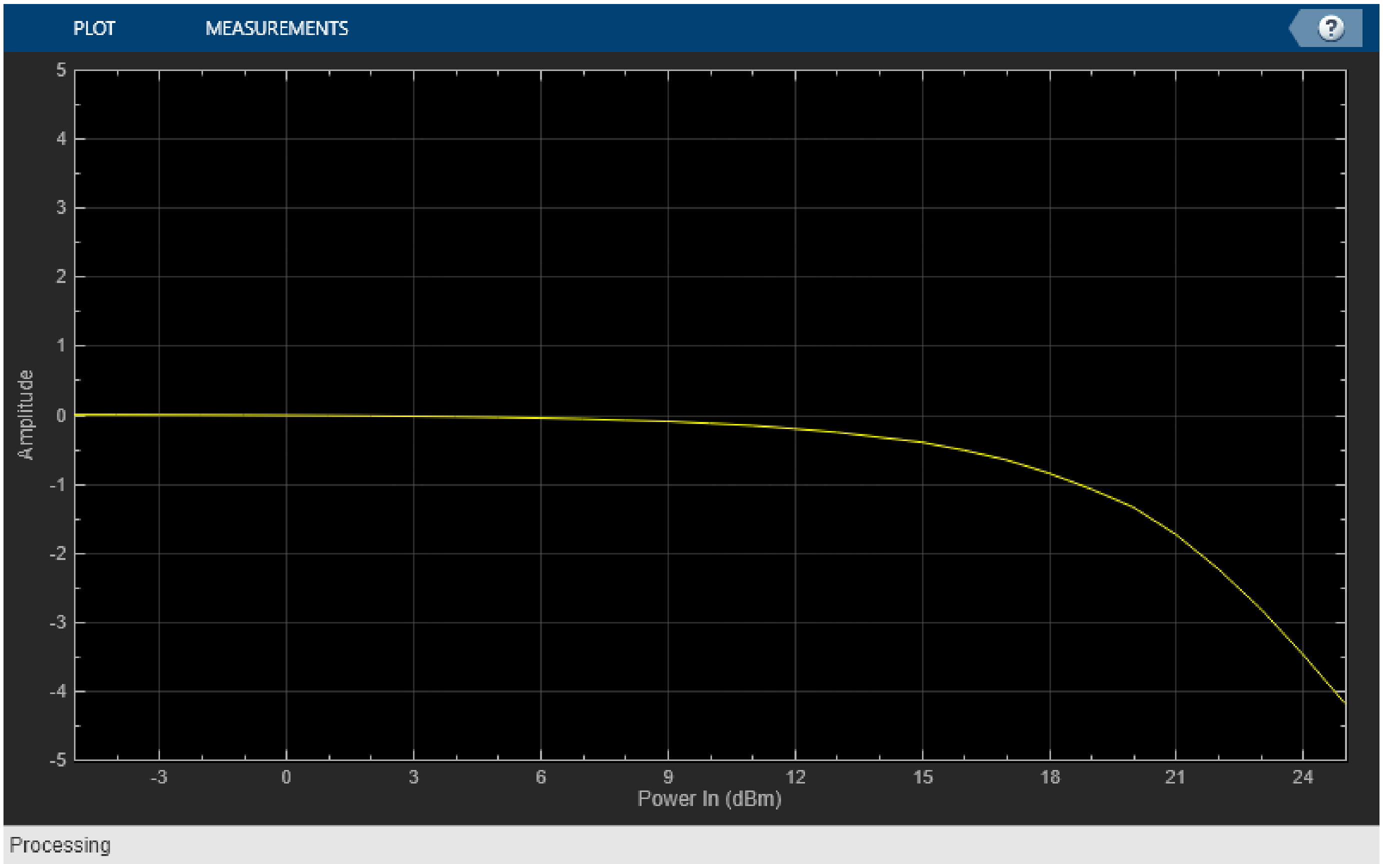
Execute the main processing loop, which includes these steps.

- Generate random data symbols.
- Modulate the data symbols and adjust the average power of the signal.
- Filter the modulated signal.
- Amplify the signal.
- Measure the gain.

```
for k = 1:length(pin)
    data = randi([0 (M - 1)],1000,1);
    modSig = qammod(data,M,'UnitAveragePower',true)*sqrt(pin(k));
    filtSig = txfilter(modSig);
    ampSig = amplifier(filtSig);
    gain(k) = 10*log10(var(ampSig)/var(filtSig));
end
```

Plot the amplifier gain as a function of the input signal power. The 1 dB gain compression point occurs for an input power of 18.5 dBm. To increase the point at which a 1 dB compression is observed, increase the third-order intercept point, `amplifier.IIP3`.

```
arrayplot = dsp.ArrayPlot('PlotType','Line','XLabel','Power In (dBm)', ...
    'XOffset',-5,'YLimits',[-5 5]);

arrayplot(gain)
```



Distort 16-QAM Signal with Custom Power Amplifier Nonlinearities

Apply nonlinear power amplifier (PA) characteristics to a 16-QAM signal by setting the `Method` property to `'Lookup table'`.

Define parameters for the modulation order, samples per symbol, and input power. Create random data.

```
M = 16; % Modulation order
sps = 4; % Samples per symbol
pindBm = -2; % Input power
pin = 10.^((pindBm-30)/10); % power in Watts
data = randi([0 (M - 1)],1000,1);
refdata = 0:M-1;
refconst = qammod(refdata,M,'UnitAveragePower',true);
```

Create a memoryless nonlinearity System object, a transmit filter System object, and a constellation diagram System object. The default lookup table values are used for the memoryless nonlinearity System object.

```
amplifier = comm.MemorylessNonlinearity('Method','Lookup table');
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.3, ...
    'FilterSpanInSymbols',6,'OutputSamplesPerSymbol',sps,'Gain',sqrt(sps));
constellation = comm.ConstellationDiagram('SamplesPerSymbol',4,'ReferenceConstellation',refconst
    'Title','Amplified/Distorted Signal');
```

Modulate the random data. Filter and apply the nonlinear amplifier characteristics to the modulation symbols.

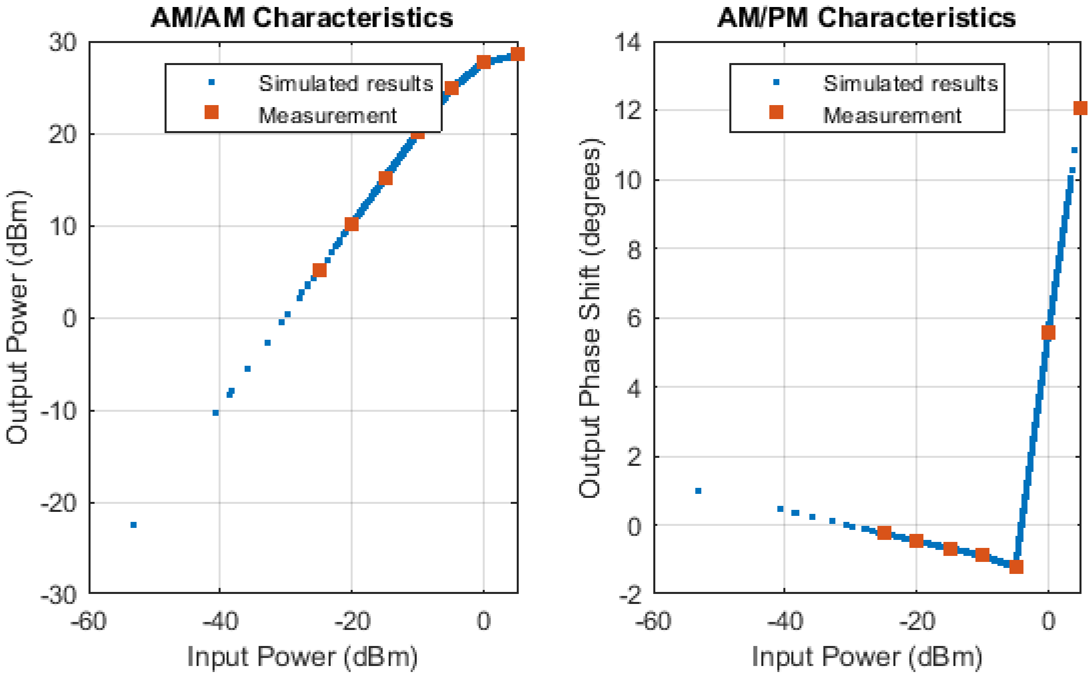
```
modSig = qammod(data,M,'UnitAveragePower',true)*sqrt(pin);
filtSig = txfilter(modSig);
ampSig = amplifier(filtSig);
```

Compute input and output signal levels and the phase shift.

```
poutdBm = (20*log10(abs(ampSig))) + 30;
simulated_pindBm = (20*log10(abs(filtSig))) + 30;
phase = angle(ampSig.*conj(filtSig))*180/pi;
```

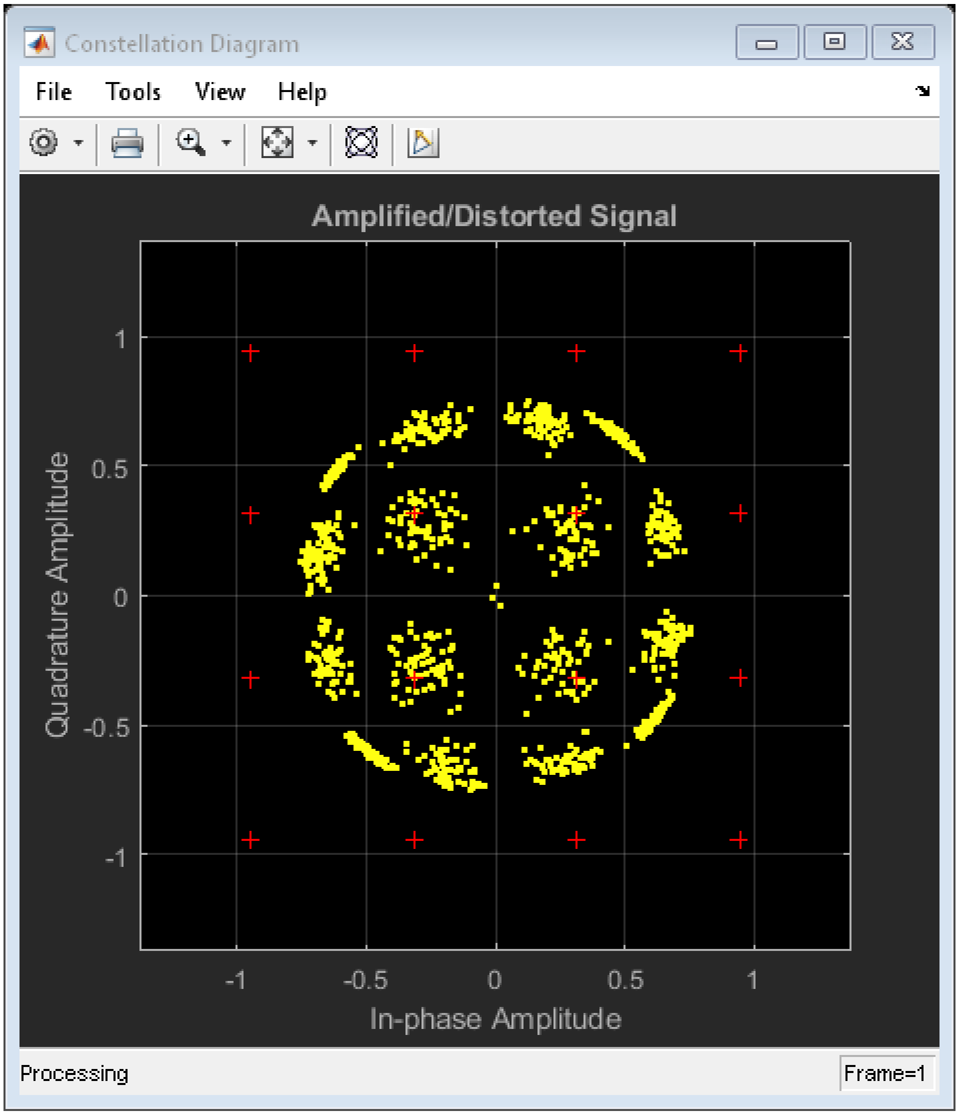
Plot AM/AM characteristics, AM/PM characteristics, and the constellation results.

```
figure
set(gcf,'units','normalized','position',[.25 1/3 .5 1/3])
subplot(1,2,1)
plot(simulated_pindBm,poutdBm,'.');
hold on
plot(amplifier.Table(:,1),amplifier.Table(:,2),'.','Markersize',15);
xlabel('Input Power (dBm)')
ylabel('Output Power (dBm)');
grid on;
title('AM/AM Characteristics');
leglabels = {'Simulated results','Measurement'};
legend (leglabels,'Location','north');

subplot(1,2,2)
plot(simulated_pindBm,phase,'.');
hold on
plot(amplifier.Table(:,1),amplifier.Table(:,3),'.','Markersize',15);
legend (leglabels,'Location','north');
xlabel('Input Power (dBm)');
ylabel('Output Phase Shift (degrees)');
grid on; title('AM/PM Characteristics');
```



Generate a constellation diagram of the amplified signal and reference constellation. The nonlinear amplifier characteristics cause compression of the amplified signal constellation compared to the reference constellation.

```
constellation(ampSig)
```



Distort 16-QAM Signal with Measured Power Amplifier Nonlinearities

Apply nonlinear power amplifier (PA) characteristics to a 16-QAM signal by setting the `Method` property to `'Lookup table'`.

Define parameters for the modulation order, samples per symbol, and input power. Create random data.

```
M = 16; % Modulation order
sps = 4; % Samples per symbol
pindBm = -8; % Input power
pin = 10.^((pindBm-30)/10); % power in Watts
data = randi([0 (M - 1)],1000,1);
refdata = 0:M-1;
refconst = qammod(refdata,M,'UnitAveragePower',true);
paChar = pa_performance_characteristics();
```

Create a memoryless nonlinearity System object, a transmit filter System object, and a constellation diagram System object. The default lookup table values are used for the memoryless nonlinearity System object.

```
amplifier = comm.MemorylessNonlinearity('Method','Lookup table','Table',paChar);
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.3, ...
    'FilterSpanInSymbols',6,'OutputSamplesPerSymbol',sps,'Gain',sqrt(sps));
constellation = comm.ConstellationDiagram('SamplesPerSymbol',4, ...
    'Title','Amplified/Distorted Signal','NumInputPorts',2, ...
    'ReferenceConstellation', refconst,'ShowLegend',true, ...
    'ChannelNames',{'Filtered signal','Amplified signal'});
```

Modulate the random data. Filter and apply the nonlinear amplifier characteristics to the modulation symbols.

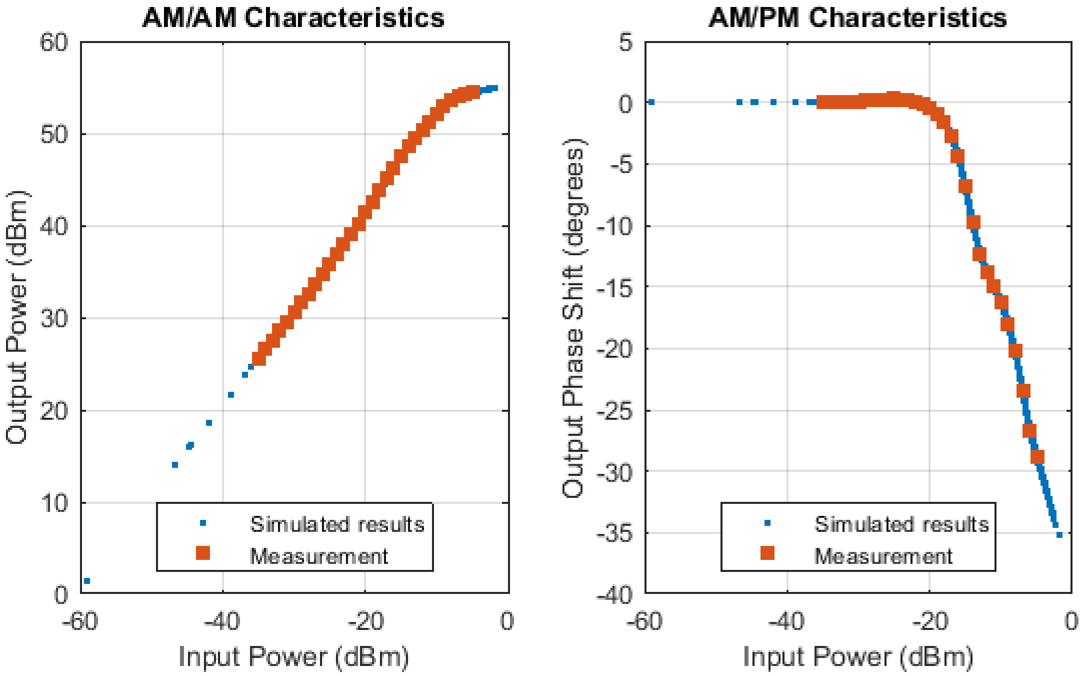
```
modSig = qammod(data,M,'UnitAveragePower',true)*sqrt(pin);
filtSig = txfilter(modSig);
ampSig = amplifier(filtSig);
```

Compute input and output signal levels and the phase shift.

```
poutdBm = (20*log10(abs(ampSig))) + 30;
simulated_pindBm = (20*log10(abs(filtSig))) + 30;
phase = angle(ampSig.*conj(filtSig))*180/pi;
```

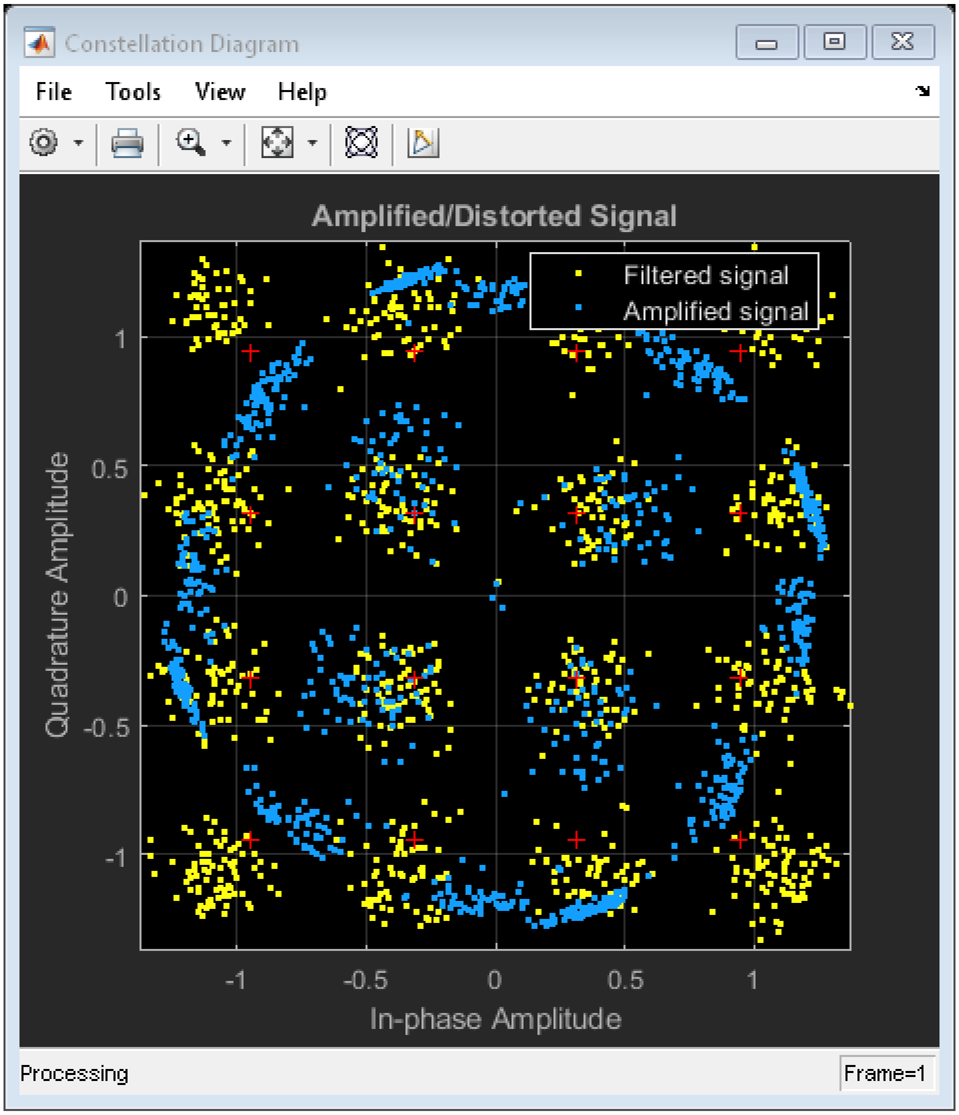
Plot AM/AM characteristics, AM/PM characteristics, and the constellation results.

```
figure
set(gcf,'units','normalized','position',[.25 1/3 .5 1/3])
subplot(1,2,1)
plot(simulated_pindBm,poutdBm,'.');
hold on
plot(amplifier.Table(:,1),amplifier.Table(:,2),'.','Markersize',15);
xlabel('Input Power (dBm)')
ylabel('Output Power (dBm)');
grid on;
title('AM/AM Characteristics');
leglabel = {'Simulated results','Measurement'};
legend (leglabel,'Location','south');

subplot(1,2,2)
plot(simulated_pindBm,phase,'.');
hold on
plot(amplifier.Table(:,1),amplifier.Table(:,3),'.','Markersize',15);
legend (leglabel,'Location','south');
xlabel('Input Power (dBm)');
ylabel('Output Phase Shift (degrees)');
grid on;
title('AM/PM Characteristics');
```



For the purpose of constellation comparison, normalize the amplified signal and the filtered signal. Generate a constellation diagram of the filtered signal and amplified signal. The nonlinear amplifier characteristics cause compression of the amplified signal constellation compared to the filtered constellation.

```
filtSig = filtSig/mean(abs(filtSig)); % Normalized filtered signal
ampSig = ampSig/mean(abs(ampSig)); % Normalized amplified signal
constellation(filtSig,ampSig)
```



The `pa_performance_characteristics` helper function calculates the amplifier performance characteristics. The data is extracted from figure 4 of Hammi, Oualid, et al. "Power amplifiers' model assessment and memory effects intensity quantification using memoryless post-compensation technique." IEEE Transactions on Microwave Theory and Techniques 56.12 (2008): 3170-3179.

```
function paChar = pa_performance_characteristics()
```

The operating specification for the LDMOS-based Doherty amplifier are:

- A frequency of 2110 MHz
- A peak power of 300 W
- A small signal gain of 61 dB

Each row in `HAV08_Table` specifies Pin (dBm), gain (dB), phase shift (degrees).

```
HAV08_Table =...
    [-35,60.53,0.01;
    -34,60.53,0.01;
    -33,60.53,0.08;
    -32,60.54,0.08;
    -31,60.55,0.1;
    -30,60.56,0.08;
    -29,60.57,0.14;
    -28,60.59,0.19;
    -27,60.6,0.23;
    -26,60.64,0.21;
    -25,60.69,0.28;
    -24,60.76,0.21;
    -23,60.85,0.12;
    -22,60.97,0.08;
    -21,61.12,-0.13;
    -20,61.31,-0.44;
    -19,61.52,-0.94;
    -18,61.76,-1.59;
    -17,62.01,-2.73;
    -16,62.25,-4.31;
    -15,62.47,-6.85;
    -14,62.56,-9.82;
    -13,62.47,-12.29;
    -12,62.31,-13.82;
    -11,62.2,-15.03;
    -10,62.15,-16.27;
    -9,62,-18.05;
    -8,61.53,-20.21;
    -7,60.93,-23.38;
    -6,60.2,-26.64;
    -5,59.38,-28.75];
```

Convert the second column of the HAV08_Table from gain to Pout for use by the memoryless nonlinearity System object.

```
paChar = HAV08_Table;
paChar(:,2) = paChar(:,1) + paChar(:,2);
end
```

More About

Memoryless Nonlinear Impairments

Memoryless nonlinear impairments distort the input signal amplitude and phase. The amplitude distortion is amplitude-to-amplitude modulation (AM/AM) and the phase distortion is amplitude-to-phase modulation (AM/PM).

Model Method	Memoryless Nonlinear Impairment
Cubic polynomial	AM/AM and AM/PM
Hyperbolic tangent	
Saleh model	
Ghorbani model	
Rapp model	AM/AM only

Model Method	Memoryless Nonlinear Impairment
Lookup table	Applies impairment according to [P_{in}, P_{out}, ΔΦ] amplifier characteristics specified by the `Table` property

The modeled impairments apply the AM/AM and AM/PM distortions differently, according to the model method you specify. The models apply the memoryless nonlinear impairment to the input signal by following these steps.

1 Multiply the signal by an input gain factor.

Note You can normalize the signal to 1 by setting the input scaling gain to the inverse of the input signal amplitude.

2 Split the complex signal into its magnitude and angle components.

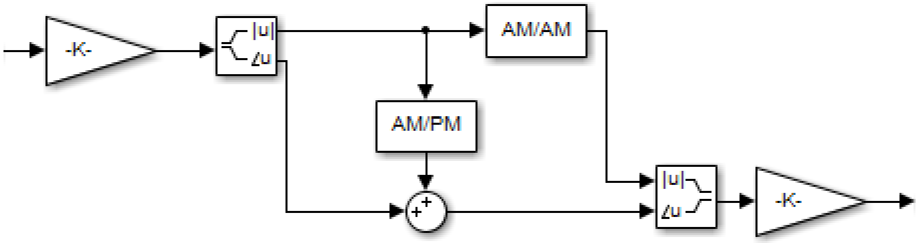
3 Apply an AM/AM distortion to the magnitude of the signal, according to the selected model method, to produce the magnitude of the output signal.

4 Apply an AM/PM distortion to the phase of the signal, according to the selected model method, to produce the angle of the output signal.

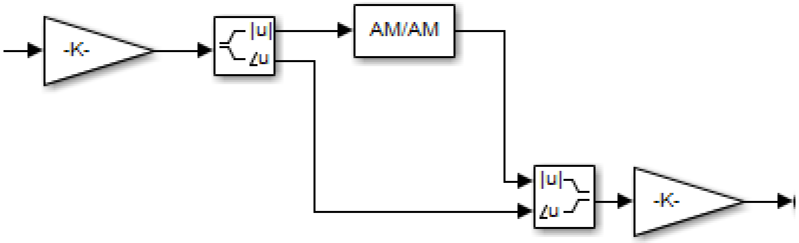
Note This step does not apply for the Rapp model.

5 Combine the new magnitude and angle components into a complex signal. Then, multiply the result by an output gain factor.

The first four model methods (cubic polynomial, hyperbolic tangent, Saleh model, and Ghorbani model) apply AM/AM and AM/PM impairments as shown in this figure.



The Rapp model method applies AM/AM distortion as shown in this figure.



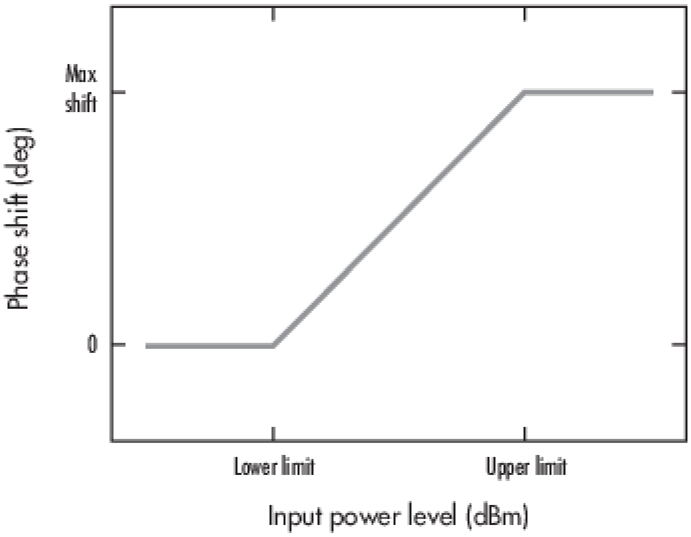
The lookup table method uses the power amplifier (PA) characteristics lookup table, specified as an *N*-by-3 matrix of measured power amplifier (PA) characteristics. Each row is of the form [P_{in}, P_{out},

$\Delta\Phi$]. P_{in} specifies the PA input signal in dBm, P_{out} specifies the PA output signal in dBm, and $\Delta\Phi$ specifies the output phase shift in degrees. The measured PA characteristics defined by the `Table` property are used to compute the AM/AM (in dBm/dBm) and AM/PM (in deg/dBm) nonlinear impairment characteristics. The System object distorts the input signal by the computed AM/AM (in dBm/dBm) and AM/PM (in deg/dBm) values.

Note To determine appropriate P_{out} and $\Delta\Phi$ for P_{in} values outside the range of values specified in the `Table` property, the System object applies linear extrapolation from the first two or last two [P_{in}, P_{out}, $\Delta\Phi$] rows of `Table`.

Cubic Polynomial and Hyperbolic Tangent Model Methods

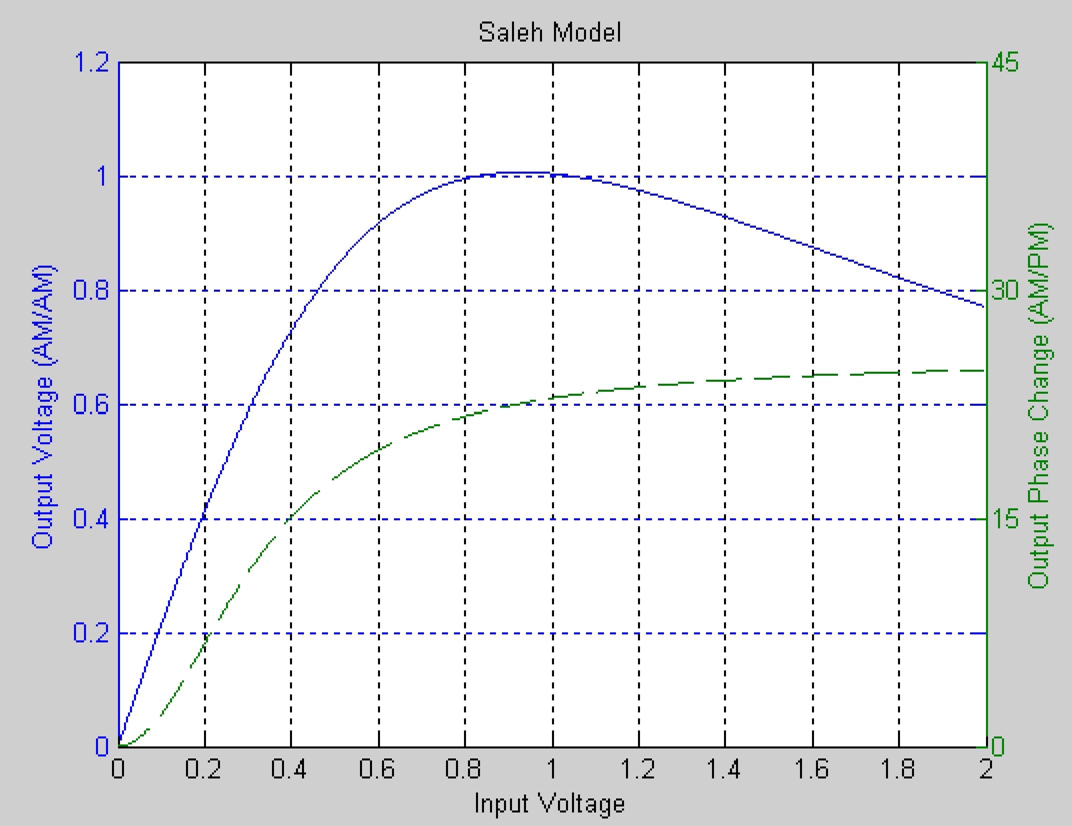
This figure shows the AM/PM conversion behavior for the cubic polynomial and hyperbolic tangent model methods.



The AM/PM conversion scales linearly with an input power value between the lower and upper limits of the input power level. Outside this range, the AM/PM conversion is constant at the values corresponding to the lower and upper input power limits, which are zero and (*AM/PM conversion*) × (*upper input power limit – lower input power limit*), respectively.

Saleh Model Method

This figure shows the AM/AM behavior (output voltage versus input voltage for the AM/AM distortion) and the AM/PM behavior (output phase versus input voltage for the AM/PM distortion) for the Saleh model method.



The AM/AM parameters, α_{AMAM} and β_{AMAM}, are used to compute the amplitude distortion of the input signal by using

$$F_{\mathrm{AMAM}}(u) = \frac{\alpha_{\mathrm{AMAM}} \times u}{1 + \beta_{\mathrm{AMAM}} \times u^2},$$

where u is the magnitude of the scaled signal.

The AM/PM parameters, α_{AMPM} and β_{AMPM}, are used to compute the phase distortion of the input signal by using

$$F_{\mathrm{AMPM}}(u) = \frac{\alpha_{\mathrm{AMPM}} \times u^2}{1 + \beta_{\mathrm{AMPM}} \times u^2},$$

where u is the magnitude of the scaled signal. The α and β parameters for AM/AM and AM/PM are similarly named but distinct.

Ghorbani Model Method

The Ghorbani model method applies AM/AM and AM/PM distortion as described in this section.

The AM/AM parameters (x_1, x_2, x_3, and x_4) are used to compute the amplitude distortion of the input signal by using

$$F_{\mathrm{AMAM}}(u) = \frac{x_1 u^{x_2}}{1 + x_3 u^{x_2}} + x_4 u,$$

where u is the magnitude of the scaled signal.

The AM/PM parameters (y_1, y_2, y_3, and y_4) are used to compute the phase distortion of the input signal by using

$$F_{\mathrm{AMPM}}(u) = \frac{y_1 u^{y_2}}{1 + y_3 u^{y_2}} + y_4 u,$$

where u is the magnitude of the scaled signal.

Rapp Model Method

The Rapp model method applies AM/AM distortion as described in this section. The Rapp model does not apply AM/PM distortion to the input signal.

The smoothness factor and output saturation level are used to compute the amplitude distortion of the input signal given by

$$F_{\mathrm{AMAM}}(u) = \frac{u}{\left(1 + \left(\frac{u}{O_{\mathrm{sat}}}\right)^{2\mathrm{S}}\right)^{1/2\mathrm{S}}},$$

where

- u is the magnitude of the scaled signal.
- S is the smoothness factor.
- O_{sat} is the output saturation level.

References

[1] Saleh, A.A.M. “Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers.” *IEEE Transactions on Communications* 29, no. 11 (November 1981): 1715–20. https://doi.org/10.1109/TCOM.1981.1094911.

[2] Ghorbani, A., and M. Sheikhan. "The Effect of Solid State Power Amplifiers (SSPAs) Nonlinearities on MPSK and M-QAM Signal Transmission." In *1991 Sixth International Conference on Digital Processing of Signals in Communications*, 193–97, 1991.

[3] Rapp, Ch. "Effects of HPA-Nonlinearity on a 4-DPSK/OFDM-Signal for a Digital Sound Broadcasting System." In *Proceedings Second European Conf. on Sat. Comm. (ESA SP-332)*, 179–84. Liege, Belgium, 1991. https://elib.dlr.de/33776/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.PhaseNoise`

Blocks
Memoryless Nonlinearity

Introduced in R2012a

comm.MER

Package: comm

Measure modulation error ratio

Description

The comm.MER (modulation error ratio) object measures the signal-to-noise ratio (SNR) in digital modulation applications. You can use MER measurements to determine system performance in communications applications. For example, determining whether a DVB-T system conforms to applicable radio transmission standards requires accurate MER measurements. The block measures all outputs in dB.

To measure modulation error ratio:

1. Define and set up your MER object. See “Construction” on page 3-949.
2. Call step to measure the modulation error ratio according to the properties of comm.MER. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

MER = comm.MER creates a modulation error ratio (MER) System object, MER. This object measures the signal-to-noise ratio (SNR) in digital modulation applications.

MER = comm.MER(Name,Value) creates an MER object with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Example: MER = comm.MER('ReferenceSignalSource','Estimated from reference constellation') creates an object, MER, that measures the MER of a received signal by using a reference constellation.

Properties

ReferenceSignalSource

Reference signal source

Reference signal source, specified as either 'Input port' (default) or 'Estimated from reference constellation'. To provide an explicit reference signal against which the input signal is measured, set this property to 'Input port'. To measure the MER of the input signal against a reference constellation, set this property to 'Estimated from reference constellation'.

`ReferenceConstellation`

Reference constellation

Reference constellation, specified as a vector. This property is available when the `ReferenceSignalSource` property is `'Estimated from reference constellation'`.

The default is `[0.7071 - 0.7071i; -0.7071 - 0.7071i; -0.7071 + 0.7071i; 0.7071 + 0.7071i]`, which corresponds to a standard QPSK constellation. You can derive constellation points by using modulation functions or objects. For example, to derive the reference constellation for a 16-QAM signal, you can use `qammod(0:15,16)`.

`MeasurementIntervalSource`

Measurement interval source

Measurement interval source, specified as one of the following: `'Input length'` (default), `'Entire history'`, `'Custom'`, or `'Custom with periodic reset'`. This property affects the RMS and maximum MER outputs only.

- To calculate MER using only the current samples, set this property to `'Input length'`.
- To calculate MER for all samples, set this property to `'Entire history'`.
- To calculate MER over an interval you specify and to use a sliding window, set this property to `'Custom'`.
- To calculate MER over an interval you specify and to reset the object each time the measurement interval is filled, set this property to `'Custom with periodic reset'`.

`MeasurementInterval`

Measurement interval

Measurement interval over which the MER is calculated, specified in samples as a real positive integer. This property is available when `MeasurementIntervalSource` is `'Custom'` or `'Custom with periodic reset'`. The default is `100`.

`AveragingDimensions`

Averaging dimensions

Averaging dimensions, specified as a positive integer or row vector of positive integers. This property determines the dimensions over which the averaging is performed. For example, to average across the rows, set this property to `2`. The default is `1`.

The object supports variable-size inputs over the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must remain constant between `step` calls. For example, if the input has size `[4 3 2]` and `Averaging dimensions` is `[1 3]`, the output size is `[1 3 1]`, and the second dimension must remain fixed at `3`.

`MinimumMEROutputPort`

Minimum MER measurement output port

Minimum MER measurement output port, specified as a logical scalar. To create an output port for minimum MER measurements, set this property to `true`. The default is `false`.

`XPercentileMEROutputPort`

X-percentile MER measurement output port

X-percentile MER measurement output port, specified as a logical scalar. To create an output port for *X*-percentile MER measurements, set this property to `true`. The *X*-percentile MER measurements persist until you reset the object. These measurements are calculated by using all of the input frames since the last reset. The default is `false`.

`XPercentileValue`

X-percentile value

X-percentile value above which *X*% of the MER measurements fall, specified as a real scalar from `0` to `100`. This property applies when `XPercentileMEROutputPort` is `true`. The default is `95`.

`SymbolCountOutputPort`

Symbol count output port

Symbol count output port, specified as a logical scalar. To output the number of accumulated symbols used to calculate the *X*-percentile MER measurements, set this property to `true`. This property is available when `XPercentileMEROutputPort` property is `true`. The default is `false`.

Methods

reset	Reset states of MER measurement object
step	Measure modulation error ratio

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Measure MER of Noisy 16-QAM Modulated Signal

Create an MER object which outputs minimum MER, 90-percentile MER, and the number of symbols.

```
mer = comm.MER('MinimumMEROutputPort',true, ...
    'XPercentileMEROutputPort',true,'XPercentileValue',90,...
    'SymbolCountOutputPort',true);
```

Generate random data. Apply 16-QAM modulation having unit average power. Pass the signal through an AWGN channel.

```
data = randi([0 15],1000,1);
refsym = qammod(data,16,'UnitAveragePower',true);
rxsym = awgn(refsym,20);
```

Determine the RMS, minimum, and 90th percentile MER values.

```
[MERdB,MinMER,PercentileMER,NumSym] = mer(refsym,rxsym)
```

```
MERdB = 20.1071

MinMER = 11.4248

PercentileMER = 16.5850

NumSym = 1000
```

Measure MER Using Reference Constellation

Generate random data symbols, and apply 8-PSK modulation.

```
d = randi([0 7],2000,1);
txSig = pskmod(d,8,pi/8);
```

Pass the modulated signal through an AWGN channel.

```
rxSig = awgn(txSig,30);
```

Create an MER object. Measure the MER using the transmitted signal as the reference.

```
mer = comm.MER;
mer1 = mer(txSig,rxSig);
```

Release the MER object. Set the object to use a reference constellation for making MER measurements.

```
release(mer)
mer.ReferenceSignalSource = 'Estimated from reference constellation';
mer.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Measure the MER using only the received signal as an input. Verify that it matches the result obtained with a reference signal.

```
mer2 = mer(rxSig);
[mer1 mer2]
```

```
ans = 1×2

   30.0271   30.0271
```

Measure MER Using Custom Measurement Interval

Measure the MER of a noisy 8-PSK signal using two types of custom measurement intervals. Display the results.

Set the number of frames, M, and the number of subframes per frame, K.

```
M = 2;
K = 5;
```

Set the number of symbols in a subframe. Calculate the corresponding frame length.

```
sfLen = 100;
frmLen = K*sfLen

frmLen = 500
```

Create an MER object. Configure the object to use a custom measurement interval equal to the frame length.

```
mer1 = comm.MER('MeasurementIntervalSource','Custom', ...
    'MeasurementInterval',frmLen);
```

Configure the object to measure MER using an 8-PSK reference constellation.

```
mer1.ReferenceSignalSource = 'Estimated from reference constellation';
mer1.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Create an MER object, and configure it use a 500-symbol measurement interval with a periodic reset. Configure the object to measure MER using an 8-PSK reference constellation.

```
mer2 = comm.MER('MeasurementIntervalSource','Custom with periodic reset', ...
    'MeasurementInterval',frmLen);
mer2.ReferenceSignalSource = 'Estimated from reference constellation';
mer2.ReferenceConstellation = pskmod(0:7,8,pi/8);
```

Initialize the MER and signal-to-noise arrays.

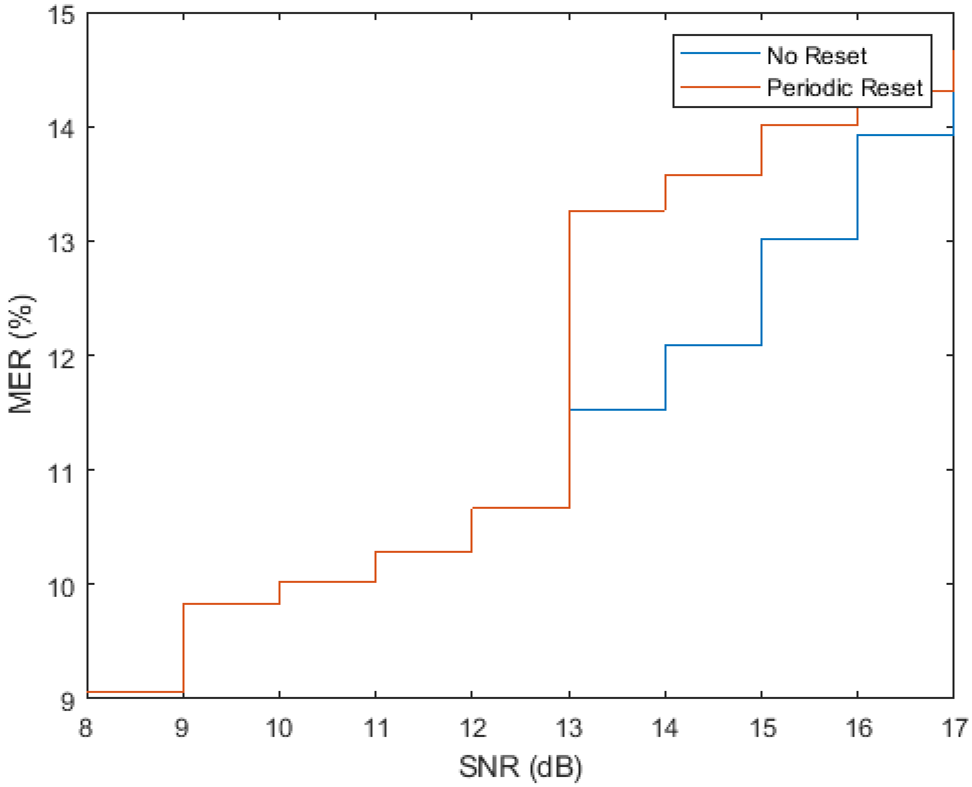
```
merNoReset = zeros(K,M);
merReset = zeros(K,M);
snrdB = zeros(K,M);
```

Measure the MER for a noisy 8-PSK signal using both objects. The SNR is increases by 1 dB from subframe to subframe. For `merNoReset`, the 500 most recent symbols are used to compute the estimate. In this case, a sliding window is used so that an entire data frame is used as the basis for the estimate. For `merReset`, the symbols are cleared each time a new frame is encountered.

```
for m = 1:M
    for k = 1:K
        data = randi([0 7],sfLen,1);
        txSig = pskmod(data,8,pi/8);
        snrdB(k,m) = k+(m-1)*K+7;
        rxSig = awgn(txSig,snrdB(k,m));
        merNoReset(k,m) = mer1(rxSig);
        merReset(k,m) = mer2(rxSig);
    end
end
```

Display the MER measured using the two approaches. The windowing used in the first case provides an averaging across the subframes. In the second case, the MER object resets after the first frame so that the calculated MER values more accurately reflect the current SNR.

```
stairs(snrdB(:),[merNoReset(:) merReset(:)])
xlabel('SNR (dB)')
ylabel('MER (%)')
legend('No Reset','Periodic Reset')
```



Measure MER Across Different Dimensions

Create OFDM modulator and demodulator objects.

```
ofdmmod = comm.OFDMModulator('FFTLength',32,'NumSymbols',4);
ofdmdemod = comm.OFDMDemodulator('FFTLength',32,'NumSymbols',4);
```

Determine the number of subcarriers and symbols in the OFDM signal.

```
ofdmDims = info(ofdmmod);
numSC = ofdmDims.DataInputSize(1)
```

```
numSC = 21
```

```
numSym = ofdmDims.DataInputSize(2)
```

```
numSym = 4
```

Generate random symbols and apply QPSK modulation.

```
msg = randi([0 3],numSC,numSym);
modSig = pskmod(msg,4,pi/4);
```

OFDM modulate the QPSK signal. Pass the signal through an AWGN channel. Demodulate the noisy signal.

```
txSig = ofdmmod(modSig);
rxSig = awgn(txSig,10,'measured');
demodSig = ofdmdemod(rxSig);
```

Create an MER object, where the result is averaged over the subcarriers. Measure the MER. There are four entries corresponding to each of the 4 OFDM symbols.

```
mer = comm.MER('AveragingDimensions',1);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 1×4

   11.2338   12.5315   12.8882   12.7015
```

Overwrite the MER object, where the result is averaged over the OFDM symbols. Measure the MER. There are 21 entries corresponding to each of the 21 subcarriers.

```
mer = comm.MER('AveragingDimensions',2);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 21×1

   10.8054
   14.9655
   14.5721
   13.6024
   13.0132
   12.1391
   10.4012
    9.5017
    8.8055
   13.3824
      ⋮
```

Measure the MER and average over both the subcarriers and the OFDM symbols.

```
mer = comm.MER('AveragingDimensions',[1 2]);
modErrorRatio = mer(demodSig,modSig)
```

```
modErrorRatio = 12.2884
```

Algorithms

MER is a measure of the SNR in a modulated signal calculated in dB. The MER over *N* symbols is

$$\mathrm{MER} = 10 \cdot \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}(e_k)}\right) \mathrm{dB},$$

The MER for the *k*th symbol is

$$MER_k = 10 * \log_{10}\left(\frac{\frac{1}{N}\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{e_k}\right) \text{dB}.$$

The minimum MER represents the minimum MER value in a burst, or

$$MER_{\min} = \min_{k \in [1, \ldots, N]} \{MER_k\} ,$$

where:

- $e_k = e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k = In-phase measurement of the kth symbol in the burst
- Q_k = Quadrature phase measurement of the kth symbol in the burst
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The block computes the X-percentile MER by creating a histogram of all the incoming MER_k values. The output provides the MER value above which X% of the MER values fall.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ACPR` | `comm.CCDF` | `comm.EVM`

Introduced in R2012a

reset

System object: `comm.MER`
Package: `comm`

Reset states of MER measurement object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MER` object, `H`.

step

System object: `comm.MER`
Package: `comm`

Measure modulation error ratio

Syntax

```
MERDB= step(MER,REFSYM,RXSYM)
MERDB = step(MER,RXSYM)
[ ___ ,MINMER] = step( ___ )
[ ___ ,XMER] = step( ___ )
[ ___ ,NUMSYM] = step( ___ )
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`MERDB= step(MER,REFSYM,RXSYM)` returns the measured MER, `MERDB`, of the received signal `RXSYM`, based on reference signal `REFSYM`. MER values are measured in dB.

`REFSYM`. `REFSYM` and `RXSYM` inputs are complex column vectors of equal dimensions and data type. The data type can be double, single, signed integer, or signed fixed point with power-of-two slope and zero bias. All outputs of the object are of data type `double`. To set the interval over which the MER is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`MERDB = step(MER,RXSYM)` returns the measured MER of received signal `RXSYM` based on a reference signal specified in the `ReceivedConstellation` property.

`[ ___ ,MINMER] = step( ___ )` returns the minimum MER, `MINMER`, given either of the two previous syntaxes.

To return minimum MER, set the `MinimumMEROutputPort` property to `true`. To set the interval over which `MINMER` is measured, use the `MeasurementIntervalSource` and `MeasurementInterval` properties.

`[ ___ ,XMER] = step( ___ )` returns the *X*-percentile MER, `XMER`.

To return the *X*-percentile MER, set the `XPercentileMEROutputPort` property to `true`. `XMER` is the MER above which *X*% of the measurements fall, where *X* is set by the `XPercentileValue` property. `XMER` is measured using all the input frames since the last reset.

`[ ___ ,NUMSYM] = step( ___ )` returns the number of symbols, `NUMSYM`, used to calculate the *X*-percentile MER.

To return `NUMSYM`, set the `SymbolCountOutputPort` to `true`. `NUMSYM` is measured using all the input frames since the last reset.

Note `MER` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MIMOChannel

Package: `comm`

Filter input signal through MIMO multipath fading channel

Description

A comm.MIMOChannel object filters an input signal through a multiple-input/multiple-output (MIMO) multipath fading channel. This object models both Rayleigh and Rician fading and employs the Kronecker model for modeling the spatial correlation between the links. For processing details, see the Algorithms on page 3-983 section.

To filter an input signal through a MIMO multipath fading channel:

1. Create the `comm.MIMOChannel` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
mimochan = comm.MIMOChannel
mimochan = comm.MIMOChannel(Name,Value)
```

Description

`mimochan = comm.MIMOChannel` creates a multiple-input multiple-output (MIMO) frequency-selective or frequency-flat fading channel System object. This object filters a real or complex input signal through the multipath MIMO channel to obtain the channel-impaired signal.

`mimochan = comm.MIMOChannel(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.MIMOChannel('SampleRate',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Input signal sample rate
`1` (default) | positive scalar

Input signal sample rate in hertz, specified as a positive scalar.

Data Types: `double`

`PathDelays` — Discrete path delay

`0` (default) | scalar | row vector

Discrete path delay in seconds, specified as a scalar or row vector.

- When you set `PathDelays` to a scalar, the MIMO channel is frequency flat.
- When you set `PathDelays` to a vector, the MIMO channel is frequency selective.

Data Types: `double`

`AveragePathGains` — Average path gains (dB)

`0` (default) | scalar | row vector

Average path gains in decibels, specified as a scalar or row vector. `AveragePathGains` must have the same size as PathDelays.

Data Types: `double`

`NormalizePathGains` — Normalize path gains

`true` (default) | `false`

Normalize path gains, specified as `true` or `false`.

- When you set this property to `true`, the fading processes are normalized so that the total power of the path gains, averaged over time, is `0` dB.
- When you set this property to `false`, there is no normalization on path gains.

The average powers of the path gains are specified by the AveragePathGains property.

Data Types: `logical`

`FadingDistribution` — Fading distribution

`'Rayleigh'` (default) | `'Rician'`

Fading distribution to use for the channel, specified as `'Rayleigh'` or `'Rician'`.

Data Types: `char`

`KFactor` — K-factor of Rician fading channel

`3` (default) | positive scalar | row vector

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of positive-valued elements. N_P equals number of path delays specified by the PathDelays property.

- If you set `KFactor` to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of `KFactor`. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set `KFactor` to a row vector, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process with a Rician K-factor specified by that element. The discrete path corresponding to a zero-valued element of the `KFactor` vector is a Rayleigh fading process.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`DirectPathDopplerShift` — Doppler shifts for line-of-sight components (Hz)

`0` (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This property must have the same size as KFactor.

- If you set `DirectPathDopplerShift` to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set `DirectPathDopplerShift` to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of `DirectPathDopplerShift` that correspond to positive elements in the KFactor vector.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`DirectPathInitialPhase` — Initial phases for line-of-sight components (Radians)

`0` (default) | scalar | row vector

Initial phases for the line-of-sight components of the Rician fading channel in radians, specified as a scalar or row vector. This property must have the same size as KFactor.

- If you set `DirectPathInitialPhase` to a scalar, it represents the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set `DirectPathInitialPhase` to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the elements of `DirectPathInitialPhase` that correspond to positive elements in the KFactor vector.

Dependencies

This property applies when FadingDistribution is `Rician`.

Data Types: `double`

`MaximumDopplerShift` — Maximum Doppler shift for all channel paths (Hz)

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

The Doppler shift applies to all channel paths. When you set this property to `0`, the channel remains static for the entire input. You can use the `reset` object function to generate a new channel realization.

`MaximumDopplerShift` must be smaller than (SampleRate/10)/f_c for each path, where f_c represents the cutoff frequency factor of the path. For more information on the cutoff frequency, see Cutoff Frequency Factor on page 3-984.

Data Types: `double`

`DopplerSpectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this property is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.
- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the PathDelays property.

The maximum Doppler shift value necessary to specify the Doppler spectrum/spectra is given by the MaximumDopplerShift property.

Dependencies

This property applies when MaximumDopplerShift is greater than zero.

If you assign the FadingTechnique property to `'Sum of sinusoids'`, you must set `DopplerSpectrum` to `doppler('Jakes')`.

`SpatialCorrelationSpecification` — Spatial correlation specification

`'Separate Tx Rx'` (default) | `'None'` | `'Combined'`

Spatial correlation specification, specified as `'Separate Tx Rx'`, `'None'`, or `'Combined'`.

- Choose `'Spatial Tx Rx'` to separately specify the transmit and receive spatial correlation matrices from which the number of transmit antenna (N_T) and number of receive antennas (N_R) are derived.
- Choose `'None'` to specify the number of transmit and receive antennas.
- Choose `'Combined'` to specify a single correlation matrix for the whole channel, from which the product of N_T and N_R is derived.

Data Types: `char`

`NumTransmitAntennas` — Number of transmit antennas

2 (default) | positive integer

Number of transmit antennas, specified as a positive integer.

Dependencies

This property applies when SpatialCorrelationSpecification is `'None'` or `'Combined'`.

Data Types: `double`

`NumReceiveAntennas` — Number of receive antennas

2 (default) | positive integer

Number of receive antennas, specified as a positive integer.

Dependencies

This property applies when SpatialCorrelationSpecification is `'None'` or `'Combined'`.

Data Types: `double`

`TransmitCorrelationMatrix` — Spatial correlation of transmitter

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the transmitter as an N_T-by-N_T matrix or N_T-by-N_T-by-N_P array. N_T is the number of transmit antennas, and N_P equals the value of the PathDelays property.

- If `PathDelays` is a scalar, the channel is frequency-flat, and `TransmitCorrelationMatrix` is an N_T-by-N_T Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If `PathDelays` is a vector, the channel is frequency selective, and you can specify `TransmitCorrelationMatrix` as a matrix. Each path has the same transmit spatial correlation matrix.
- Alternatively, you can specify `TransmitCorrelationMatrix` as an N_T-by-N_T-by-N_P array, where each path can have its own different transmit spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Separate Tx Rx'`.

Data Types: `double`
Complex Number Support: Yes

`ReceiveCorrelationMatrix` — Spatial correlation of receiver

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the receiver as an N_R-by-N_R matrix or N_R-by-N_R-by-N_P array. N_R is the number of receive antennas, and N_P equals the value of the PathDelays property.

- If PathDelays is a scalar, the channel is frequency flat, and `ReceiveCorrelationMatrix` is an N_R-by-N_R Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If PathDelays is a vector, the channel is frequency selective, and you can specify `ReceiveCorrelationMatrix` as a matrix. Each path has the same receive spatial correlation matrix.
- Alternatively, you can specify `ReceiveCorrelationMatrix` as an N_R-by-N_R-by-N_P array, where each path can have its own different receive spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Separate Tx Rx'`.

Data Types: `double`
Complex Number Support: Yes

`SpatialCorrelationMatrix` — Combined spatial correlation matrix

`[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]` (default) | matrix | 3-D array

Combined spatial correlation matrix, specified as an N_{TR}-by-N_{TR} matrix or N_{TR}-by-N_{TR}-by-N_P array, where $N_{TR} = (N_T \times N_R)$, and N_P equals the value of the PathDelays property.

- If PathDelays is a scalar, the channel is frequency flat, and `SpatialCorrelationMatrix` is an N_{TR}-by-N_{TR} Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If PathDelays is a vector, the channel is frequency selective, and you can specify `SpatialCorrelationMatrix` as a matrix. Each path has the same spatial correlation matrix.
- Alternatively, you can specify `SpatialCorrelationMatrix` as an N_{TR}-by-N_{TR}-by-N_P array, where each path can have its own different combined spatial correlation matrix.

Dependencies

This property applies when you set the SpatialCorrelationSpecification property to `'Combined'`.

Data Types: `double`

`AntennaSelection` — Antenna selection scheme

`'Off'` (default) | `'Tx'` | `'Rx'` | `'Tx and Rx'`

Antenna selection scheme, specified as `'Off'`, `'Tx'`, `'Rx'`, or `'Tx and Rx'`.

`Tx` represents transmit antennas and `Rx` represents receive antennas. When you configure any antenna selection other than the default setting, the object requires one or more inputs to specify which antennas are selected for signal transmission. For more information, see Antenna Selection on page 3-984.

Data Types: `char`

`NormalizeChannelOutputs` — Normalize channel outputs

`true` (default) | `false`

Normalize channel outputs, specified as `true` or `false`.

- When you set this property to `true`, channel outputs are normalized by the number of receive antennas.
- When you set this property to `false`, channel outputs are not normalized.

Data Types: `logical`

FadingTechnique — Channel model fading technique

`'Filtered Gaussian noise'` (default) | `'Sum of sinusoids'`

Channel model fading technique, specified as `'Filtered Gaussian noise'` or `'Sum of sinusoids'`.

Data Types: `char`

`NumSinusoids` — Number of sinusoids used

48 (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

This property applies when FadingTechnique is `'Sum of sinusoids'`.

Data Types: `double`

`InitialTimeSource` — Source to control start time of fading process

`'Property'` (default) | `'Input port'`

Source to control the start time of the fading process, specified as `'Property'` or `'Input port'`.

- `'Property'` -- Use the InitialTime property to set the initial time offset.
- `'Input port'` -- Specify the start time of the fading process by using the `initialtime` input to the object. The input value can change in consecutive calls to the object.

Dependencies

This property applies when FadingTechnique is `'Sum of sinusoids'`.

`InitialTime` — Initial time offset

`0` (default) | nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

When `InitialTime` is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Dependencies

This property applies when the FadingTechnique property is set to `'Sum of sinusoids'` and the InitialTimeSource property is set to `'Property'`.

Data Types: `double`

`RandomStream` — Source of random number stream

`'Global stream'` (default) | `'mt19937ar with seed'`

Source of the random number stream, specified as `'Global stream'` or `'mt19937ar with seed'`.

- `'Global stream'` -- The current global random number stream is used for normally distributed random number generation. In this case, the `reset` object function resets the filters only.
- `'mt19937ar with seed'` -- The mt19937ar algorithm is used for normally distributed random number generation. In this case, the `reset` object function resets the filters and also reinitializes the random number stream to the value of the Seed property.

Data Types: `char`

Seed — Initial seed of mt19937ar random number stream

`73` (default) | nonnegative integer

Initial seed of the mt19937ar random number stream, specified as a nonnegative integer. When the `reset` object function is called, the mt19937ar random number stream is reinitialized to the `Seed` value.

Dependencies

This property applies when you set the RandomStream property to `'mt19937ar with seed'`.

Data Types: `double`

`PathGainsOutputPort` — Option to output path gains

`false` (default) | `true`

Option to output path gains, specified as `false` or `true`. Set this property to `true` to output the channel path gains of the underlying fading process.

Data Types: `logical`

`Visualization` — Channel visualization

`'Off'` (default) | `'Impulse response'` | `'Frequency response'` | `'Impulse and frequency responses'` | `'Doppler spectrum'`

Channel visualization preference, specified as `'Off'`, `'Impulse response'`, `'Frequency response'`, `'Impulse and frequency responses'`, or `'Doppler spectrum'`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

Dependencies

Visualization applies only when the FadingTechnique property is set to `'Filtered Gaussian noise'`.

`AntennaPairsToDisplay` — Transmit-receive antenna pair to display

`[1 1]` (default) | row vector

Transmit-receive antenna pair to display, specified as a 1-by-2 vector, where the first element corresponds to the desired transmit antenna and the second element corresponds to the desired receive antenna. At this time, only a single pair can be displayed.

Dependencies

This property applies when Visualization is not `Off`.

`PathsForDopplerDisplay` — Path for which the Doppler spectrum is displayed

`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_P, where N_P equals the value of the PathDelays property.

Dependencies

This property applies when Visualization is set to `'Doppler spectrum'`.

`SamplesToDisplay` — Percentage of samples to display

`25%` (default) | `10%` | `50%` | `100%`

Percentage of samples to display, specified as `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This property applies when Visualization is `'Impulse response'`, `'Frequency response'`, or `'Impulse and frequency responses'`.

Usage

Syntax

```
outsignal = mimochan(insignal)
outsignal = mimochan(insignal,seltx)
outsignal = mimochan(insignal,selrx)
outsignal = mimochan(insignal,seltx,selrx)
outsignal = mimochan( ___ ,initialtime)
[outsignal,pathgains] = mimochan( ___ )
```

Description

`outsignal = mimochan(insignal)` filters the input signal through the MIMO fading channel specified by `mimochan` and returns the result in `outsignal`.

`outsignal = mimochan(insignal,seltx)` turns on the transmit antennas selected by `seltx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Tx'`.

For example, to select the first and third transmit antenna index as active:

```
mimochan = comm.MIMOChannel('AntennaSelection','Tx');
seltx = [1 0 1];
...
outsignal = mimochan(insignal,seltx);
```

`outsignal = mimochan(insignal,selrx)` turns on receive antennas, selected by `selrx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Rx'`.

For example, to select the second receive antenna index as active:

```
mimochan = comm.MIMOChannel('AntennaSelection','Rx');
selrx = [0 1];
...
outsignal = mimochan(insignal,selrx);
```

`outsignal = mimochan(insignal,seltx,selrx)` turns on transmit and receive antennas, selected by `seltx` and `selrx` for channel processing.

This syntax applies when you set the AntennaSelection property of the object to `'Tx and Rx'`.

For example:

```
mimochan = comm.MIMOChannel('AntennaSelection','Tx and Rx');
seltx = [1 1];
selrx = [0 1];
...
outsignal = mimochan(insignal,selrx);
```

`outsignal = mimochan( ___ ,initialtime)` specifies a start time for the fading process.

This syntax applies when you set the FadingTechnique property of the object to `'Sum of sinusoids'` and the InitialTimeSource property of the object to `'Input port'`. The syntax supports input options from prior syntaxes.

`[outsignal,pathgains] = mimochan( ___ )` also returns the MIMO channel path gains for antenna selection schemes. The syntax supports input options from prior syntaxes.

Input Arguments

`insignal` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, an N_S element column vector, an N_S-by-N_T matrix, or an N_S-by-N_{ST} matrix.

- N_S is the number of samples.
- N_T is the number of transmit antennas. N_T is determined by the TransmitCorrelationMatrix or NumTransmitAntennas property values of the object.
- N_{ST} is the number of selected transmit antennas, as determined by the number of elements set to `1` in the vector provided to the seltx input.

The number of transmit antennas is determined by the TransmitCorrelationMatrix or NumTransmitAntennas property values of the object.

Data Types: `double` | `single`
Complex Number Support: Yes

`seltx` — Select active transmit antennas
binary vector

Select active transmit antennas, specified as a 1-by-N_T binary vector. N_T represents the number of transmit antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Data Types: `double`

`selrx` — Select active receive antennas
binary vector

Select active receive antennas, specified as a 1-by-N_R binary vector. N_R represents the number of receive antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Data Types: `double`

`initialtime` — Initial time offset
`0` (default) | nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

The initial time offset must be greater than the last frame end time. When `initialtime` is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Data Types: `double`

Output Arguments

`outsignal` — Output signal
matrix

Output data signal, returned as an N_S-by-N_R or N_S-by-N_{SR} matrix.

- N_S is the number of samples.

- N_R is the number of receive antennas. N_R is determined by the ReceiveCorrelationMatrix or NumReceiveAntennas property values of the object.
- N_{SR} is the number of selected receive antennas, as determined by the number of elements set to `1` in the vector provided to the selrx input.

`pathgains` — Output path gains
4-D array

Output path gains, returned as an N_S-by-N_P-by-N_T-by-N_R array with `NaN` values for the unselected transmit-receive antenna pairs.

- N_S is the number of samples.
- N_P equals the value of the PathDelays property.
- N_T is the number of transmit antennas.
- N_R is the number of receive antennas.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to comm.MIMOChannel

info	Characteristic information about fading channel object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Note

- If you set the `RandomStream` property of the object to `'Global stream'`, the `reset` object function resets the filters only.
- If you set `RandomStream` to `'mt19937ar with seed'`, the `reset` object function resets the filters and also reinitializes the random number stream to the value of the `Seed` property.

Examples

Pass QPSK Data Through 4-by-2 MIMO Channel

Create a 4-by-2 MIMO channel by using the MIMO channel System object. Pass modulated and spatially encoded data through the channel.

Generate QPSK-modulated data.

```
data = randi([0 3],1000,1);
modData = pskmod(data,4,pi/4);
```

Create an orthogonal space-time block encoder to encode the modulated data into four spatially separated streams. Encode the data.

```
ostbc = comm.OSTBCEncoder('NumTransmitAntennas',4,'SymbolRate',1/2);
txSig = ostbc(modData);
```

Create a MIMO channel object, using name-value pairs to set the properties. The channel consists of two paths with a maximum Doppler shift of 5 Hz. Set the `SpatialCorrelationSpecification` property to `'None'`, which requires that you specify the number of transmit and receive antennas. Set the number of transmit antennas to 4 and the number of receive antennas to 2.

```
mimochannel = comm.MIMOChannel(...
    'SampleRate',1000, ...
    'PathDelays',[0 2e-3], ...
    'AveragePathGains',[0 -5], ...
    'MaximumDopplerShift',5, ...
    'SpatialCorrelationSpecification','None', ...
    'NumTransmitAntennas',4, ...
    'NumReceiveAntennas',2);
```

Pass the modulated and encoded data through the MIMO channel.

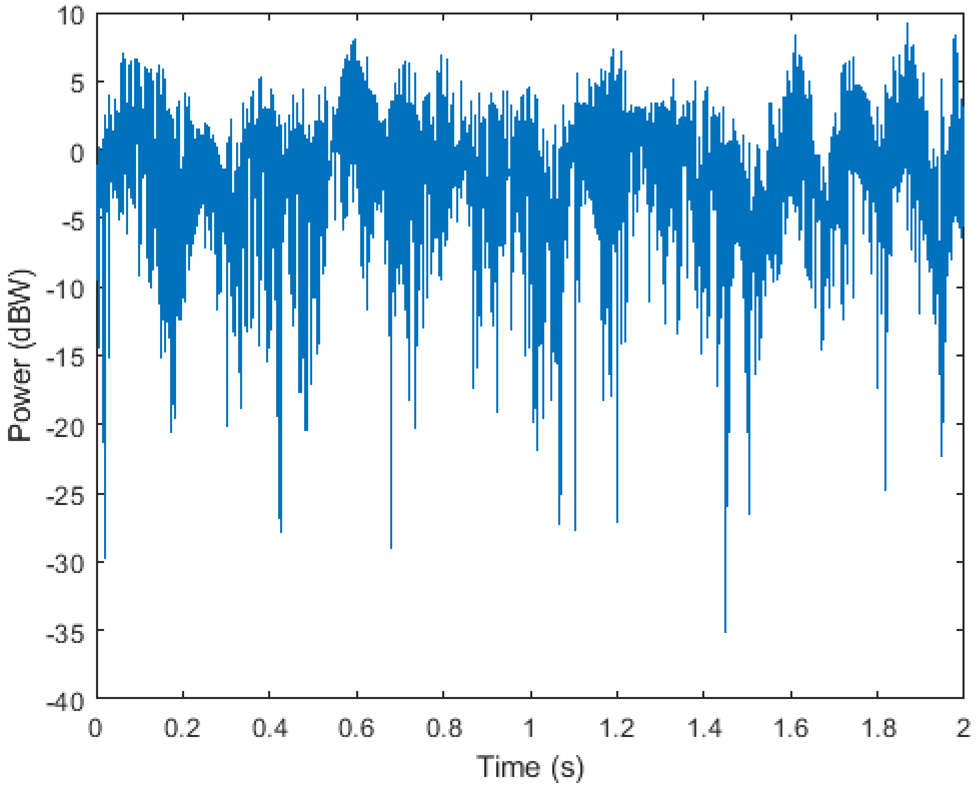
```
rxSig = mimochannel(txSig);
```

Create a time vector, `t`, to use for plotting the power of the received signal.

```
ts = 1/mimochannel.SampleRate;
t = (0:ts:(size(txSig,1)-1)*ts)';
```

Calculate and plot the power of the signal received by antenna 1.

```
pwrdB = 20*log10(abs(rxSig(:,1)));

plot(t,pwrdB)
xlabel('Time (s)')
ylabel('Power (dBW)')
```



Examine Spatial Correlation Characteristics of 2-by-2 Rayleigh Fading Channel

Without specifying antenna selection, filter PSK-modulated data through a 2-by-2 Rayleigh fading channel and examine the spatial correlation characteristics of the channel realization. Use the `release` object function to unlock the object to set the `AntennaSelection` property to `'Tx and Rx'` and then confirm the unselected transmit-receive antenna pairs.

Examine Spatial Correlation Characteristics Without Specifying Antenna Selection

Create a PSK modulator System object™ to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
modData = pskModulator(randi([0 pskModulator.ModulationOrder-1],1e5,1));
```

Split the modulated data into two spatial streams.

```
channelInput = reshape(modData,[2 5e4]).';
```

Create a 2-by-2 MIMO channel System object with two discrete paths. Each path has different transmit and receive correlation matrices, specified by the `TransmitCorrelationMatrix` and `ReceiveCorrelationMatrix` properties.

```
mimoChan = comm.MIMOChannel('SampleRate',1000, 'PathDelays',[0 1e-3], ...
    'AveragePathGains',[3 5], 'NormalizePathGains',false, 'MaximumDopplerShift',5, ...
    'TransmitCorrelationMatrix',cat(3,eye(2),[1 0.1;0.1 1]), ...
    'ReceiveCorrelationMatrix',cat(3,[1 0.2;0.2 1],eye(2)), ...
    'RandomStream','mt19937ar with seed', 'Seed',33, 'PathGainsOutputPort',true);
```

Filter the modulated data using the MIMO channel object.

```
[~,pathGains] = mimoChan(channelInput);
```

The transmit spatial correlation for the first discrete path at the first receive antenna is specified as an identity matrix in the `TransmitCorrelationMatrix` property. Confirm that the channel output `pathGains` exhibits the same statistical characteristics by using the `corrcoef` function.

```
disp('Tx spatial correlation, first path, first Rx:');
```

```
Tx spatial correlation, first path, first Rx:
```

```
disp(corrcoef(squeeze(pathGains(:,1,:,1))));
```

```
   1.0000 + 0.0000i   0.0357 - 0.0253i
   0.0357 + 0.0253i   1.0000 + 0.0000i
```

The transmit spatial correlation for the second discrete path at the second receive antenna is specified as `[1 0.1;0.1 1]` in the `TransmitCorrelationMatrix` property. Confirm that the channel output `pathGains` exhibits the same statistical characteristics.

```
disp('Tx spatial correlation, second path, second Rx:');
```

```
Tx spatial correlation, second path, second Rx:
```

```
disp(corrcoef(squeeze(pathGains(:,2,:,2))));
```

```
   1.0000 + 0.0000i   0.0863 + 0.0009i
   0.0863 - 0.0009i   1.0000 + 0.0000i
```

The receive spatial correlation for the first discrete path at the second transmit antenna is specified as `[1 0.2;0.2 1]` in the `ReceiveCorrelationMatrix` property. Confirm that the channel output `pathGains` exhibits the same statistical characteristics.

```
disp('Rx spatial correlation, first path, second Tx:');
```

```
Rx spatial correlation, first path, second Tx:
```

```
disp(corrcoef(squeeze(pathGains(:,1,2,:))));
```

```
   1.0000 + 0.0000i   0.2236 + 0.0550i
   0.2236 - 0.0550i   1.0000 + 0.0000i
```

The receive spatial correlation for the second discrete path at the first transmit antenna is specified as an identity matrix in the `ReceiveCorrelationMatrix` property. Confirm that the channel output `pathGains` exhibits the same statistical characteristics.

```
disp('Rx spatial correlation, second path, first Tx:');
```

```
Rx spatial correlation, second path, first Tx:
```

```
disp(corrcoef(squeeze(pathGains(:,2,1,:))));
```

```
   1.0000 + 0.0000i  -0.0088 - 0.0489i
  -0.0088 + 0.0489i   1.0000 + 0.0000i
```

Examine Spatial Correlation Characteristics Specifying Antenna Selection

Enable transmit and receive antenna selection for the mimoChan object. The input frame size is shortened to 100.

```
release(mimoChan);
mimoChan.AntennaSelection = 'Tx and Rx';
modData = pskModulator(randi([0 pskModulator.ModulationOrder-1],100,1));
```

Select the first transmit antenna and second receive antenna.

```
[channelOutput,pathGains] = mimoChan(modData,[1 0],[0 1]);
```

Confirm that the path gains that MATLAB® returns have NaN values for the unselected transmit-receive antenna pairs.

```
disp('Return 1 if the path gains for the second transmit antenna are NaN:');
```

```
Return 1 if the path gains for the second transmit antenna are NaN:
```

```
disp(isequal(isnan(squeeze(pathGains(:,:,2,:))), ones(100,2,2)));
```

```
   1
```

```
disp('Return 1 if the path gains for the first receive antenna are NaN:');
```

```
Return 1 if the path gains for the first receive antenna are NaN:
```

```
disp(isequal(isnan(squeeze(pathGains(:,:,:,1))), ones(100,2,2)));
```

```
   1
```

Display Impulse and Frequency Responses of Frequency Selective Channel

Create a frequency selective MIMO channel and display its impulse and frequency responses.

Set the sample rate to 10 MHz and specify path delays and gains using the extended vehicular A (EVA) channel parameters. Set the maximum Doppler shift to 70 Hz.

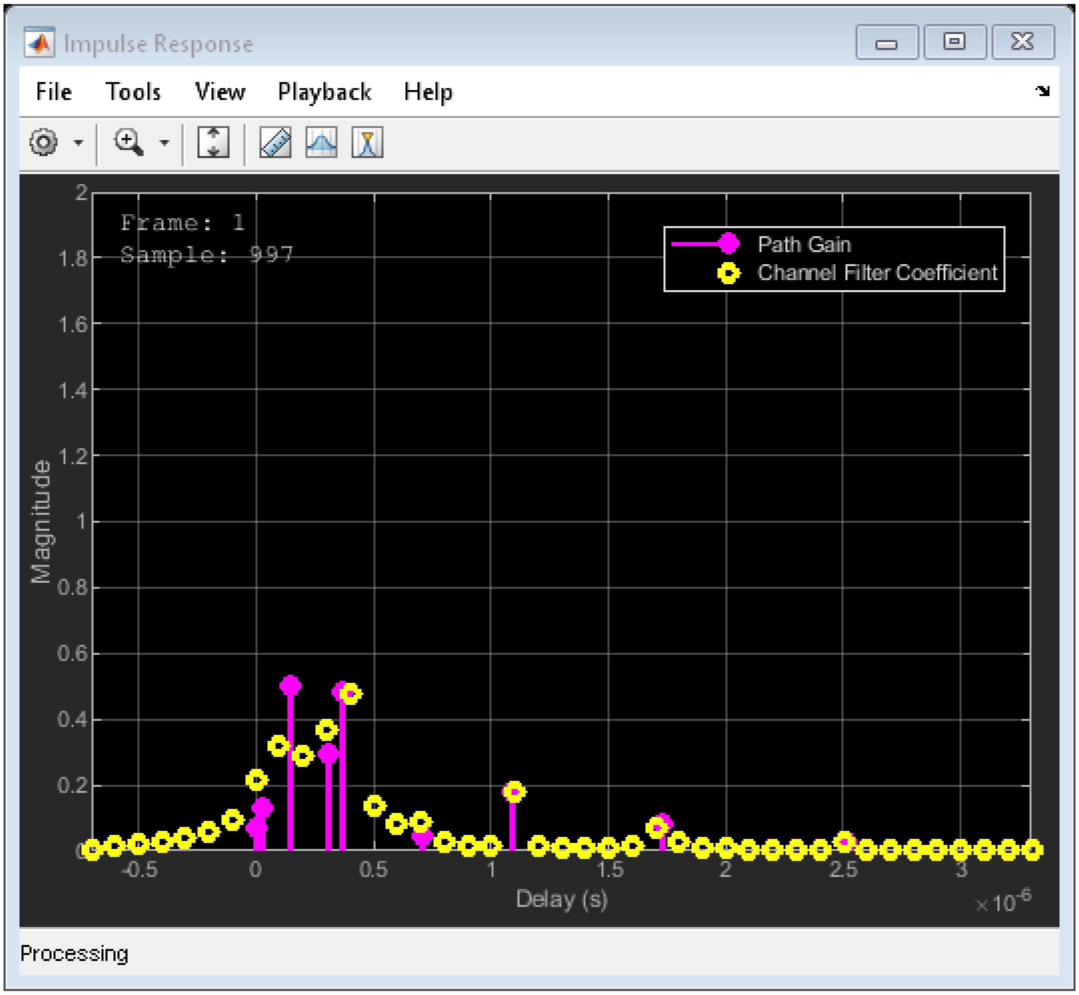
```
fs = 10e6;                                                  % Hz
pathDelays = [0 30 150 310 370 710 1090 1730 2510]*1e-9;    % sec
avgPathGains = [0 -1.5 -1.4 -3.6 -0.6 -9.1 -7 -12 -16.9];   % dB
fD = 70;                                                    % Hz
```

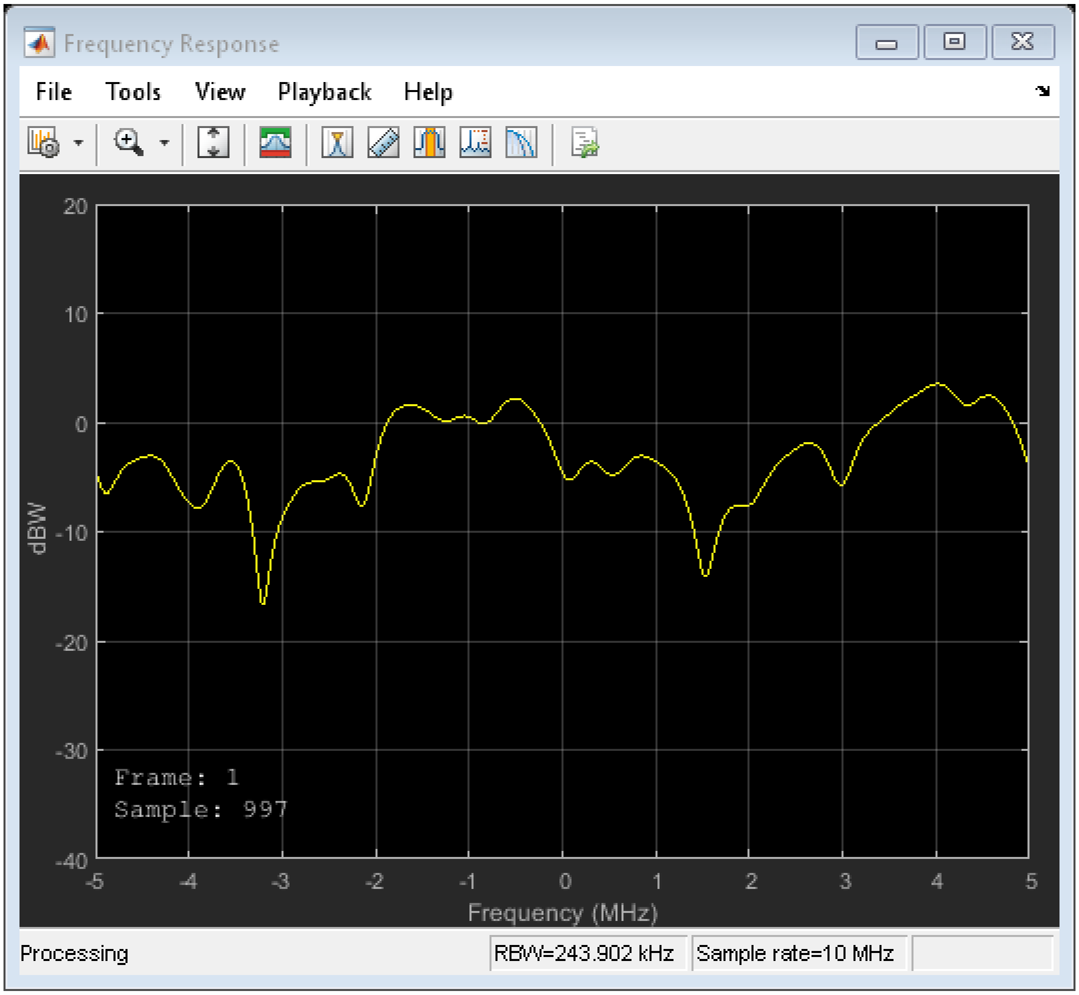
Create a 2x2 MIMO channel System object with the previously defined parameters and set the Visualization property to Impulse and frequency responses using name-value pairs. By default, the antenna pair corresponding to transmit antenna 1 and receive antenna 1 will be displayed.

```
mimoChan = comm.MIMOChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the MIMO channel. The impulse response plot allows you to easily identify the individual paths and their corresponding filter coefficients. The frequency selective nature of the EVA channel is shown by the frequency response plot.

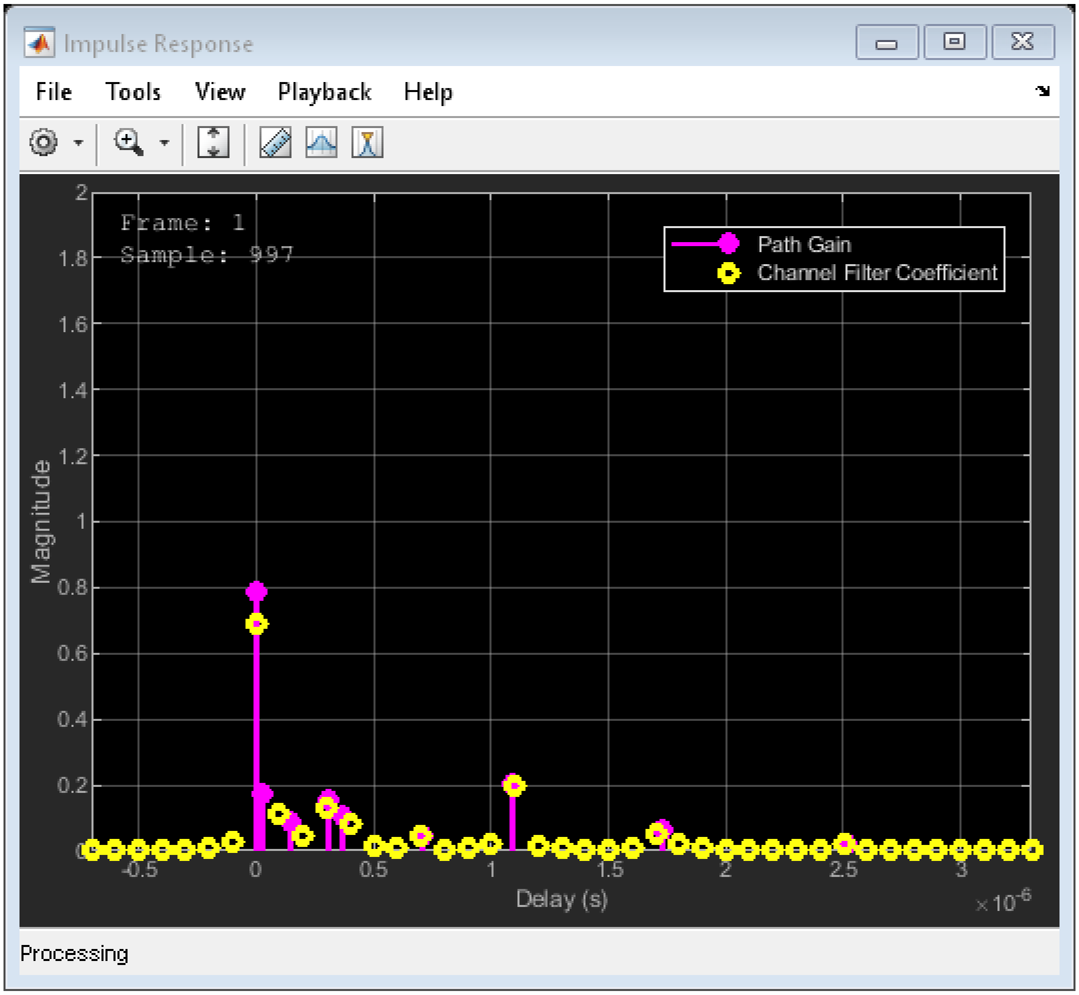
```
x = randi([0 1],1000,2);
y = mimoChan(x);
```



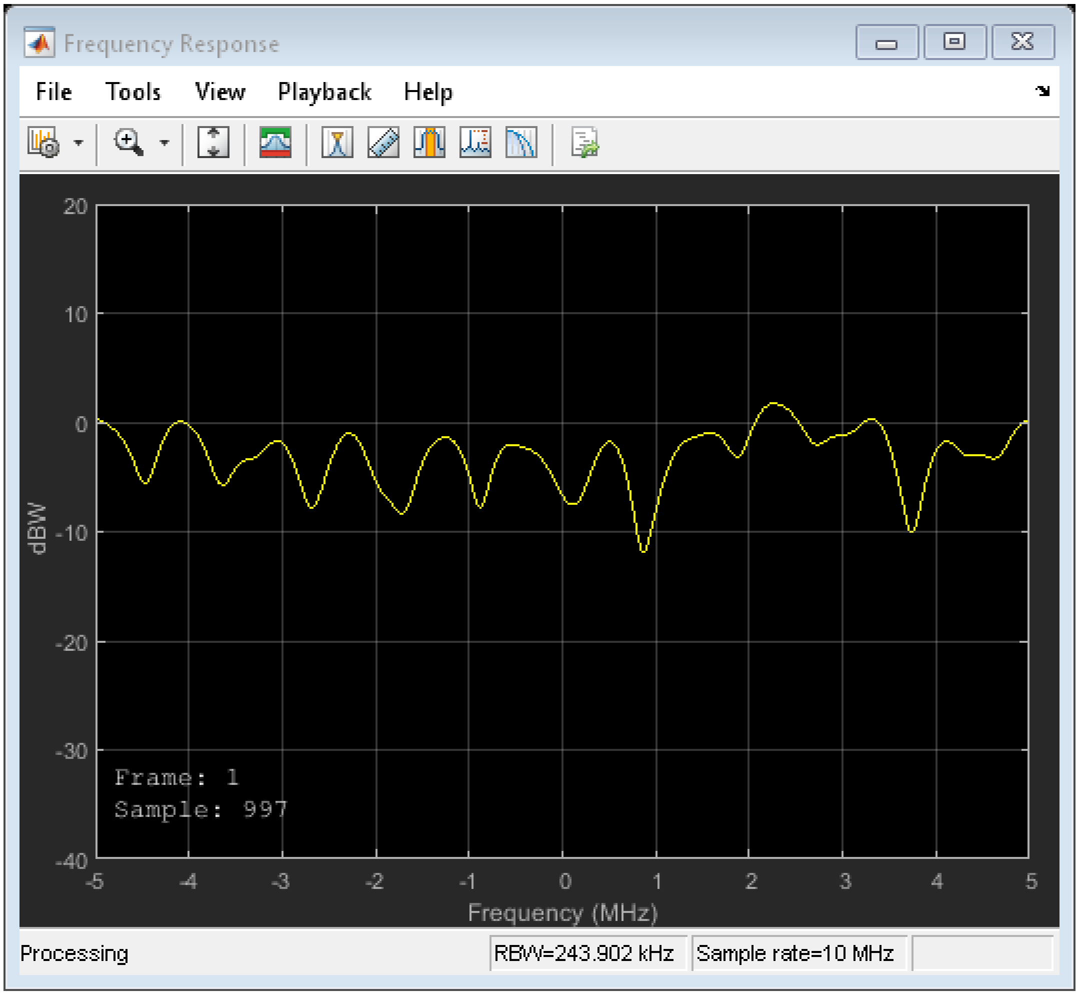


Release `mimoChan` and set the `AntennaPairsToDisplay` property to [2 1] to view the antenna pair corresponding to transmit antenna 2 and receive antenna 1. It is necessary to release the object as the property is non-tunable.

```
release(mimoChan)
mimoChan.AntennaPairsToDisplay = [2 1];
y = mimoChan(x);
```



Impulse Response
File
Tools
View
Playback
Help
Frame: 1
Sample: 997
Path Gain
Channel Filter Coefficient
Magnitude
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.5
0
0.5
1
1.5
2
2.5
3
Delay (s)
$\times 10^{-6}$
Processing



Display Doppler for 2x2 MIMO Channel

Create and visualize the Doppler spectra of a MIMO channel having two paths.

Construct a cell array of Doppler structures to be used in creating the channel. The Doppler spectrum of the first path is set to have a bell shape while the second path is set to be flat.

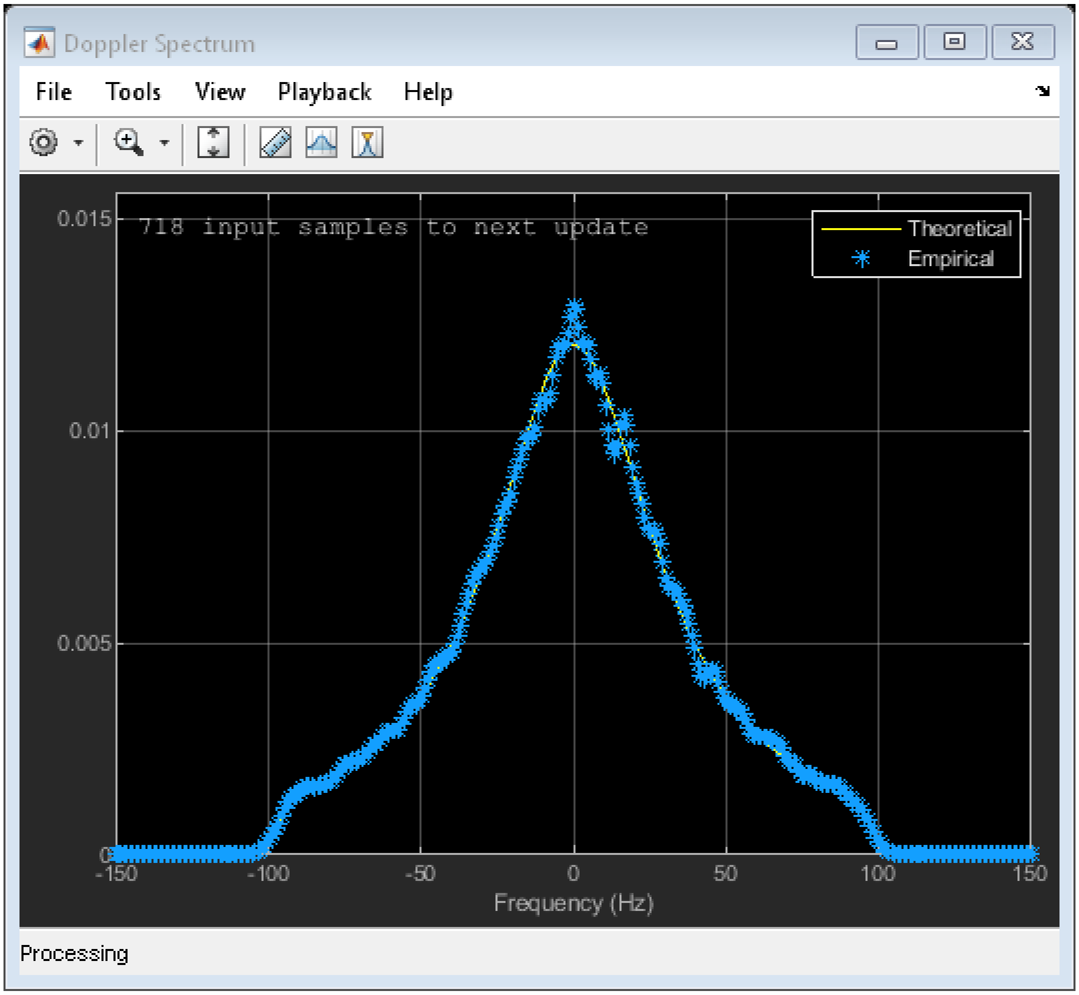
```
dp{1} = doppler('Bell');
dp{2} = doppler('Flat');
```

Create a default 2x2 MIMO channel with two paths and a 100 Hz maximum Doppler shift using name-value pairs. Set the `Visualization` property to `Doppler spectrum` and set `PathsForDopplerDisplay` to 1. The Doppler spectrum of the first path will be displayed.

```
mimoChan = comm.MIMOChannel('SampleRate',1000, ...
    'PathDelays',[0 0.002], ...
    'AveragePathGains',[0 -3], ...
    'MaximumDopplerShift',100, ...
    'DopplerSpectrum',dp, ...
    'Visualization','Doppler spectrum', ...
    'PathsForDopplerDisplay',1);
```

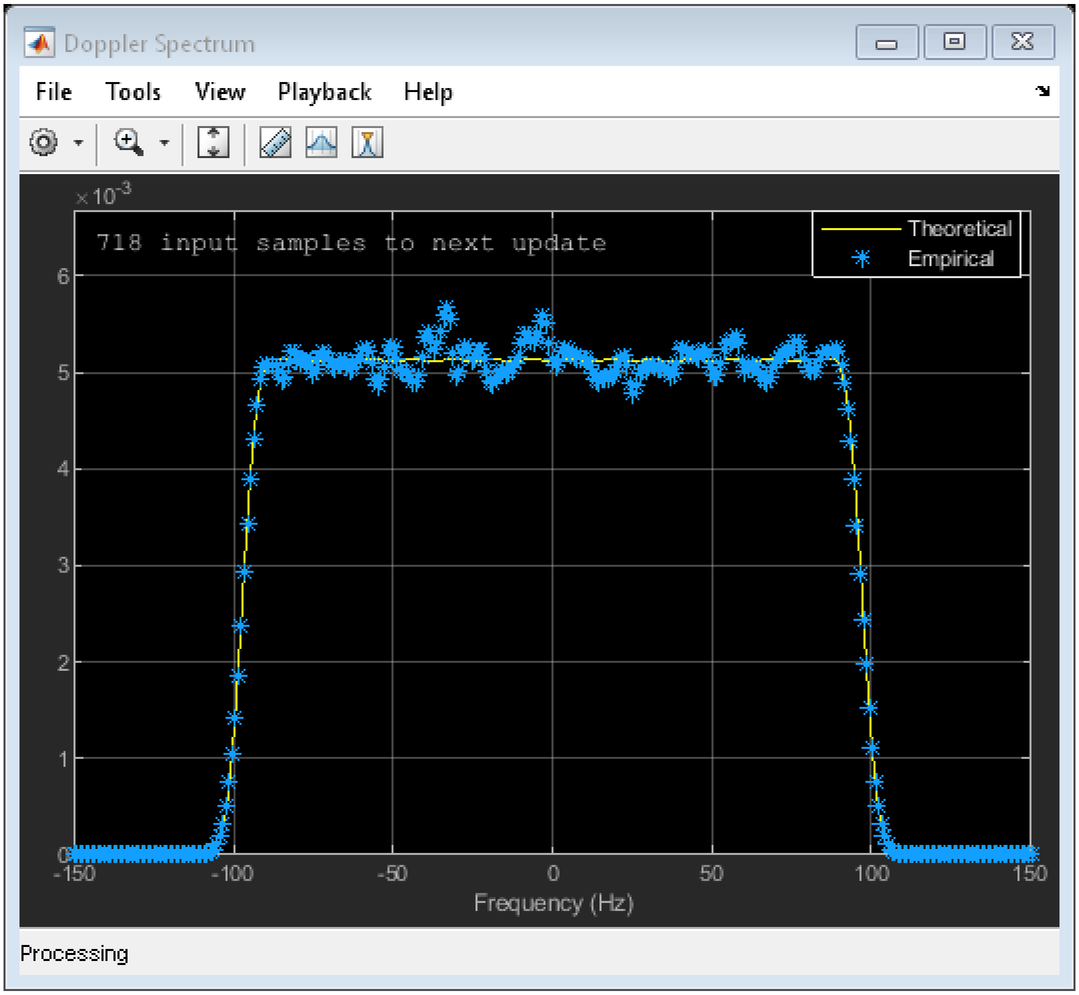
Pass random data through the MIMO channel to generate the Doppler spectrum of the first path. Since the Doppler spectrum plot only updates when its buffer is filled, the `mimoChan` function is invoked multiple times to improve the accuracy of the estimate. Observe that the spectrum has a bell shape and that its minimum and maximum frequencies fall within the limits set by `MaximumDopplerShift`.

```
for k = 1:25
    x = randi([0 1],10000,2);
    y = mimoChan(x);
end
```



Release `mimoChan` and set the `PathsForDopplerDisplay` property to 2. It is necessary to release the object as the property is non-tunable. Call the function multiple times to display the Doppler spectrum of the second path. Observe that the spectrum is flat.

```
release(mimoChan)
mimoChan.PathsForDopplerDisplay = 2;
for k = 1:25
    x = randi([0 1],10000,2);
    y = mimoChan(x);
end
```



Model MIMO Channel Using Sum-of-Sinusoids Technique

Create a MIMO channel object and pass data through it using the sum-of-sinusoids technique. The example demonstrates how the channel state is maintained in cases in which data is discontinuously transmitted.

Define the overall simulation time and three time segments for which data will be transmitted. In this case, the channel is simulated for 1 s with a 1000 Hz sampling rate. One 1000-sample, continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at time 0.1 s, 0.4 s, and 0.7 s.

```
t0 = 0:0.001:0.999;   % Transmission 0
t1 = 0.1:0.001:0.199; % Transmission 1
t2 = 0.4:0.001:0.499; % Transmission 2
t3 = 0.7:0.001:0.799; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],1000,2);  % 1000 samples
d1 = randi([0 1],100,2);   % 100 samples
```

```
d2 = randi([0 1],100,2);   % 100 samples
d3 = randi([0 1],100,2);   % 100 samples
```

Create a flat fading 2x2 MIMO channel System object with the `Sum of sinusoids` fading technique. So that results can be repeated, specify a seed using a name-value pair. As the `InitialTime` property is not specified, the fading channel will be simulated from time 0. Enable the path gains output port.

```
mimoChan1 = comm.MIMOChannel('SampleRate',1000, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the MIMO channel System object. Set the `InitialTimeSource` property to `Input port` so that the fading channel offset time can be specified as an input argument to the `mimoChan` function.

```
mimoChan2 = clone(mimoChan1);
mimoChan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first channel object, `mimoChan1`. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = mimoChan1(d0);
```

Pass random data through the second channel object, `mimoChan2`, where the initial time offsets are provided as input arguments.

```
[~,pg1] = mimoChan2(d1,0.1);
[~,pg2] = mimoChan2(d2,0.4);
[~,pg3] = mimoChan2(d3,0.7);
```

Compare the number of samples processed by the two channels using the `info` method. You can see that 1000 samples were processed by `mimoChan1` while only 300 were processed by `mimoChan2`.

```
G = info(mimoChan1);
H = info(mimoChan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

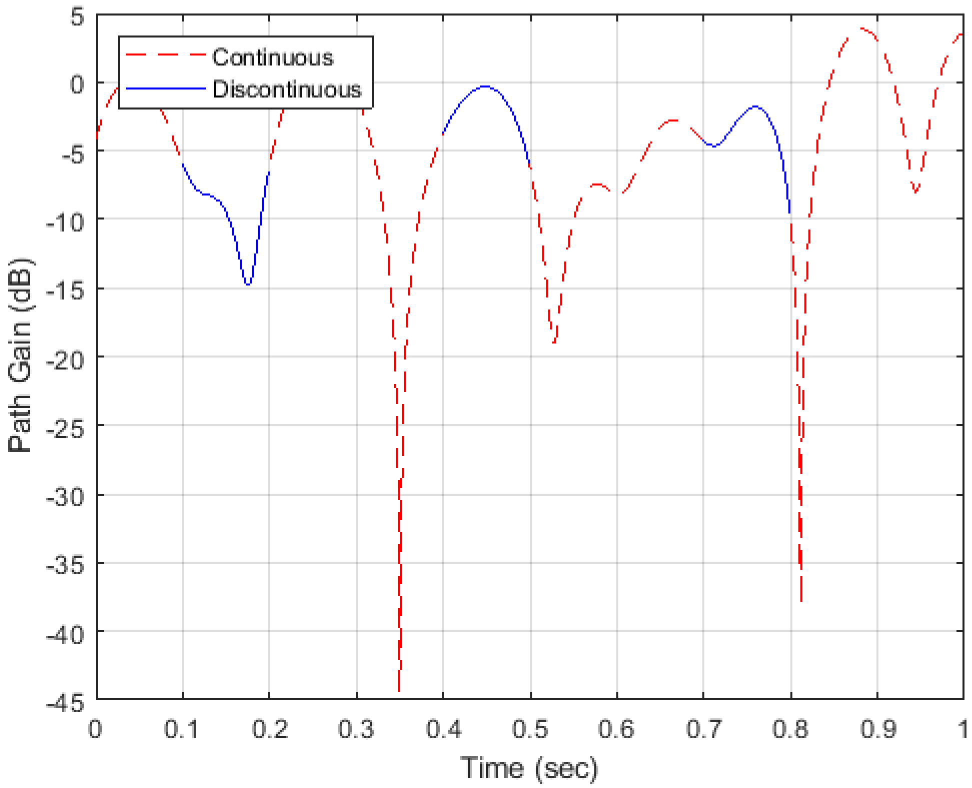
```
ans = 1×2

        1000         300
```

Convert the path gains into decibels for the path corresponding to the first transmit and first receive antenna.

```
pathGain0 = 20*log10(abs(pg0(:,1,1,1)));
pathGain1 = 20*log10(abs(pg1(:,1,1,1)));
pathGain2 = 20*log10(abs(pg2(:,1,1,1)));
pathGain3 = 20*log10(abs(pg3(:,1,1,1)));
```

Plot the path gains for the continuous and discontinuous cases. Observe that the gains for the three segments perfectly match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is ideally suited to the simulation of packetized data as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
```



Calculate Execution Time Advantage Using Sum of Sinusoids

Demonstrate the advantage of using the sum of sinusoids fading technique when simulating a channel with burst data.

Set the simulation parameters such that the sampling rate is 100 kHz, the total simulation time is 100 seconds, and the duty cycle for the burst data is 25%.

```
fs = 1e5;            % Hz
tsim = 100;          % seconds
dutyCycle = 0.25;
```

Create a flat fading 2x2 MIMO channel object using the default `Filtered Gaussian noise` technique.

```
fgn = comm.MIMOChannel('SampleRate',fs);
```

Create a similar MIMO channel object using the `Sum of sinusoids` technique where the fading process start times are given as an input argument.

```
sos = comm.MIMOChannel('SampleRate',fs, ...
    'FadingTechnique','Sum of sinusoids', ...
    'NumSinusoids',48, ...
    'InitialTimeSource','Input port');
```

Run a continuous sequence of random bits through the filtered Gaussian noise MIMO channel object. Use the tic/toc stopwatch timer functions to measure the execution time of the function call.

```
tic
y = fgn(randi([0 1],fs*tsim,2));
tFGN = toc;
```

To transmit a data burst each second, pass random bits through the sum of sinusoids MIMO channel object by calling the `sos` function inside of a for loop. Use the tic/toc stopwatch timer to measure the execution time.

```
tic
for k = 1:tsim
    z = sos(randi([0 1],fs*dutyCycle,2),0.5+(k-1));
end
tSOS = toc;
```

Compare the ratio of the sum of sinusoids execution time to the filtered Gaussian noise execution time. The ratio is less than one, which indicates that the sum of sinusoids technique is faster.

```
tSOS/tFGN
```

```
ans = 0.2713
```

Algorithms

The fading processing per link is described in Methodology for Simulating Multipath Fading Channels and assumes the same parameters for all ($N_T \times N_R$) links of the MIMO channel. Each link comprises all multipaths for that link.

The Kronecker Model

The Kronecker model assumes that the spatial correlations at the transmit and receive sides are separable. Equivalently, the direction of departure (DoD) and directions of arrival (DoA) spectra are assumed to be separable. The full correlation matrix is:

$$R_H = E[R_t \otimes R_r]$$

- The ⊗ symbol represents the Kronecker product.
- R_t represents the correlation matrix at the transmit side: $R_t = E\left[H^H H\right]$, of size N_T-by-N_T.
- R_r represents the correlation matrix at the receive side: $R_r = E\left[H H^H\right]$, of size N_R-by-N_R.

You can obtain a realization of the MIMO channel matrix as:

$$H = R_r^{\frac{1}{2}} A R_t^{\frac{1}{2}}$$

A is an N_R-by-N_T matrix of independent identically distributed complex Gaussian variables with zero mean and unit variance.

Cutoff Frequency Factor

The cutoff frequency factor, f_c, is determined for different Doppler spectrum types.

- For any Doppler spectrum type other than Gaussian and biGaussian, f_c equals 1.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation` × $\sqrt{2\log 2}$.
- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)` × $\sqrt{2\log 2}$.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - In all other cases, f_c equals 1.

Antenna Selection

When the object is in antenna selection mode, it uses the following algorithms to process an input signal:

- All random path gains are always generated and keep evolving for each link, whether or not a given link is selected. The path gain values output for the non-selected links are populated with `NaN`.
- The spatial correlation only applies to the selected transmit and/or receive antennas, and the correlation coefficients are the corresponding entries in the transmit, receive, or combined correlation matrices. In other words, the spatial correlation matrix for the selected transmit or receive antennas is a submatrix of the transmit, receive, or combined spatial correlation matrix property value.
- For signal paths associated with nonactive antennas, a signal with zero power is transmitted to the channel filter.
- Channel output normalization happens over the number of selected receive antennas.

References

[1] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*, Academic Press, 2007.

[2] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*, Academic Press, 2006.

[3] Kermoal, J. P., L. Schumacher, K. I. Pedersen, P. E. Mogensen, and F. Frederiksen. "A stochastic MIMO radio channel model with experimental validation." *IEEE Journal on Selected Areas of Communications*. Vol. 20, Number 6, 2002, pp. 1211–1226.

[4] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*, Second Edition, New York: Kluwer Academic/Plenum, 2000.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

Objects

`comm.AWGNChannel` | `comm.ChannelFilter` | `comm.RayleighChannel` | `comm.RicianChannel`

Blocks

MIMO Channel

Topics

Channel Visualization

Introduced in R2012a

comm.MLSEEqualizer

Package: `comm`

Equalize using maximum likelihood sequence estimation

Description

The `MLSEEqualizer` object uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The object processes input frames and outputs the maximum likelihood sequence estimate (MLSE) of the signal. This processing uses an estimate of the channel modeled as a finite impulse response (FIR) filter.

To equalize a linearly modulated signal and output the maximum likelihood sequence estimate:

1. Define and set up your maximum likelihood sequence estimate equalizer object. See "Construction" on page 3-986.
2. Call `step` to equalize a linearly modulated signal and output the maximum likelihood sequence estimate according to the properties of `comm.MLSEEqualizer`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MLSEEqualizer` creates a maximum likelihood sequence estimation equalizer (MLSEE) System object, `H`. This object uses the Viterbi algorithm and a channel estimate to equalize a linearly modulated signal that has been transmitted through a dispersive channel.

`H = comm.MLSEEqualizer(Name,Value)` creates an MLSEE object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.MLSEEqualizer(CHANNEL,Name,Value)` creates an MLSEE object, `H`. This object has the `Channel` property set to `CHANNEL`, and the other specified properties set to the specified values.

Properties

`ChannelSource`

Source of channel coefficients

Specify the source of the channel coefficients as one of `Input port` | `Property`. The default is `Property`.

`Channel`

Channel coefficients

Specify the channel as a numeric, column vector containing the coefficients of an FIR filter. The default is [1;0.7;0.5;0.3]. The length of this vector determines the memory length of the channel. This must be an integer multiple of the samples per symbol, that you specify in the SamplesPerSymbol on page 3-0 property. This property applies when you set the ChannelSource on page 3-0 property to Property.

Constellation

Input signal constellation

Specify the constellation of the input modulated signal as a complex vector. The default is [1+1i -1+1i -1-1i 1-1i].

TracebackDepth

Traceback depth of Viterbi algorithm

Specify the number of trellis branches (the number of symbols), the Viterbi algorithm uses to construct each traceback path. The default is 21. The traceback depth influences the decoding accuracy and delay. The decoding delay represents the number of zero symbols that precede the first decoded symbol in the output. When you set the TerminationMethod on page 3-0 property to Continuous, the decoding delay equals the number of zero symbols of this property. When you set the TerminationMethod property to Truncated, there is no output delay.

TerminationMethod

Termination method of Viterbi algorithm

Specify the termination method of the Viterbi algorithm as one of Continuous | Truncated. The default is Truncated. When you set this property to Continuous, the object initializes the Viterbi algorithm metrics of all the states to 0 in the first call to the step method. Then, the object saves its internal state metric at the end of each frame, for use with the next frame. When you set this property to Truncated, the object resets at every frame. The Viterbi algorithm processes each frame of data independently, resetting the state metric at the end of each frame. The traceback path always starts at the state with the minimum metric. The initialization of the state metrics depends on whether you specify a preamble or postamble. If you set the PreambleSource on page 3-0 property to None, the object initializes the metrics of all the states to 0 at the beginning of each data frame. If you set the PreambleSource property to Property, the object uses the preamble that you specify at the Preamble on page 3-0 property, to initialize the state metrics at the beginning of each data frame. When you specify a preamble, the traceback path ends at one of the states represented by that preamble. If you set the PostambleSource on page 3-0 property to None, the traceback path starts at the state with the smallest metric. If you set the PostambleSource property to Property, the traceback path begins at the state represented by the postamble that you specify at the Postamble on page 3-0 property. If the postamble does not decode to a unique state, the decoder identifies the smallest of all possible decoded states that are represented by the postamble. The decoder then begins traceback decoding at that state. When you set this property to Truncated, the step method input data signal must contain at least TracebackDepth on page 3-0 symbols, not including an optional preamble.

ResetInputPort

Enable equalizer reset input

Set this property to true to enable an additional input to the step method. The default is false. When this input is a nonzero, double-precision or logical scalar value, the object resets the states of the equalizer. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

PreambleSource

Source of preamble

Specify the source of the preamble that is expected to precede the input signal. Choose from `None` | `Property`. The default is `None`. Set this property to `Property` to specify a preamble using the `Preamble` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Truncated`.

Preamble

Preamble that precedes input signals

Specify a preamble that is expected to precede the data in the input signal as an integer, row vector. The default is `[0 3 2 1]`. The values of the preamble should be between `0` and *M*-`1`, where *M* is the length of the signal constellation that you specify in the `Constellation` on page 3-0 property. An integer value of *k*–`1` in the vector corresponds to the *k*-th entry in the vector stored in the `Constellation` property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Truncated` and the `PreambleSource` on page 3-0 property to `Property`.

PostambleSource

Source of postamble

Specify the source of the postamble that is expected to follow the input signal. Choose from `None` | `Property`. The default is `None`. Set this property to `Property` to specify a postamble in the `Postamble` on page 3-0 property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Truncated`.

Postamble

Postamble that follows input signals

Specify a postamble that is expected to follow the data in the input signal as an integer row vector. The default is `[0 2 3 1]`. The values of the postamble should be between `0` and *M*–`1`. In this case, *M* indicates the length of the `Constellation` on page 3-0 property. An integer value of *k*–`1` in the vector corresponds to the *k*-th entry in the vector specified in the `Constellation` property. This property applies when you set the `TerminationMethod` on page 3-0 property to `Truncated` and the `PostambleSource` on page 3-0 property to `Property`. The default is `[0 2 3 1]`.

SamplesPerSymbol

Number of samples per symbol

Specify the number of samples per symbol in the input signal as an integer scalar value. The default is `1`.

Methods

reset	Reset states of MLSEE object
step	Equalize using maximum likelihood sequence estimation

Common to All System Objects	
release	Allow System object property value changes

Examples

MLSE Equalize QPSK Signal Through Dispersive Channel

This example shows how to use an MLSE equalizer to remove the effects of a frequency-selective channel.

Specify static channel coefficients.

```
chCoeffs = [.986; .845; .237; .12345+.31i];
```

Create an MLSE equalizer object. Create an error rate calculator object.

```
mlse = comm.MLSEEqualizer('TracebackDepth',10,...
    'Channel',chCoeffs,'Constellation',pskmod(0:3,4,pi/4));
errorRate = comm.ErrorRate;
```

The main processing loop includes these steps:

- Data generation
- QPSK modulation
- Channel filtering
- Signal equalization
- QPSK demodulation
- Error computation

```
for n = 1:50
    data= randi([0 3],100,1);
    modSignal = pskmod(data,4,pi/4,'gray');

    % Introduce channel distortion.
    chanOutput = filter(chCoeffs,1,modSignal);

    % Equalize the channel output and demodulate.
    eqSignal = mlse(chanOutput);
    demodData = pskdemod(eqSignal,4,pi/4,'gray');

    % Compute BER.
    errorStats = errorRate(data,demodData);
end
```

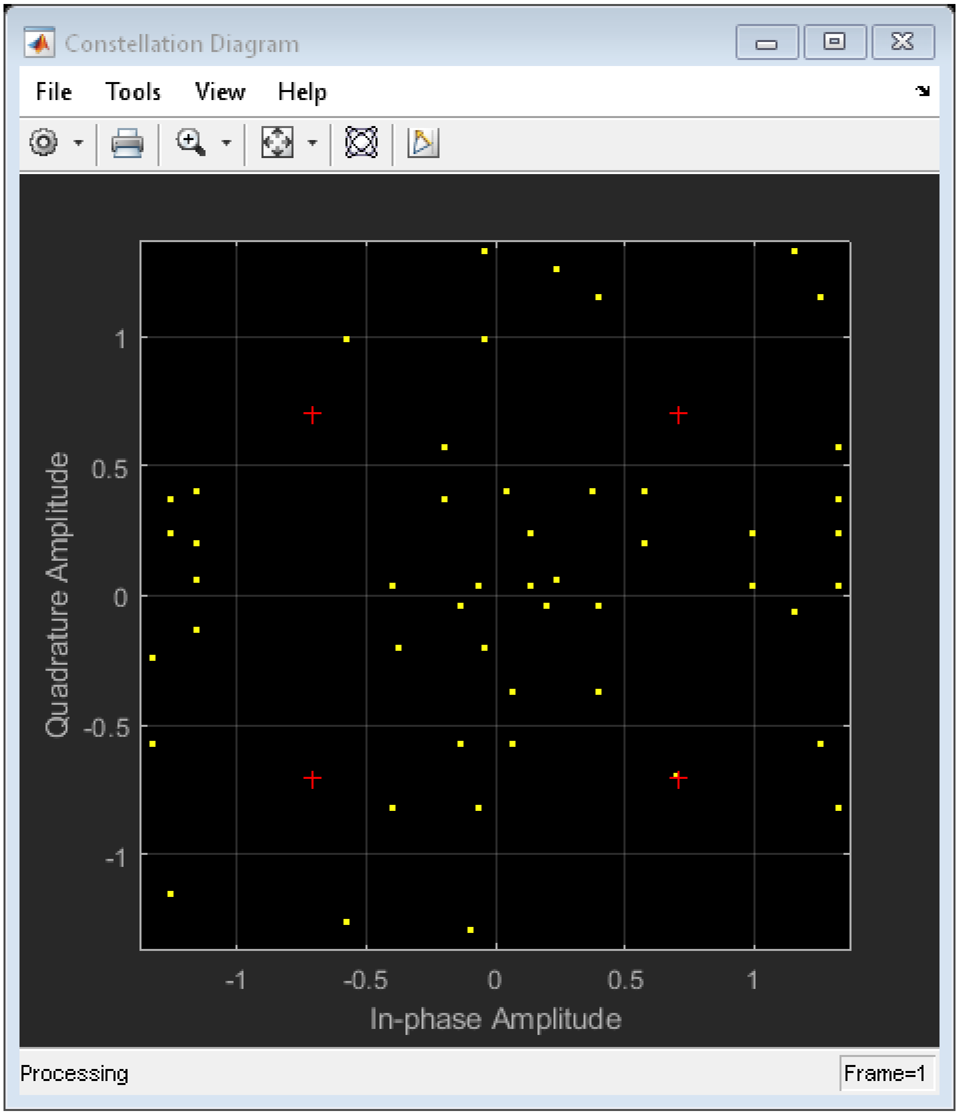
Display the bit error rate and the number of errors.

```
ber = errorStats(1)
numErrors = errorStats(2)
```

```
ber =

     0


numErrors =

     0
```

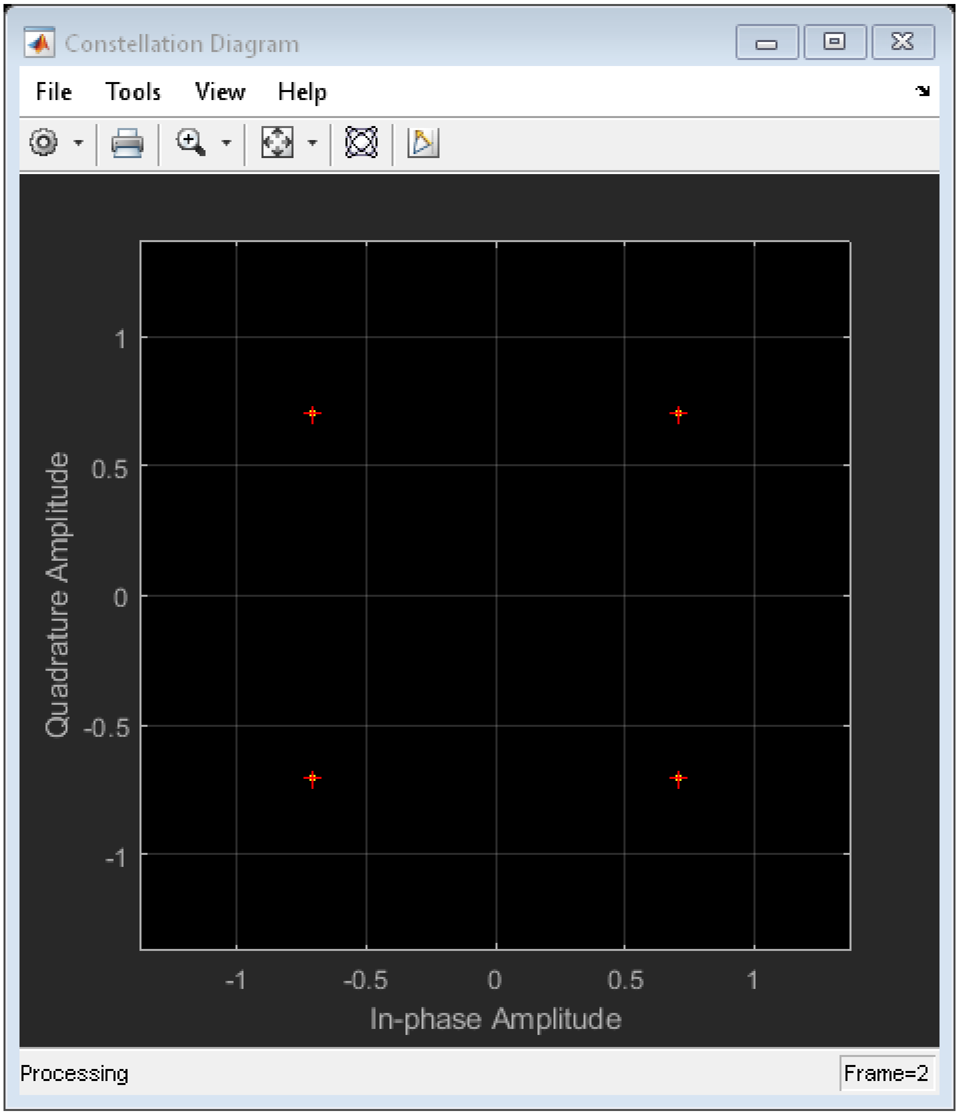
Plot the signal constellation prior to equalization.

```
constDiagram = comm.ConstellationDiagram;
constDiagram(chanOutput)
```



Plot the signal constellation after equalization.

```
constDiagram(eqSignal)
```



The equalized symbols align perfectly with the QPSK reference constellation.

Algorithms

This object implements the algorithm, inputs, and outputs described on the MLSE Equalizer block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
`mlseeq`

Objects
`comm.DecisionFeedbackEqualizer` | `comm.LinearEqualizer` | `comm.ViterbiDecoder`

Blocks
MLSE Equalizer

Topics
“MLSE Equalizers”

Introduced in R2012a

reset

System object: `comm.MLSEEqualizer`
Package: `comm`

Reset states of MLSEE object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MLSEEqualizer` object, `H`.

step

System object: comm.MLSEEqualizer
Package: comm

Equalize using maximum likelihood sequence estimation

Syntax

```
Y = step(H,X)
Y = step(H,X,CHANNEL)
Y = step(H,X,RESET)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) equalizes the linearly modulated data input, X, using the Viterbi algorithm. The step method outputs Y, the maximum likelihood sequence estimate of the signal. Input X must be a column vector of data type double or single.

Y = step(H,X,CHANNEL) uses CHANNEL as the channel coefficients when you set the ChannelSource property to 'Input port'. The channel coefficients input, CHANNEL, must be a numeric, column vector containing the coefficients of an FIR filter in descending order of powers of z. The length of this vector is the channel memory, which must be an integer multiple of the samples per input symbol specified in the SamplesPerSymbol property.

Y = step(H,X,RESET) uses RESET as the reset signal when you set the TerminationMethod property to 'Continuous' and the ResetInputPort property to true. The object resets when RESET has a non-zero value. RESET must be a double precision or logical scalar. You can combine optional input arguments when you set their enabling properties. Optional inputs must be listed in the same order as the order of the enabling properties. For example, Y = step(H,X,CHANNEL,RESET).

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.MSKDemodulator

Package: `comm`

Demodulate using MSK method and the Viterbi algorithm

Description

The comm.MSKDemodulator object demodulates a signal that was modulated using the minimum shift keying method. The object expects the input signal to be a baseband representation of a coherent modulated signal with no precoding. The initial phase offset property sets the initial phase of the modulated waveform.

To demodulate a signal that was modulated using minimum shift keying:

1. Define and set up your MSK demodulator object. See “Construction” on page 3-995.
2. Call `step` to demodulate the signal according to the properties of `comm.MSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input minimum shift keying (MSK) modulated data using the Viterbi algorithm.

`H = comm.MSKDemodulator(Name,Value)` creates an MSK demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer values. The default is `false`.

When you set this property to `false`, the `step` method outputs a column vector with a length equal to *N*/`SamplesPerSymbol` on page 3-0 . *N* represents the length of the input signal, which is the number of input baseband modulated symbols. The elements of the output vector are `-1` or `1`.

When you set the `BitOutput` on page 3-0 property to `true`, the `step` method outputs a binary column vector with a length equal to *N*/`SamplesPerSymbol`. The vector elements are bit values of `0` or `1`.

InitialPhaseOffset

Initial phase offset

Specify the initial phase offset of the input modulated waveform in radians as a real, numeric scalar value. The default is 0.

SamplesPerSymbol

Number of samples per input symbol

Specify the expected number of samples per input symbol as a positive, integer scalar value. The default is 8.

TracebackDepth

Traceback depth for Viterbi algorithm

Specify the number of trellis branches that the Viterbi algorithm uses to construct each traceback path as a positive, integer scalar value. The default is 16. The value of this property is also the output delay This value indicates number of zero symbols that precede the first meaningful demodulated symbol in the output.

OutputDataType

Data type of output

Specify the output data type as one of int8 | int16 | int32 | double, when you set the BitOutput on page 3-0 property to false. The default is double.

When you set the BitOutput property to true, specify the output data type as one of logical | double.

Methods

reset	Reset states of the MSK demodulator object
step	Demodulate using MSK method and the Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate an MSK signal with bit inputs and phase offset

```
% Create an MSK modulator, an AWGN channel, and an MSK demodulator.  Use a
% phase offset of pi/4.
 hMod = comm.MSKModulator('BitInput', true, ...
                    'InitialPhaseOffset', pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.MSKDemodulator('BitOutput', true, ...
                    'InitialPhaseOffset', pi/4);
 % Create an error rate calculator, account for the delay caused by the Viterbi algorithm
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.000000
Number of errors = 0
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK Demodulator Baseband block reference page. The object properties correspond to the block parameters. For MSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.MSKModulator`

Introduced in R2012a

reset

System object: `comm.MSKDemodulator`
Package: `comm`

Reset states of the MSK demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MSKDemodulator` object, `H`.

step

System object: comm.MSKDemodulator
Package: comm

Demodulate using MSK method and the Viterbi algorithm

Syntax

Y = step(H,X)

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) demodulates input data, X, with the MSK demodulator System object, H, and returns Y. X must be a double or single precision column vector with a length equal to an integer multiple of the number of samples per symbol you specify in the SamplesPerSymbol property.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.MSKModulator

Package: comm

Modulate using MSK method

Description

The MSKModulator object modulates using the minimum shift keying method. The output is a baseband representation of the modulated signal. The initial phase offset property sets the initial phase of the output waveform, measured in radians.

To modulate a signal using minimum shift keying:

1. Define and set up your MSK modulator object. See "Construction" on page 3-1001.
2. Call step to modulate the signal according to the properties of comm.MSKModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.MSKModulator creates a modulator System object, H. This object modulates the input signal using the minimum shift keying (MSK) modulation method.

H = comm.MSKModulator(Name,Value) creates an MSK modulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is false.

When you set the BitInput on page 3-0 property to false, the step method input must be a column vector with a double-precision or signed integer data type and of values equal to -1 or 1.

When you set the BitInput property to true, the step method input requires double-precision or logical data type column vector of 0s and 1s.

InitialPhaseOffset

Initial phase offset

Specify the initial phase of the modulated waveform in radians as a real, numeric scalar value. The default is `0`.

`SamplesPerSymbol`

Number of samples per output symbol

Specify the upsampling factor at the output as a real, positive, integer scalar value. The default is `8`. The upsampling factor indicates the number of output samples that the step method produces for each input sample.

`OutputDataType`

Data type of output

Specify output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of the MSK modulator object
step	Modulate using MSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate an MSK signal with bit inputs and phase offset

```
% Create an MSK modulator, an AWGN channel, and an MSK demodulator.  Use a
% phase offset of pi/4.
 hMod = comm.MSKModulator('BitInput', true, ...
                   'InitialPhaseOffset', pi/4);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                   'Signal to noise ratio (SNR)','SNR',0);
    hDemod = comm.MSKDemodulator('BitOutput', true, ...
                   'InitialPhaseOffset', pi/4);
 % Create an error rate calculator, account for the delay caused by the Viterbi algorithm
    hError = comm.ErrorRate('ReceiveDelay', hDemod.TracebackDepth);
    for counter = 1:100
      % Transmit 100 3-bit words
      data = randi([0 1],300,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.000000
Number of errors = 0
```

Map Binary Data to GMSK Signal

This example illustrates the mapping of binary sequences of zeros and ones to the output of a GMSK modulator. The relationship also applies for MSK modulation.

Create a GMSK modulator that accepts binary inputs. Specify the pulse length and samples per symbol to be 1.

```
gmsk = comm.GMSKModulator('BitInput',true,'PulseLength',1,'SamplesPerSymbol',1);
```

Create an input sequence of all zeros. Modulate the sequence.

```
x = zeros(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 - 1.0000i
  -1.0000 + 0.0000i
   0.0000 + 1.0000i
   1.0000 - 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
```

```
theta = 5×1

         0
   -1.5708
   -3.1416
   -4.7124
   -6.2832
```

A sequence of zeros causes the phase to shift by -π/2 between samples.

Reset the modulator. Modulate an input sequence of all ones.

```
reset(gmsk)
x = ones(5,1);
y = gmsk(x)
```

```
y = 5×1 complex

   1.0000 + 0.0000i
  -0.0000 + 1.0000i
  -1.0000 - 0.0000i
   0.0000 - 1.0000i
   1.0000 + 0.0000i
```

Determine the phase angle for each point. Use the `unwrap` function to better show the trend.

```
theta = unwrap(angle(y))
theta = 5×1

         0
    1.5708
    3.1416
    4.7124
    6.2832
```

A sequence of ones causes the phase to shift by +π/2 between samples.

GMSK vs. MSK

Compare Gaussian minimum shift keying (GMSK) and minimum shift keying (MSK) modulation schemes by plotting the eye diagram for GMSK with different pulse lengths and for MSK.

Set the samples per symbol variable.

```
sps = 8;
```

Generate random binary data.

```
data = randi([0 1],1000,1);
```

Create GMSK and MSK modulators that accept binary inputs. Set the `PulseLength` property of the GMSK modulator to `1`.

```
gmskMod = comm.GMSKModulator('BitInput',true,'PulseLength',1, ...
    'SamplesPerSymbol',sps);
mskMod = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',sps);
```

Modulate the data using the GMSK and MSK modulators.

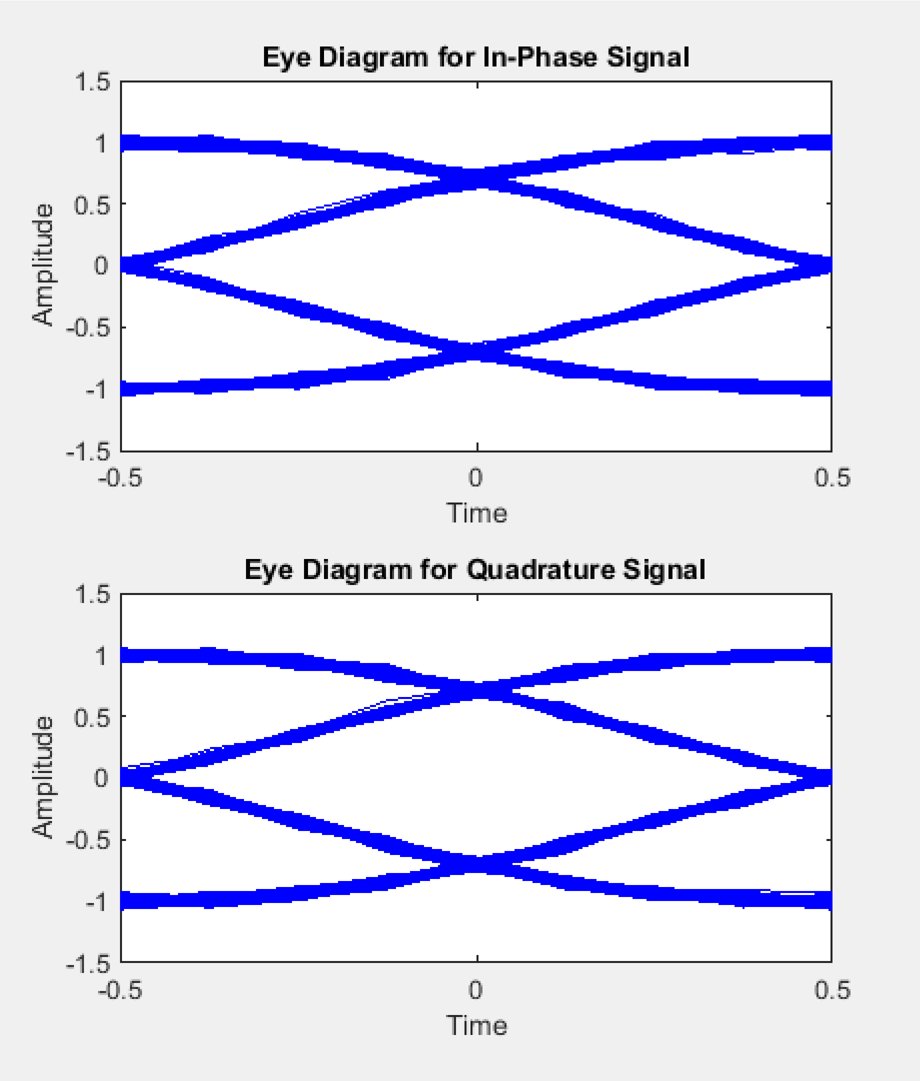
```
modSigGMSK = gmskMod(data);
modSigMSK = mskMod(data);
```

Pass the modulated signals through an AWGN channel having an SNR of `30` dB.

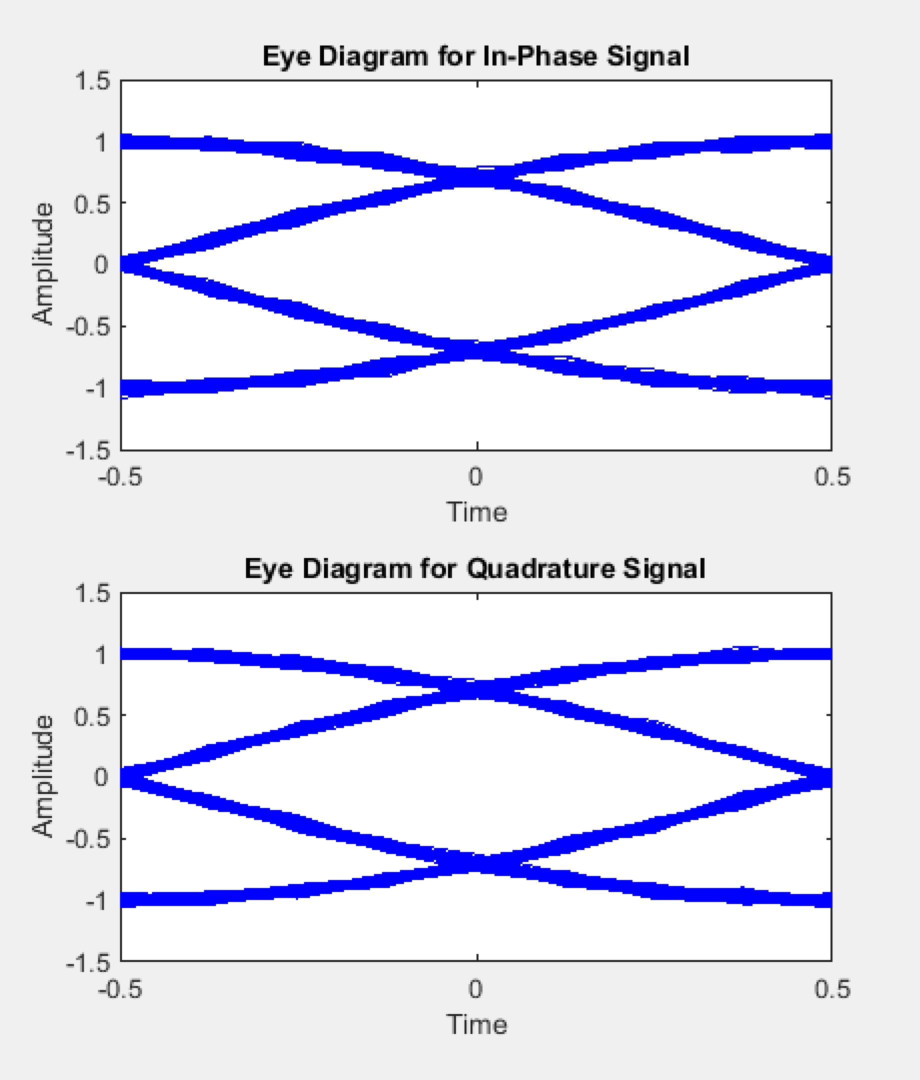
```
rxSigGMSK = awgn(modSigGMSK,30);
rxSigMSK = awgn(modSigMSK,30);
```

Use the `eyediagram` function to plot the eye diagrams of the noisy signals. With the GMSK pulse length set to `1`, the eye diagrams are nearly identical.

```
eyediagram(rxSigGMSK,sps,1,sps/2)
```



```
eyediagram(rxSigMSK,sps,1,sps/2)
```



Set the `PulseLength` property for the GMSK modulator object to `3`. Because the property is nontunable, the object must be released first.

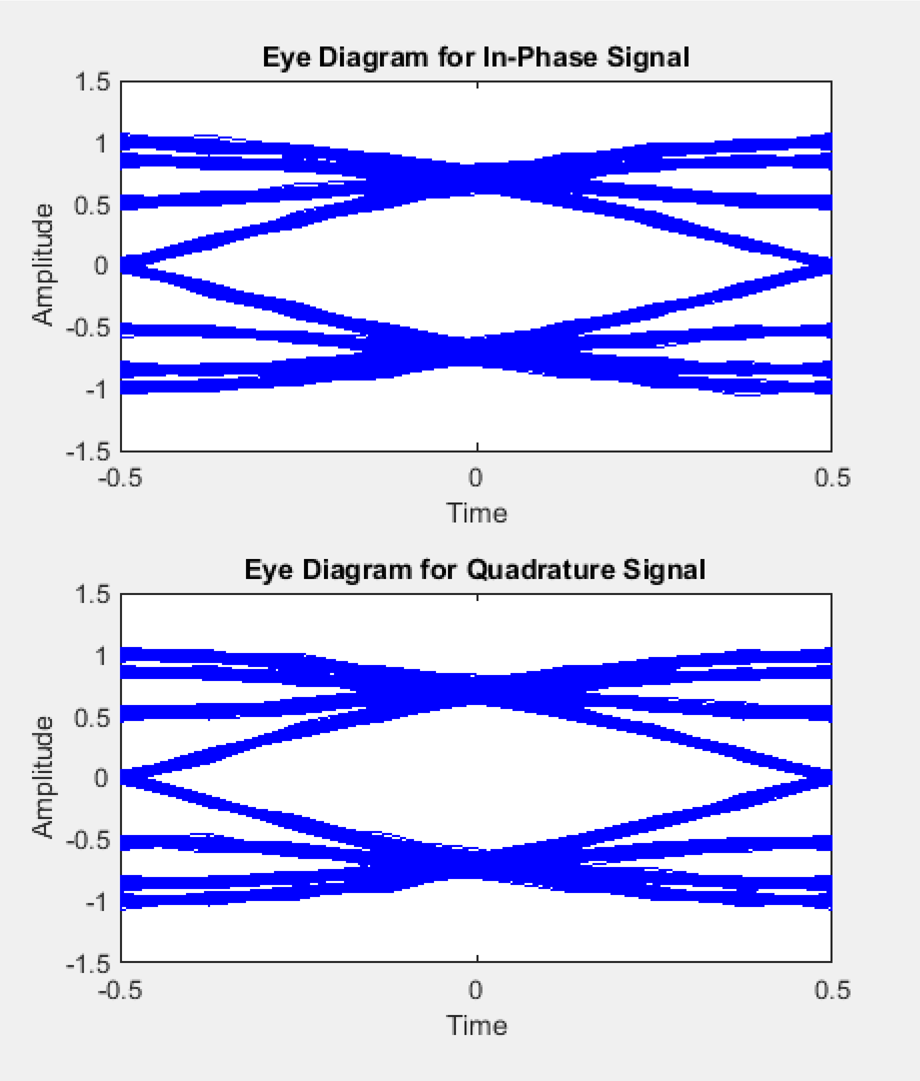
```
release(gmskMod)
gmskMod.PulseLength = 3;
```

Generate a modulated signal using the updated GMSK modulator object and pass it through the AWGN channel.

```
modSigGMSK = gmskMod(data);
rxSigGMSK = awgn(modSigGMSK,30);
```

With continuous phase modulation (CPM) waveforms, such as GSMK, the waveform depends on values of the previous symbols as well as the present symbol. Plot the eye diagram of the GMSK signal to see that the increased pulse length results in an increase in the number of paths in the eye diagram.

```
eyediagram(rxSigGMSK,sps,1,sps/2)
```



Experiment by changing the `PulseLength` parameter of the GMSK modulator object to other values. If you set the property to an even number, you should set `gmskMod.InitialPhaseOffset` to `pi/4` and update the offset argument of the `eyediagram` function from `sps/2` to `0` for a better view of the modulated signal. In order to more clearly view the Gaussian pulse shape, you must use scopes that display the phase of the signal, as described in the “CPM Phase Tree” example.

Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK Demodulator Baseband block reference page. The object properties correspond to the block parameters. For MSK the phase shift per symbol is π/2, which is a modulation index of 0.5.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CPMDemodulator` | `comm.CPMModulator` | `comm.MSKDemodulator`

Introduced in R2012a

reset

System object: `comm.MSKModulator`
Package: `comm`

Reset states of the MSK modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MSKModulator` object, `H`.

step

System object: `comm.MSKModulator`
Package: `comm`

Modulate using MSK method

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the MSK modulator object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be a double precision, signed integer, or logical column vector. The length of output vector, `Y`, is equal to the number of input samples times the number of samples per symbol you specify in the `SamplesPerSymbol` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MSKTimingSynchronizer

Package: comm

Recover symbol timing phase using fourth-order nonlinearity method

Description

The MSKTimingSynchronizer object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This object implements a general non-data-aided feedback method that is independent of carrier phase recovery. This method requires prior compensation for the carrier frequency offset. This object is suitable for systems that use baseband minimum shift keying (MSK) modulation.

To recover the symbol timing phase of the input signal:

1. Define and set up your MSK timing synchronizer object. See “Construction” on page 3-1011.
2. Call step to recover the symbol timing phase of the input signal according to the properties of comm.MSKTimingSynchronizer. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.MSKTimingSynchronizer creates a timing phase synchronizer System object, H. This object recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method.

H = comm.MSKTimingSynchronizer(Name,Value) creates an MSK timing synchronizer object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

SamplesPerSymbol

Number of samples representing each symbol

Specify the number of samples that represent each symbol in the input signal as an integer-valued scalar greater than 1. The default is 4.

ErrorUpdateGain

Error update step size

Specify the step size for updating successive timing phase estimates as a positive, real scalar value. The default is 0.05. Typically, this number is less than 1/SamplesPerSymbol on page 3-0 , which corresponds to a slowly varying timing phase. This property is tunable.

ResetInputPort

Enable synchronization reset input

Set this property to `true` to enable resetting the timing phase recovery process based on an input argument value. The default is `false`.

When you set this property to `true`, you must specify a reset input value to the `step` method.

When the reset input is a nonzero value, the object restarts the timing phase recovery process. When you set this property to `false`, the object does not restart.

ResetCondition

Condition for timing phase recovery reset

Specify the conditions to reset the timing phase recovery process as one of `Never` | `Every frame`. The default is `Never`.

When you set this property to `Never`, the phase recovery process never restarts. The object operates continuously, retaining information from one symbol to the next.

When you set this property to `Every frame`, the timing phase recovery restarts at the start of each frame of data. Thus, each time the object calls the step method. This property applies when you set the `ResetInputPort` on page 3-0 property to `false`.

Methods

reset	Reset states of MSK timing phase synchronizer object
step	Recover symbol timing phase using fourth-order nonlinearity method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Recover Timing Phase of MSK Signal

Create MSK modulator, variable fractional delay, and MSK timing synchronizer System objects.

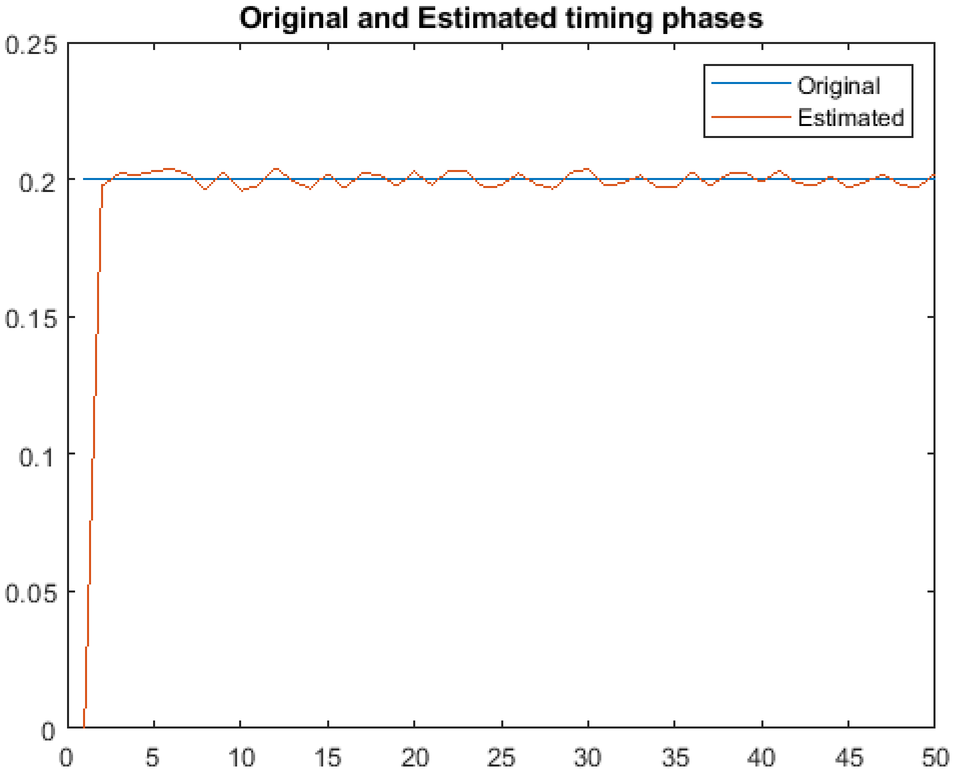
```
mskMod = comm.MSKModulator('BitInput',true,'SamplesPerSymbol',14);
timingOffset = 0.2;
varDelay = dsp.VariableFractionalDelay;
mskTimingSync = comm.MSKTimingSynchronizer('SamplesPerSymbol',14,'ErrorUpdateGain', 0.05);
```

Main processing loop.

```
phEst = zeros(50,1);
for i = 1:50
    data = randi([0 1],100,1);    % Generate data
    modData = mskMod(data);       % Modulate data
    % Apply timing offset error.
    impairedData = varDelay(modData,timingOffset*14);
    % Perform timing phase recovery.
    [~,phase] = mskTimingSync(impairedData);
    phEst(i) = phase(1)/14;
end
```

Plot the results.

```
plot(1:50,[0.2*ones(50,1) phEst]);
legend( 'Original','Estimated')
title('Original and Estimated timing phases');
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the MSK-Type Signal Timing Recovery block reference page. The object properties correspond to the block parameters, except:

- The object corresponds to the MSK-Type Signal Timing Recovery block with the **Modulation type** parameter set to MSK.
- The **Reset** parameter corresponds to the ResetInputPort on page 3-0 and ResetCondition on page 3-0 properties.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.SymbolSynchronizer`

Introduced in R2012a

reset

System object: `comm.MSKTimingSynchronizer`
Package: `comm`

Reset states of MSK timing phase synchronizer object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of `MSKTimingSynchronizer` object, `H`.

step

System object: `comm.MSKTimingSynchronizer`
Package: `comm`

Recover symbol timing phase using fourth-order nonlinearity method

Syntax

```
[Y,PHASE] = step(H,X)
[Y,PHASE] = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,PHASE] = step(H,X)` recovers the timing phase and returns the time-synchronized signal, `Y`, and the estimated timing phase, `PHASE`, for input signal `X`. `X` must be a double or single precision complex column vector.

`[Y,PHASE] = step(H,X,R)` restarts the timing phase recovery process when you input a reset signal, `R`, that is non-zero. `R` must be a logical or double scalar. This syntax applies when you set the `ResetInputPort` property to true.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MultiplexedDeinterleaver

Package: `comm`

Deinterleave input symbols using set of shift registers with specified delays

Description

The `MultiplexedDeinterleaver` object restores the original ordering of a sequence that was interleaved using the General Multiplexed Interleaver object.

To deinterleave the input symbols:

1. Define and set up your multiplexed deinterleaver object. See “Construction” on page 3-1017.
2. Call `step` to restore the original ordering of the input sequence according to the properties of `comm.MultiplexedDeinterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MultiplexedDeinterleaver` creates a multiplexed deinterleaver System object, `H`. This object restores the original ordering of a sequence that was interleaved using the multiplexed interleaver System object.

`H = comm.MultiplexedDeinterleaver(Name,Value)` creates a multiplexed deinterleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Delay`

Interleaver delay

Specify the lengths of the shift registers as an integer column vector. The default is `[2;0;1;3;10]`.

`InitialConditions`

Initial conditions of shift registers

Specify the initial values in each shift register as a numeric scalar value or a column vector. The default is `0`. When you set this property to a column vector, the vector length must equal the value of the `Delay` on page 3-0 property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*th shift register.

Methods

reset	Reset states of the multiplexed deinterleaver object
step	Deinterleave input symbols using a set of shift registers with specified delays

Common to All System Objects	
release	Allow System object property value changes

Examples

Multiplexed Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.MultiplexedInterleaver('Delay',[1; 0; 2; 1]);
deinterleaver = comm.MultiplexedDeinterleaver('Delay',[1; 0; 2; 1]);
```

Generate a random data sequence. Pass the data sequence through the interleaver and deinterleaver.

```
[dataIn,dataOut] = deal([]);            % Initialize data arrays

for k = 1:50
    data = randi([0 7],20,1);           % Generate data sequence
    intData = interleaver(data);        % Interleave sequence
    deIntData = deinterleaver(intData); % Deinterleave sequence

    dataIn = cat(1,dataIn,data);        % Save original data
    dataOut = cat(1,dataOut,deIntData); % Save deinterleaved data
end
```

Determine the delay through the interleaver and deinterleaver.

```
intlvrDelay = finddelay(dataIn,dataOut)
```

```
intlvrDelay = 8
```

After accounting for the delay, confirm that the original and deinterleaved sequences are identical.

```
isequal(dataIn(1:end-intlvrDelay),dataOut(intlvrDelay+1:end))
```

```
ans = logical
   1
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Multiplexed Deinterleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalDeinterleaver` | `comm.MultiplexedInterleaver`

Introduced in R2012a

reset

System object: `comm.MultiplexedDeinterleaver`
Package: `comm`

Reset states of the multiplexed deinterleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MultiplexedDeinterleaver` object, `H`.

step

System object: `comm.MultiplexedDeinterleaver`
Package: `comm`

Deinterleave input symbols using a set of shift registers with specified delays

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` restores the original ordering of the sequence, `X`, that was interleaved using a multiplexed interleaver and returns `Y`. The input `X` must be a column vector. The data type for `X` can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The multiplexed deinterleaver object uses N shift registers, where N is the number of elements in the vector specified by the `Delay` property. When a new input symbol enters the deinterleaver, a commutator switches to a new register. The new symbol shifts in while the oldest symbol in that register is shifted out. When the commutator reaches the Nth register, upon the next new input, it returns to the first register. The multiplexed deinterleaver associated with a multiplexed interleaver has the same number of registers as the interleaver. The delay in a particular deinterleaver register depends on the largest interleaver delay minus the interleaver delay for the given register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.MultiplexedInterleaver

Package: `comm`

Permute input symbols using set of shift registers with specified delays

Description

The `MultiplexedInterleaver` object permutes the symbols in the input signal. Internally, the object uses a set of shift registers, each with its own delay value.

To permute the symbols in the input signal:

1. Define and set up your multiplexed interleaver object. See “Construction” on page 3-1022.
2. Call `step` to interleave the input signal according to the properties of `comm.MultiplexedInterleaver`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.MultiplexedInterleaver` creates a multiplexed interleaver System object, `H`. This object permutes the symbols in the input signal using a set of shift registers with specified delays.

`H = comm.MultiplexedInterleaver(Name,Value)` creates a multiplexed interleaver object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`Delay`

Interleaver delay

Specify the lengths of the shift registers as an integer column vector. The default is `[2;0;1;3;10]`.

`InitialConditions`

Initial conditions of shift registers

Specify the initial values in each shift register as a numeric scalar value or a column vector. The default is `0`. When you set this property to a column vector, the length must equal the value of the `Delay` on page 3-0 property. This vector contains initial conditions, where the *i*-th initial condition is stored in the *i*-th shift register.

Methods

reset	Reset states of the multiplexed interleaver object
step	Permute input symbols using a set of shift registers with specified delays

Common to All System Objects	
release	Allow System object property value changes

Examples

Multiplexed Interleaving and Deinterleaving

Create interleaver and deinterleaver objects.

```
interleaver = comm.MultiplexedInterleaver('Delay',[1; 0; 2; 1]);
deinterleaver = comm.MultiplexedDeinterleaver('Delay',[1; 0; 2; 1]);
```

Generate a random data sequence. Pass the data sequence through the interleaver and deinterleaver.

```
[dataIn,dataOut] = deal([]);            % Initialize data arrays

for k = 1:50
    data = randi([0 7],20,1);           % Generate data sequence
    intData = interleaver(data);        % Interleave sequence
    deIntData = deinterleaver(intData); % Deinterleave sequence

    dataIn = cat(1,dataIn,data);        % Save original data
    dataOut = cat(1,dataOut,deIntData); % Save deinterleaved data
end
```

Determine the delay through the interleaver and deinterleaver.

```
intlvrDelay = finddelay(dataIn,dataOut)
```

```
intlvrDelay = 8
```

After accounting for the delay, confirm that the original and deinterleaved sequences are identical.

```
isequal(dataIn(1:end-intlvrDelay),dataOut(intlvrDelay+1:end))
```

```
ans = logical
   1
```

Copyright 2012 The MathWorks, Inc.

Algorithms

This object implements the algorithm, inputs, and outputs described on the General Multiplexed Interleaver block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalInterleaver` | `comm.MultiplexedDeinterleaver`

Introduced in R2012a

reset

System object: `comm.MultiplexedInterleaver`
Package: `comm`

Reset states of the multiplexed interleaver object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `MultiplexedInterleaver` object, `H`.

step

System object: `comm.MultiplexedInterleaver`
Package: `comm`

Permute input symbols using a set of shift registers with specified delays

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` permutes input sequence, `X`, and returns interleaved sequence, `Y`. The input `X` must be a column vector and the data type can be numeric, logical, or fixed-point (fi objects). `Y` has the same data type as `X`. The multiplexed interleaver object consists of N registers, each with a specified delay. With each new input symbol, a commutator switches to a new register and the new symbol is shifted in while the oldest symbol in that register is shifted out. When the commutator reaches the Nth register, upon the next new input, it returns to the first register.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.OQPSKDemodulator

Package: `comm`

Demodulation using OQPSK method

Description

The comm.OQPSKDemodulator object applies pulse shape filtering to the input waveform and demodulates it using the offset quadrature phase shift keying (OQPSK) method. For more information, see “Pulse Shaping Filter” on page 3-1033. The input is a baseband representation of the modulated signal.

For information about delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 3-1033.

To demodulate a signal that is OQPSK modulated:

1. Create the `comm.OQPSKDemodulator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
oqpskdemod = comm.OQPSKDemodulator
oqpskdemod = comm.OQPSKDemodulator(mod)
oqpskdemod = comm.OQPSKDemodulator(Name,Value)
oqpskdemod = comm.OQPSKDemodulator(phase,Name,Value)
```

Description

`oqpskdemod = comm.OQPSKDemodulator` creates a demodulator System object. This object can jointly match-filter and decimate a waveform, and demodulate it using the offset quadrature phase shift keying (OQPSK) method.

`oqpskdemod = comm.OQPSKDemodulator(mod)` creates a demodulator System object with symmetric configuration to the OQPSK modulator object, `mod`.

`oqpskdemod = comm.OQPSKDemodulator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.OQPSKDemodulator('BitOutput',true)`

`oqpskdemod = comm.OQPSKDemodulator(phase,Name,Value)` sets the PhaseOffset property of the created object to `phase` and sets any other specified `Name`, `Value` pairs.

Example: `comm.OQPSKDemodulator(0.5*pi,'SamplesPerSymbol',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`PhaseOffset` — Phase of zeroth point of signal constellation

`0` (default) | scalar

Phase offset from π/4, specified as a scalar in radians. The phase offset is applied to the zeroth point of the signal constellation before delay of quadrature component. After the OQPSK imaginary-component delay the signal is normalized with unity power.

Example: `'PhaseOffset',pi/4` aligns the zeroth point of the QPSK signal constellation point on the axes, {(1,0), (0,j), (-1,0), (0,-j)}.

Data Types: `double`

`BitOutput` — Option to output data as bits

`false` (default) | `true`

Option to output data as bits, specified as `false` or `true`.

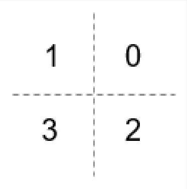
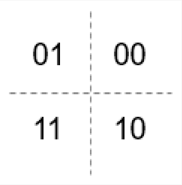
- When you set this property to `false`, the object outputs a column vector of integer values with a length equal to the number of demodulated symbols. The output values are integer representations of two bits and range from `0` to `3`.
- When you set this property to `true`, the object outputs a binary column vector of bit values. The output vector length is twice as long as the number of input symbols.

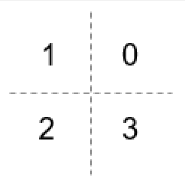
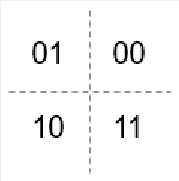
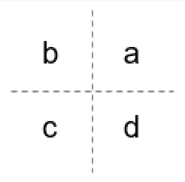
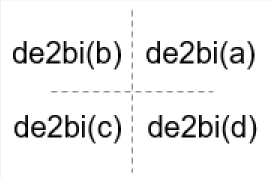
Data Types: `logical`

`SymbolMapping` — Signal constellation bit mapping

`'Gray'` (default) | `'Binary'` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `'Gray'`, `'Binary'`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Gray`	 1 0 3 2	 01 00 11 10	The signal constellation mapping is Gray-encoded.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Binary`			The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3			Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Data Types: `char` | `double`

`PulseShape` — Filtering pulse shape
`'Half sine'` (default) | `'Normal raised cosine'` | `'Root raised cosine'` | `'Custom'`

Filtering pulse shape, specified as `'Half sine'`, `'Normal raised cosine'` | `'Root raised cosine'`, or `'Custom'`.

Data Types: `char`

`RolloffFactor` — Raised cosine filter rolloff factor
`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property is enabled when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterSpanInSymbols` — Filter length
`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to `FilterSpanInSymbols` symbols.

Dependencies

This property is enabled when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterNumerator` — Filter numerator
`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This property is enabled when PulseShape is `'Custom'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

4 (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

`OutputDataType` — Data type assigned to output

`'double'` (default) | `'single'` | `'uint8'`

Data type assigned to output, specified as `'double'`, `'single'`, or `'uint8'`.

Data Types: `char`

Usage

Syntax

```
outsignal = oqpskdemod(waveform)
```

Description

`outsignal = oqpskdemod(waveform)` returns the demodulated output signal. The object produces one output symbol for each input pulse.

Input Arguments

`waveform` — Received waveform

scalar | column vector

Received waveform, specified as a complex scalar or column vector.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`outsignal` — Demodulated signal

integer vector | bit vector

Demodulated signal, returned as an N_S-element integer vector or bit vector, where N_S is the number of samples.

The received waveform is pulse shaped according to the configuration properties PulseShape and SamplesPerSymbol. The setting of the BitOutput property determines the interpretation of the received waveform.

Data Types: `double`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

OQPSK Signal in AWGN

Create an OQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
oqpskmod = comm.OQPSKModulator('BitInput',true);
oqpskdemod = comm.OQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
```

Create an error rate calculator. To account for the delay between the modulator and demodulator, set the `ReceiveDelay` property to `2`.

```
errorRate = comm.ErrorRate('ReceiveDelay',2);
```

Process 300 frames of data looping through these steps.

- Generate vectors with 100 elements of random binary data.
- OQPSK-modulate the data. The data frames are processed as 50 sample frames of 2-bit binary data.
- Pass the modulated data through the AWGN channel.
- OQPSK-demodulate the data.
- Collect error statistics on the frames of data.

```
for counter = 1:300
    txData = randi([0 1],100,1);
    modSig = oqpskmod(txData);
    rxSig = channel(modSig);
    rxData = oqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 3.3336e-05
```

```
numErrors = errorStats(2)
numErrors = 1
```

```
numBits = errorStats(3)
```

```
numBits = 29998
```

OQPSK Signal with Root Raised Cosine Filtering

Perform OQPSK modulation and demodulation and apply root raised cosine filtering to a waveform.

System initialization

Define simulation parameters and create objects for OQPSK modulation and demodulation.

```
sps = 12; % samples per symbol
bits = randi([0, 1], 800, 1); % transmission data

modulator = comm.OQPSKModulator('BitInput',true,'SamplesPerSymbol',sps,'PulseShape','Root raised
demodulator = comm.OQPSKDemodulator(modulator);
```

Waveform transmission and reception

Use the `modulator` object to apply OQPSK modulation and transmit filtering to the input data.

```
oqpskWaveform = modulator(bits);
```

Pass the waveform through a channel.

```
snr = 0;
rxWaveform = awgn(oqpskWaveform, snr);
```

Use the `demodulator` object to apply receive filtering and OQPSK demodulation to the waveform.

```
demodData = demodulator(rxWaveform);
```

Compute the bit error rate to confirm the quality of the data recovery.

```
delay = (1+modulator.BitInput)*modulator.FilterSpanInSymbols;
[~, ber] = biterr(bits(1:end-delay), demodData(delay+1:end))
```

```
ber = 0
```

Soft-Decision OQPSK Modulation-Demodulation

Use the `qamdemod` function to simulate soft decision output for OQPSK-modulated signals.

Generate an OQPSK modulated signal.

```
sps = 4;
msg = randi([0 1],1000,1);
oqpskMod = comm.OQPSKModulator('SamplesPerSymbol',sps,'BitInput',true);
oqpskSig = oqpskMod(msg);
```

Add noise to the generated signal.

```
impairedSig = awgn(oqpskSig,15);
```

Perform Soft-Decision Demodulation

Create QPSK equivalent signal to align in-phase and quadrature.

```
impairedQPSK = complex(real(impairedSig(1+sps/2:end-sps/2)), imag(impairedSig(sps+1:end)));
```

Apply matched filtering to the received OQPSK signal.

```
halfSinePulse = sin(0:pi/sps:(sps)*pi/sps);
matchedFilter = dsp.FIRDecimator(sps,halfSinePulse,'DecimationOffset',sps/2);
filteredQPSK = matchedFilter(impairedQPSK);
```

To perform soft demodulation of the filtered OQPSK signal use the `qamdemod` function. Align symbol mapping of `qamdemod` with the symbol mapping used by the `comm.OQPSKModulator`, then demodulate the signal.

```
oqpskModSymbolMapping = [1 3 0 2];
demodulated = qamdemod(filteredQPSK,4,oqpskModSymbolMapping,'OutputType','llr');
```

More About

Modulation Delays

Digital modulation and demodulation objects incur delays between their inputs and outputs that result in an offset in the arrival sample of the received data. When comparing transmitted data with received data, such as when plotting or computing error statistics, you must take system delays into account. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter and the input/output settings of the object pairs.

Pulse Shape	Input/Output Data (*)	End-to-End Delay Incurred by ay an OQPSK Modulator-Demodulator Object Pair (in samples)
`'Half sine'` or `'Custom'`	Integer	1
	Bit	2
`'Normal raised cosine'` or `'Root raised cosine'`	Integer	`FilterSpanInSymbols`
	Bit	2*`FilterSpanInSymbols`
(*) Set the data type property (`BitInput` for modulation or `BitOutput` for demodulation) to `false` for integer data and `true` for bit data.		

Pulse Shaping Filter

The OQPSK modulation scheme requires oversampling of two or greater in order to delay (or offset) the quadrature channel by 90 degrees. This oversampling is achieved through interpolation filtering implemented by pulse shaping.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

Objects
`comm.OQPSKModulator` | `comm.QPSKDemodulator`

Blocks
OQPSK Demodulator Baseband

Topics
Phase Modulation

Introduced in R2012a

comm.OQPSKModulator

Package: `comm`

Modulation using OQPSK method

Description

The comm.OQPSKModulator object modulates the input signal using the offset quadrature phase shift keying (OQPSK) method and applies pulse shape filtering to the output waveform. For more information, see “Pulse Shaping Filter” on page 3-1042. The output is a baseband representation of the modulated signal.

For information about delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 3-1042.

To modulate a signal using offset quadrature phase shift keying:

1. Create the `comm.OQPSKModulator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
oqpskmod = comm.OQPSKModulator
oqpskmod = comm.OQPSKModulator(demod)
oqpskmod = comm.OQPSKModulator(Name,Value)
oqpskmod = comm.OQPSKModulator(phase,Name,Value)
```

Description

`oqpskmod = comm.OQPSKModulator` creates a modulator System object. This object applies offset quadrature phase shift keying (OQPSK) modulation and pulse shape filtering to the input signal.

`oqpskmod = comm.OQPSKModulator(demod)` creates a modulator System object with symmetric configuration to the OQPSK demodulator object, `demod`.

`oqpskmod = comm.OQPSKModulator(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in single quotes.

Example: `comm.OQPSKModulator('BitInput',true)`

`oqpskmod = comm.OQPSKModulator(phase,Name,Value)` sets the PhaseOffset property of the created object to `phase` and sets any other specified `Name`, `Value` pairs.

Example: `comm.OQPSKModulator(0.5*pi,'SymbolMapping','Binary')`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`PhaseOffset` — Phase of zeroth point of signal constellation

`0` (default) | scalar

Phase offset from π/4, specified as a scalar in radians. The phase offset is applied to the zeroth point of the signal constellation before delay of quadrature component. After the OQPSK imaginary-component delay, the signal is normalized with unity power.

Example: `'PhaseOffset',pi/4` aligns the zeroth point of the QPSK signal constellation point on the axes, {(1,0), (0,j), (-1,0), (0,-j)}.

Data Types: `double`

`BitInput` — Option to provide input in bits

`false` (default) | `true`

Option to provide input in bits, specified as `false` or `true`.

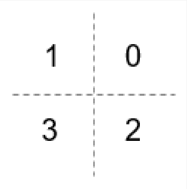
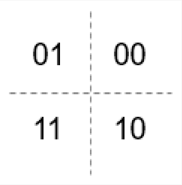
- When this property is set to `false`, the input values must be integer representations of two-bit input segments and range from 0 to 3.
- When this property is set to `true`, the input must be a binary vector of even length. Element pairs are binary representations of integers.

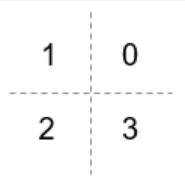
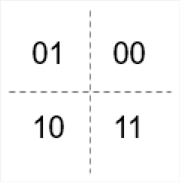
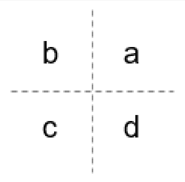
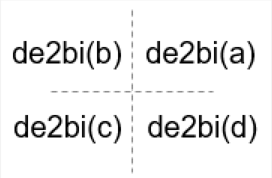
Data Types: `logical`

`SymbolMapping` — Signal constellation bit mapping

`'Gray'` (default) | `'Binary'` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `'Gray'`, `'Binary'`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Gray`			The signal constellation mapping is Gray-encoded.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Binary`			The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3			Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Data Types: `char` | `double`

`PulseShape` — Filtering pulse shape

`'Half sine'` (default) | `'Normal raised cosine'` | `'Root raised cosine'` | `'Custom'`

Filtering pulse shape, specified as `'Half sine'`, `'Normal raised cosine'`, `'Root raised cosine'`, or `'Custom'`.

Data Types: `char`

`RolloffFactor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property is enabled when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterSpanInSymbols` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to `FilterSpanInSymbols` symbols.

Dependencies

This property is enabled when PulseShape is `'Normal raised cosine'` or `'Root raised cosine'`.

Data Types: `double`

`FilterNumerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This property is enabled when PulseShape is `'Custom'`.

Data Types: `double`

`SamplesPerSymbol` — Number of samples per symbol

4 (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

`OutputDataType` — Data type assigned to output

`'double'` (default) | `'single'`

Data type assigned to output, specified as `'double'` or `'single'`.

Data Types: `char`

Usage

Syntax

`waveform = oqpskmod(insignal)`

Description

`waveform = oqpskmod(insignal)` returns baseband-modulated output. The output waveform is pulse shaped according to the configuration properties PulseShape and SamplesPerSymbol.

Input Arguments

`insignal` — Input signal

integer column vector | bit column vector

Input signal, specified as an N_S-element column vector of integers or bits, where N_S is the number of samples.

The setting of the BitInput property determines the interpretation of the input vector.

Data Types: `double`

Output Arguments

`waveform` — Output waveform

vector

Output waveform, returned as a vector. The output waveform is pulse-shaped according to the configuration properties PulseShape and SamplesPerSymbol.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

OQPSK Signal in AWGN

Create an OQPSK modulator and demodulator pair. Create an AWGN channel object having two bits per symbol.

```
oqpskmod = comm.OQPSKModulator('BitInput',true);
oqpskdemod = comm.OQPSKDemodulator('BitOutput',true);
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
```

Create an error rate calculator. To account for the delay between the modulator and demodulator, set the `ReceiveDelay` property to `2`.

```
errorRate = comm.ErrorRate('ReceiveDelay',2);
```

Process 300 frames of data looping through these steps.

- Generate vectors with 100 elements of random binary data.
- OQPSK-modulate the data. The data frames are processed as 50 sample frames of 2-bit binary data.
- Pass the modulated data through the AWGN channel.
- OQPSK-demodulate the data.
- Collect error statistics on the frames of data.

```
for counter = 1:300
    txData = randi([0 1],100,1);
    modSig = oqpskmod(txData);
    rxSig = channel(modSig);
    rxData = oqpskdemod(rxSig);
    errorStats = errorRate(txData,rxData);
end
```

Display the error statistics.

```
ber = errorStats(1)
```

```
ber = 3.3336e-05
```

```
numErrors = errorStats(2)
```

```
numErrors = 1
```

```
numBits = errorStats(3)
```

```
numBits = 29998
```

Create OQPSK Modulator Using Demodulator

Use an OQPSK demodulator object to initialize an OQPSK modulator object while creating it.

Create an OQPSK demodulator, assigning it a phase offset of $\frac{1}{2}\pi$.

```
phase = 0.5*pi;
oqpskdemod = comm.OQPSKDemodulator(phase)
```

```
oqpskdemod = 
  comm.OQPSKDemodulator with properties:

   Modulation
         PhaseOffset: 1.5708
       SymbolMapping: 'Gray'
           BitOutput: false

   Filtering
          PulseShape: 'Half sine'
    SamplesPerSymbol: 4

      OutputDataType: 'double'
```

Use the demodulator object to initialize an OQPSK modulator while creating it.

```
oqpskmod = comm.OQPSKModulator(oqpskdemod)
```

```
oqpskmod = 
  comm.OQPSKModulator with properties:

   Modulation
         PhaseOffset: 1.5708
       SymbolMapping: 'Gray'
            BitInput: false

   Filtering
          PulseShape: 'Half sine'
    SamplesPerSymbol: 4

      OutputDataType: 'double'
```

OQPSK Signal with Root Raised Cosine Filtering

Perform OQPSK modulation and demodulation and apply root raised cosine filtering to a waveform.

System initialization

Define simulation parameters and create objects for OQPSK modulation and demodulation.

```
sps = 12; % samples per symbol
bits = randi([0, 1], 800, 1); % transmission data
```

```
modulator = comm.OQPSKModulator('BitInput',true,'SamplesPerSymbol',sps,'PulseShape','Root raised
demodulator = comm.OQPSKDemodulator(modulator);
```

Waveform transmission and reception

Use the `modulator` object to apply OQPSK modulation and transmit filtering to the input data.

```
oqpskWaveform = modulator(bits);
```

Pass the waveform through a channel.

```
snr = 0;
rxWaveform = awgn(oqpskWaveform, snr);
```

Use the `demodulator` object to apply receive filtering and OQPSK demodulation to the waveform.

```
demodData = demodulator(rxWaveform);
```

Compute the bit error rate to confirm the quality of the data recovery.

```
delay = (1+modulator.BitInput)*modulator.FilterSpanInSymbols;
[~, ber] = biterr(bits(1:end-delay), demodData(delay+1:end))
```

```
ber = 0
```

Soft-Decision OQPSK Modulation-Demodulation

Use the `qamdemod` function to simulate soft decision output for OQPSK-modulated signals.

Generate an OQPSK modulated signal.

```
sps = 4;
msg = randi([0 1],1000,1);
oqpskMod = comm.OQPSKModulator('SamplesPerSymbol',sps,'BitInput',true);
oqpskSig = oqpskMod(msg);
```

Add noise to the generated signal.

```
impairedSig = awgn(oqpskSig,15);
```

Perform Soft-Decision Demodulation

Create QPSK equivalent signal to align in-phase and quadrature.

```
impairedQPSK = complex(real(impairedSig(1+sps/2:end-sps/2)), imag(impairedSig(sps+1:end)));
```

Apply matched filtering to the received OQPSK signal.

```
halfSinePulse = sin(0:pi/sps:(sps)*pi/sps);
matchedFilter = dsp.FIRDecimator(sps,halfSinePulse,'DecimationOffset',sps/2);
filteredQPSK = matchedFilter(impairedQPSK);
```

To perform soft demodulation of the filtered OQPSK signal use the `qamdemod` function. Align symbol mapping of `qamdemod` with the symbol mapping used by the `comm.OQPSKModulator`, then demodulate the signal.

```
oqpskModSymbolMapping = [1 3 0 2];
demodulated = qamdemod(filteredQPSK,4,oqpskModSymbolMapping,'OutputType','llr');
```

More About

Modulation Delays

Digital modulation and demodulation objects incur delays between their inputs and outputs that result in an offset in the arrival sample of the received data. When comparing transmitted data with received data, such as when plotting or computing error statistics, you must take system delays into account. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter and the input/output settings of the object pairs.

Pulse Shape	Input/Output Data (*)	End-to-End Delay Incurred by ay an OQPSK Modulator-Demodulator Object Pair (in samples)
`'Half sine'` or `'Custom'`	Integer	1
	Bit	2
`'Normal raised cosine'` or `'Root raised cosine'`	Integer	`FilterSpanInSymbols`
	Bit	`2*FilterSpanInSymbols`
(*) Set the data type property (`BitInput` for modulation or `BitOutput` for demodulation) to `false` for integer data and `true` for bit data.		

Pulse Shaping Filter

The OQPSK modulation scheme requires oversampling of two or greater in order to delay (or offset) the quadrature channel by 90 degrees. This oversampling is achieved through interpolation filtering implemented by pulse shaping.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

Objects

`comm.OQPSKDemodulator` | `comm.QPSKModulator`

Blocks

OQPSK Modulator Baseband

Topics

Phase Modulation

Introduced in R2012a

comm.OSTBCCombiner

Package: comm

Combine inputs using orthogonal space-time block code

Description

The OSTBCCombiner object combines the input signal (from all of the receive antennas) and the channel estimate signal to extract the soft information of the symbols encoded by an OSTBC. The input channel estimate does not need to be constant and can vary at each call to the step method. The combining algorithm uses only the estimate for the first symbol period per codeword block. A symbol demodulator or decoder would follow the Combiner object in a MIMO communications system.

To combine input signals and extract the soft information of the symbols encoded by an OSTBC:

1. Define and set up your OSTBC combiner object. See "Construction" on page 3-1044.
2. Call step to Combine inputs using an orthogonal space-time block code according to the properties of comm.OSTBCCombiner. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.OSTBCCombiner creates an orthogonal space-time block code (OSTBC) combiner System object, H. This object combines the input signal (from all of the receive antennas) with the channel estimate signal to extract the soft information of the symbols encoded by an OSTBC.

H = comm.OSTBCCombiner(Name,Value) creates an OSTBC Combiner object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.OSTBCCombiner(N,M,Name,Value) creates an OSTBC Combiner object, H. This object has the NumTransmitAntennas property set to N, the NumReceiveAntennas property set to N, and the other specified properties set to the specified values.

Properties

NumTransmitAntennas

Number of transmit antennas

Specify the number of antennas at the transmitter as 2 | 3 | 4. The default is 2.

SymbolRate

Symbol rate of code

Specify the symbol rate of the code as 3/4 | 1/2. The default is 3/4. This property applies when the NumTransmitAntennas on page 3-0 property is greater than 2. For 2 transmit antennas, the symbol rate defaults to 1.

NumReceiveAntennas

Number of receive antennas

Specify the number of antennas at the receiver as a double-precision, real, scalar integer value from 1 to 8. The default is 1.

Fixed-Point Properties

RoundingMethod

Rounding of fixed-point numeric values

Specify the rounding method as Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero. The default is Floor.

OverflowAction

Action when fixed-point numeric values overflow

Specify the overflow action as one of Wrap | Saturate. The default is Wrap. This property specifies the action to be taken in case of overflow. Such overflow occurs if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result.

ProductDataType

Data type of product

Specify the product data type as one of Full precision | Custom. The default is Full precision.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as a scaled numerictype object with a signedness of Auto. The default is numerictype([],32,16). This property applies when you set the ProductDataType property to Custom.

AccumulatorDataType

Data type of accumulator

Specify the accumulator data type as Full precision | Same as product | Custom. The default is Full precision.

CustomAccumulatorDataType

Fixed-point data type of accumulator

Specify the accumulator fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,16)`. This property applies when you set the AccumulatorDataType property to `Custom`.

`EnergyProductDataType`

Data type of energy product

Specify the complex energy product data type as one of `Full precision` | `Same as product` | `Custom`. The default is `Full precision`. This property sets the data type of the complex product in the denominator to calculate the total energy in the MIMO channel.

`CustomEnergyProductDataType`

Fixed-point data type of energy product

Specify the energy product fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,16)`. This property applies when you set the EnergyProductDataType property to `Custom`.

`EnergyAccumulatorDataType`

Data type of energy accumulator

Specify the energy accumulator data type as one of `Full precision` | `Same as energy product` | `Same as accumulator` | `Custom`. The default is `Full precision`. This property sets the data type of the summation in the denominator to calculate the total energy in the MIMO channel.

`CustomEnergyAccumulatorDataType`

Fixed-point data type of energy accumulator

Specify the energy accumulator fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,16)`. This property applies when you set the EnergyAccumulatorDataType property to `Custom`.

`DivisionDataType`

Data type of division

Specify the division data type as one of `Same as accumulator` | `Custom`. The default is `Same as accumulator`. This property sets the data type at the output of the division operation. The setting normalizes diversity combining by the total energy in the MIMO channel.

`CustomDivisionDataType`

Fixed-point data type of division

Specify the division fixed-point type as a scaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32,16)`. This property applies when you set the DivisionDataType property to `Custom`.

Methods

step	Combine inputs using orthogonal space-time block code

Common to All System Objects	
release	Allow System object property value changes

Examples

Encode with OSTBC and Calculate Errors

Determine the bit error rate for a QSPK signal employing OSTBC encoding when transmitted through a 4x2 MIMO channel. Perfect channel estimation is assumed to be used by the OSTBC combiner.

Define the system parameters.

```
numTx = 4;         % Number of transmit antennas
numRx = 2;         % Number of receive antennas
Rs = 1e6;          % Sampling rate (Hz)
tau = [0 2e-6];    % Path delays (sec)
pdb = [0 -10];     % Average path gains (dB)
maxDopp = 30;      % Maximum Doppler shift (Hz)
numBits = 12000;   % Number of bits
SNR = 6;           % Signal-to-noise ratio (dB)
```

Set the random number generator to its default state to ensure repeatable results.

```
rng default
```

Create a QPSK modulator System object™. Set the `BitInput` property to `true` and the `SymbolMapping` property to `Gray`.

```
hMod = comm.QPSKModulator(...
    'BitInput',true,...
    'SymbolMapping','Gray');
```

Create a corresponding QPSK demodulator System object. Set the `SymbolMapping` property to `Gray` and the `BitOutput` property to `true`.

```
hDemod = comm.QPSKDemodulator(...
    'SymbolMapping','Gray',...
    'BitOutput',true);
```

Create an OSTBC encoder and combiner pair, where the number of antennas is specified in the system parameters.

```
hOSTBCEnc = comm.OSTBCEncoder(...
    'NumTransmitAntennas',numTx);

hOSTBCComb = comm.OSTBCCombiner(...
    'NumTransmitAntennas',numTx,...
    'NumReceiveAntennas',numRx);
```

Create a flat 4x2 MIMO Channel System object, where the channel characteristics are set using name-value pairs. The path gains are made available to serve as a perfect channel estimate for the OSTBC combiner.

```
hChan = comm.MIMOChannel(...
    'SampleRate',Rs,...
    'PathDelays',tau,...
    'AveragePathGains',pdb,...
    'MaximumDopplerShift',maxDopp,...
    'SpatialCorrelationSpecification','None',...
    'NumTransmitAntennas',numTx,...
    'NumReceiveAntennas',numRx,...
    'PathGainsOutputPort',true);
```

Create an AWGN channel System object in which the noise method is specified as a signal-to-noise ratio.

```
hAWGN = comm.AWGNChannel(...
    'NoiseMethod','Signal to noise ratio (SNR)',...
    'SNR',SNR,...
    'SignalPower',1);
```

Generate a random sequence of bits.

```
data = randi([0 1],numBits,1);
```

Apply QPSK modulation.

```
modData = step(hMod,data);
```

Encode the modulated data using the OSTBC encoder object.

```
encData = step(hOSTBCEnc,modData);
```

Transmit the encoded data through the MIMO channel and add white noise by using the `step` functions of the MIMO and AWGN channel objects, respectively.

```
[chanOut,pathGains] = step(hChan,encData);
rxSignal = step(hAWGN,chanOut);
```

Sum the `pathGains` array along the number of paths (2nd dimension) to form the channel estimate. Apply the `squeeze` function to make its dimensions conform with those of `rxSignal`.

```
chEst = squeeze(sum(pathGains,2));
```

Combine the received MIMO signal and its channel estimate using the `step` function of the OSTBC combiner object. Demodulate the combined signal.

```
combinedData = step(hOSTBCComb,rxSignal,chEst);
receivedData = step(hDemod,combinedData);
```

Compute the number of bit errors and the bit error rate.

```
[numErrors,ber] = biterr(data,receivedData)
```

```
numErrors = 11
```

```
ber = 9.1667e-04
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OSTBC Combiner block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.OSTBCEncoder`

Introduced in R2012a

step

System object: comm.OSTBCCombiner
Package: comm

Combine inputs using orthogonal space-time block code

Syntax

```
Y = step(H,X,CEST)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X,CEST) combines the received data, X, and the channel estimate, CEST, to extract the symbols encoded by an OSTBC. Both X and CEST are complex-valued and of the same data type, which can be double, single, or signed fixed point with power-of-two slope and zero bias. When the step method input X has double or single precision, the output, Y, has the same data type as the input. The input channel estimate can remain constant or can vary during each codeword block transmission. The combining algorithm uses the estimate only for the first symbol period per codeword block.

The time domain length, *T*/SymbolRate, must be a multiple of the codeword block length. *T* is the output symbol sequence length in the time domain. Specifically, when you set the NumTransmitAntennas property to 2, *T*/SymbolRate must be a multiple of two. When you set the NumTransmitAntennas property greater than 2, *T*/SymbolRate must be a multiple of four. For an input of *T*/SymbolRate rows by NumReceiveAntennas columns, the input channel estimate, CEST, must be a matrix of size *T*/SymbolRateby NumTransmitAntennas by NumReceiveAntennas. In this case, the extracted symbol data, Y, is a column vector with *T* elements. Input matrix size can be *F* by *T*/SymbolRate by NumReceiveAntennas, where *F* is an optional dimension (typically frequency domain) over which the combining calculation is independent. In this case, the input channel estimate, CEST, must be a matrix of size F by *T*/SymbolRate by NumTransmitAntennas by NumReceiveAntennas. The extracted symbol data, Y, is an *F* rows by *T* columns matrix.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.OSTBCEncoder

Package: comm

Encode input using orthogonal space-time block code

Description

The OSTBCEncoder object encodes an input symbol sequence using orthogonal space-time block code (OSTBC). The block maps the input symbols block-wise and concatenates the output codeword matrices in the time domain.

To encode an input symbol sequence using an orthogonal space-time block code:

1. Define and set up your OSTBC encoder object. See "Construction" on page 3-1051.
2. Call step to encode an input symbol sequence according to the properties of comm.OSTBCEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.OSTBCEncoder creates an orthogonal space-time block code (OSTBC) encoder System object, H. This object maps the input symbols block-wise and concatenates the output codeword matrices in the time domain.

H = comm.OSTBCEncoder(Name,Value) creates an OSTBC encoder object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.OSTBCEncoder(N,Name,Value) creates an OSTBC encoder object, H. This object has the NumTransmitAntennas property set to N, and the other specified properties set to the specified values.

Properties

NumTransmitAntennas

Number of transmit antennas

Specify the number of antennas at the transmitter as 2 | 3 | 4. The default is 2.

SymbolRate

Symbol rate of code

Specify the symbol rate of the code as one of `3/4 | 1/2`. The default is `3/4`. This property applies when you set the `NumTransmitAntennas` on page 3-0 property to greater than `2`. For `2` transmit antennas, the symbol rate defaults to `1`.

Fixed-Point Properties

`OverflowAction`

Action when fixed-point numeric values overflow

Specify the overflow action as one of `Wrap | Saturate`. The default is `Wrap`. This property specifies the action to be taken in the case of an overflow. Such overflow occurs when the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result.

Methods

step	Encode input using orthogonal space-time block code

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Encode BPSK Modulated Data with OSTBC

Generate random binary data, modulate using the BPSK modulation scheme, and encode the modulated data using OSTBC.

Generate an 8-by-1 vector of random binary data.

```
data = randi([0 1],8,1);
```

Create BPSK Modulator System object and modulated the data using the `step` function.

```
bpskMod = comm.BPSKModulator;
modData = bpskMod(data);
```

Create an OSTBC Encoder and encode the modulated signal. As the default number of transmit antennas is 2, you can see that `encData` is an 8-by-2 vector.

```
ostbcEnc = comm.OSTBCEncoder;
encData = ostbcEnc(modData)
```

```
encData = 8×2 complex

  -1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i  -1.0000 - 0.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i
  -1.0000 + 0.0000i   1.0000 + 0.0000i
  -1.0000 + 0.0000i  -1.0000 - 0.0000i
   1.0000 + 0.0000i  -1.0000 + 0.0000i
   1.0000 + 0.0000i   1.0000 + 0.0000i
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OSTBC Encoder block reference page. The object properties correspond to the block parameters.

When this object processes variable-size signals:

- If the input signal is a column vector, the first dimension can change, but the second dimension must remain fixed at `1`.
- If the input signal is a matrix, both dimensions can change.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.OSTBCCombiner`

Introduced in R2012a

step

System object: `comm.OSTBCEncoder`
Package: `comm`

Encode input using orthogonal space-time block code

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` encodes the input data, `X`, using OSTBC encoder object, `H`. The input is a complex-valued column vector or matrix of data type double, single, or signed fixed-point with power-of-two slope and zero bias. The step method output, `Y`, is the same data type as the input data. The time domain length, *T*, of `X` must be a multiple of the number of symbols in each codeword matrix. Specifically, when you set the NumTransmitAntennas property is `2` or the SymbolRate property is `1/2`, *T* must be a multiple of two and when the `SymbolRate` property to `3/4`, *T* must be a multiple of three. For a time or spatial domain input of *T* rows by one column, the encoded output data, `Y`, is a (*T*/`SymbolRate`)-by-`NumTransmitAntennas` matrix. The input matrix size can be *F* rows by *T* columns, where *F* is the additional dimension (typically the frequency domain) over which the encoding calculation is independent. In this case, the output is an *F*-by-(*T*/`SymbolRate`)-by-`NumTransmitAntennas` matrix.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.OVSFCode

Package: `comm`

Generate OVSF code

Description

The `OVSFCode` object generates an orthogonal variable spreading factor (OVSF) code from a set of orthogonal codes. OVSF codes were first introduced for 3G communication systems. They are primarily used to preserve orthogonality between different channels in a communication system.

To generate an OVSF code:

1. Define and set up your OVSF code object. See “Construction” on page 3-1055.
2. Call `step` to generate an OVSF code according to the properties of `comm.OVSFCode`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.OVSFCode` creates an orthogonal variable spreading factor (OVSF) code generator System object, `H`. This object generates an OVSF code.

`H = comm.OVSFCode(Name,Value)` creates an OVSF code generator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`SpreadingFactor`

Length of generated code

Specify the length of the generated code as an integer scalar value with a power of two. The default is `64`.

`Index`

Index of code of interest

Specify the index of the desired code from the available set of codes that have the spreading factor specified in the `SpreadingFactor` on page 3-0 property. This property must be an integer scalar in the range `0` to `SpreadingFactor–1`. The default is `60`.

OVSF codes are defined as the rows of an *n*-by-*n* matrix, *Cn*, where *n* is the value specified in the `SpreadingFactor` property.

You can define the matrix *Cn* recursively as follows:
First, define *C*1 = [1].
Next, assume that *Cn* is defined and let *Cn*(*k*) denote the *k*-th row of *Cn*.
Then, *C*2*n* = [*Cn*(0) *Cn*(0); *Cn*(0) -*Cn*(0); ... ; *Cn*(*n*-1) *Cn*(*n*–1); *Cn*(*n*–1)–*Cn*(*n*–1)].
Cn is only defined for values of n that are a power of 2. Set this property to a value of *k* to choose the *k*-th row of the *C* matrix as the code of interest.

SamplesPerFrame

Number of output samples per frame

Specify the number of OVSF code samples that the `step` method outputs as a numeric, positive, integer scalar value. The default is `1`. If you set this property to a value of *M*, then the `step` method outputs *M* samples of an OVSF code of length *N*. *N* is the length of the OVSF code that you specify in the `SpreadingFactor` on page 3-0 property.

OutputDataType

Data type of output

Specify output data type as one of `double` | `int8`. The default is `double`.

Methods

reset	Reset states of OVSF code generator object
step	Generate OVSF code

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Generate 10 samples of an OVSF code with a spreading factor of 64.

```
    hOVSF = comm.OVSFCode('SamplesPerFrame', 10,'SpreadingFactor',64);
    seq = step(hOVSF)
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the OVSF Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

See Also

`comm.HadamardCode` | `comm.WalshCode`

Introduced in R2012a

reset

System object: `comm.OVSFCode`
Package: `comm`

Reset states of OVSF code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `OVSFCode` object, `H`.

step

System object: `comm.OVSFCode`
Package: `comm`

Generate OVSF code

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H)` outputs a frame of the OVSF code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.PAMDemodulator

Package: `comm`

(Not recommended) Demodulate using M-ary PAM method

Note `comm.PAMDemodulator` is not recommended. Use `pamdemod` instead.

Description

The `PAMDemodulator` object demodulates a signal that was modulated using M-ary pulse amplitude modulation. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using M-ary pulse amplitude modulation:

1. Define and set up your PAM demodulator object. See “Construction” on page 3-1059.
2. Call `step` to demodulate the signal according to the properties of `comm.PAMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.PAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the M-ary pulse amplitude modulation (M-PAM) method.

`H = comm.PAMDemodulator(Name,Value)` creates an M-PAM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.PAMDemodulator(M,Name,Value)` creates an M-PAM demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is `4`. When you set the `BitOutput` on page 3-0 property to `false`, this value must be even. When you set the `BitOutput` property to `true`, this value requires an integer power of two.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`.

When you set this property to `true` the `step` method outputs a column vector of bit values with length equal to **log2**(`ModulationOrder` on page 3-0) times the number of demodulated symbols.

When you set this property to `false`, the `step` method outputs a column vector, with length equal to the input data vector. This value contains integer symbol values between `0` and `ModulationOrder`–1.

`SymbolMapping`

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol as one of `Binary` | `Gray`. The default is `Gray`.

When you set this property to `Gray`, the object uses a Gray-encoded signal constellation.

When you set this property to `Binary`, the integer *m*, between 0 ≤ *m* ≤(`ModulationOrder`–1) maps to the complex value 2*m*-`ModulationOrder`+1.

`NormalizationMethod`

Constellation normalization method

Specify the method used to normalize the signal constellation as one of `Minimum distance between symbols` | `Average power` | `Peak power`. The default is `Minimum distance between symbols`.

`MinimumDistance`

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is `2`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Minimum distance between symbols`.

`AveragePower`

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Average power`.

`PeakPower`

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Peak power`.

`OutputDataType`

Data type of output

Specify the output data type as one of `Full precision` | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`. The default is `Full precision`.

When you set this property to `Full precision`, and the input data type is single or double precision, the output data has the same data type that of the input.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in `Smallest unsigned integer` or `Full precision` mode.

When the input data is of a fixed-point type, the output data type behaves as if you had set the `OutputDataType` on page 3-0 property to `Smallest unsigned integer`.

When you set the `BitOutput` on page 3-0 property to `true`, then `logical` data type becomes a valid option.

Fixed-Point Properties

`FullPrecisionOverride`

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set `FullPrecisionOverride` to `true`, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set `FullPrecisionOverride` to `false`, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects” on page 3-1063.

`DenormalizationFactorDataType`

Data type of denormalization factor

Specify the denormalization factor data type as one of `Same word length as input` | `Custom`. The default is `Same word length as input`.

`CustomDenormalizationFactorDataType`

Fixed-point data type of denormalization factor

Specify the denormalization factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DenormalizationFactorDataType` on page 3-0 property to `Custom`.

`ProductDataType`

Data type of product

Specify the product data type as one of `Full precision` | `Custom`. The default is `Full precision`. When you set this property to `Full precision` the object calculates the full-precision product word and fraction lengths. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `ProductDataType` on page 3-0 property to `Custom`.

ProductRoundingMethod

Rounding of fixed-point numeric value of product

Specify the product rounding method as one of `Ceiling` | `Convergent` | `Floor` | `Nearest` | `Round` | `Simplest` | `Zero`. The default is `Floor`. This property applies when the object is not in a full precision configuration

ProductOverflowAction

Action when fixed-point numeric value of product overflows

Specify the product overflow action as one of `Wrap` | `Saturate`. The default is `Wrap`. This property applies when the object is not in a full precision configuration.

SumDataType

Data type of sum

Specify the sum data type as one of `Full precision` | `Same as product` | `Custom`. The default is `Full precision`. When you set this property to `Full precision`, the object calculates the full-precision sum word and fraction lengths. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false

CustomSumDataType

Fixed-point data type of sum

Specify the sum fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false and the `SumDataType` on page 3-0 property to `Custom`.

Methods

constellation	(Not recommended) Calculate or plot ideal signal constellation
step	(Not recommended) Demodulate using M-ary PAM method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate and demodulate a signal using 16-PAM modulation.

```
    hMod = comm.PAMModulator(16);
    hAWGN = comm.AWGNChannel('NoiseMethod', ...
                    'Signal to noise ratio (SNR)', ...
                    'SNR',20, 'SignalPower', 85);
    hDemod = comm.PAMDemodulator(16);
    %Create an error rate calculator
    hError = comm.ErrorRate;
    for counter = 1:100
      % Transmit a 50-symbol frame
      data = randi([0 hMod.ModulationOrder-1],50,1);
      modSignal = step(hMod, data);
      noisySignal = step(hAWGN, modSignal);
      receivedData = step(hDemod, noisySignal);
      errorStats = step(hError, data, receivedData);
    end
    fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

Compatibility Considerations

comm.PAMDemodulator is not recommended

comm.PAMDEmodulator is not recommended. Use pamdemod instead.

```
n = 10000; % Number of symbols to process
M = 8; % Modulation order
x = randi([0 M-1],n,1); % Create message signal.

%% Using PAM modulation and demodulation system objects
pammodObj = comm.PAMModulator(M);
pamdemodObj = comm.PAMDemodulator(M);
yOld = pammodObj(x); % Modulate.
% ... channel filtering ...
zOld = pamdemodObj(complex(y)); % Demodulate.

%% Using PAM modulation and demodulation functions
yNew = pammod(x,M); % Modulate.
% ... channel filtering ...
zNew = pamdemod(y,M); % Demodulate.
```

More About

Full Precision for Fixed-Point System Objects

FullPrecisionOverride is a convenience property that, when you set to true, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input.

For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Full precision for System objects does not optimize coefficient values. When you set the FullPrecisionOverride property to true, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set FullPrecisionOverride to false.

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`pamdemod` | `pammod`

Introduced in R2012a

constellation

System object: comm.PAMDemodulator
Package: comm

(Not recommended) Calculate or plot ideal signal constellation

Note comm.PAMDemodulator is not recommended. Use pamdemod and comm.ConstellationDiagram instead.

Syntax

```
y = constellation(h)
constellation(h)
```

Description

y = constellation(h) returns the numerical values of the constellation.

constellation(h) generates a constellation plot for the object.

Examples

Calculate Ideal PAM Signal Constellation

Create comm.PAMModulator and comm.PAMDemodulator System objects™, and then calculate their ideal signal constellations.

Create a modulator and demodulator objects.

```
mod = comm.PAMModulator;
demod = comm.PAMModulator;
```

Calculate the constellation points.

```
refMod = constellation(mod)
```

```
refMod = 4×1

    -3
    -1
     1
     3
```

```
refDemod = constellation(demod)
```

```
refDemod = 4×1

    -3
    -1
     1
     3
```

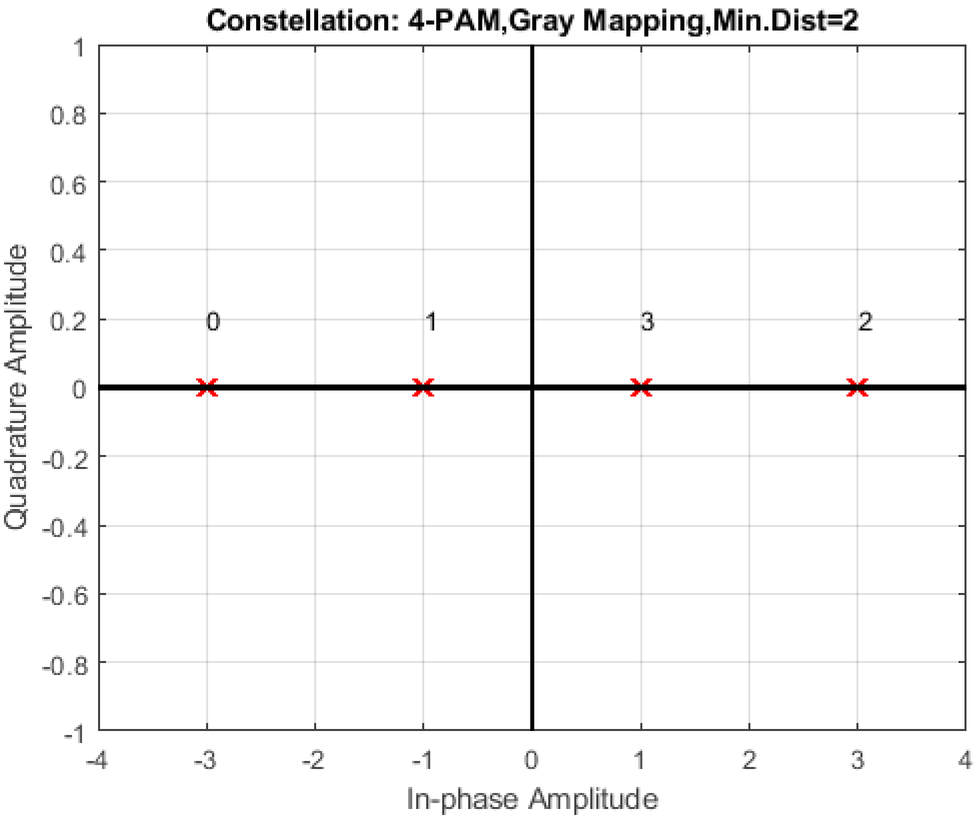
Verify that both objects produce the same points.

```
isequal(refMod,refDemod)
```

```
ans = logical
   1
```

Display the ideal signal constellation.

```
constellation(mod)
```



step

System object: `comm.PAMDemodulator`
Package: `comm`

(Not recommended) Demodulate using M-ary PAM method

Note `comm.PAMDemodulator` is not recommended. Use `pamdemod` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates data, `X`, with the M-PAM demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or column vector. The data type of the input can be double or single precision, signed integer, or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.PAMModulator

Package: comm

(Not recommended) Modulate using M-ary PAM method

Note comm.PAMModulator is not recommended. Use pammod instead.

Description

The PAMModulator object modulates using M-ary pulse amplitude modulation. The output is a baseband representation of the modulated signal. The M-ary number parameter, M, represents the number of points in the signal constellation and requires an even integer.

To modulate a signal using M-ary pulse amplitude modulation:

1. Define and set up your PAM modulator object. See “Construction” on page 3-1068.
2. Call step to modulate the signal according to the properties of comm.PAMModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.PAMModulator creates a modulator System object, H. This object modulates the input signal using the M-ary pulse amplitude modulation (M-PAM) method.

H = comm.PAMModulator(Name,Value) creates an M-PAM modulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.PAMModulator(M,Name,Value) creates an M-PAM modulator object, H. This object has the ModulationOrder property set to M and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is 4. When you set the BitInput on page 3-0 property to false, ModulationOrder must be even. When you set the BitInput property to true, ModulationOrder must be an integer power of two.

BitInput

Assume bit inputs

Specify whether the input is in bits or integers. The default is false.

When you set this property to true, the step method input requires a column vector of bit values whose length is an integer multiple of **log2**(ModulationOrder on page 3-0). This vector contains bit representations of integers between 0 and ModulationOrder–1.

When you set this property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder–1.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder on page 3-0) input bits to the corresponding symbol as one of Binary | Gray. The default is Gray.

When you set this property to Gray, the object uses a Gray-encoded signal constellation.

When you set this property to Binary, the input integer *m*, between 0 ≤ *m* ≤ModulationOrder-1) maps to the complex value 2*m*– ModulationOrder + 1.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as one of Minimum distance between symbols | Average power | Peak power. The default is Minimum distance between symbols.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is 2. This property applies when you set the NormalizationMethod on page 3-0 property to Minimum distance between symbols.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is 1. This property applies when you set the NormalizationMethod on page 3-0 property to Average power.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is 1. This property applies when you set the NormalizationMethod on page 3-0 property to Peak power.

OutputDataType

Data type of output

Specify the output data type as one of double | single | Custom. The default is double.

Fixed-Point Properties

CustomOutputDataType

Fixed-point data type of output

Specify the output fixed-point type as a numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the OutputDataType on page 3-0 property to Custom.

Methods

constellation	(Not recommended) Calculate or plot ideal signal constellation
step	(Not recommended) Modulate using M-ary PAM method

Common to All System Objects	
release	Allow System object property value changes

Examples

Modulate data using 16-PAM modulation, and visualize the data in a scatter plot.

```
% Create binary data for 100, 4-bit symbols
data = randi([0 1],400,1);
% Create a 16-PAM modulator System object with bits as inputs and
% Gray-coded signal constellation
hModulator = comm.PAMModulator(16,'BitInput',true);
% Modulate and plot the data
modData = step(hModulator, data);
constellation(hModulator)
```

Compatibility Considerations

comm.PAMModulator is not recommended

comm.PAMModulator is not recommended. Use pammod instead.

```
n = 10000; % Number of symbols to process
M = 8; % Modulation order
x = randi([0 M-1],n,1); % Create message signal.

%% Using PAM modulation and demodulation system objects
pammodObj = comm.PAMModulator(M);
pamdemodObj = comm.PAMDemodulator(M);
yOld = pammodObj(x); % Modulate.
% ... channel filtering ...
zOld = pamdemodObj(complex(y)); % Demodulate.
```

```
%% Using PAM modulation and demodulation functions
yNew = pammod(x,M); % Modulate.
% ... channel filtering ...
zNew = pamdemod(y,M); % Demodulate.
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

pamdemod | pammod

Introduced in R2012a

constellation

System object: comm.PAMModulator
Package: comm

(Not recommended) Calculate or plot ideal signal constellation

Note comm.PAMModulator is not recommended. Use pammod and comm.ConstellationDiagram instead.

Syntax

```
y = constellation(h)
constellation(h)
```

Description

y = constellation(h) returns the numerical values of the constellation.

constellation(h) generates a constellation plot for the object.

Examples

Calculate Ideal PAM Signal Constellation

Create comm.PAMModulator and comm.PAMDemodulator System objects™, and then calculate their ideal signal constellations.

Create a modulator and demodulator objects.

```
mod = comm.PAMModulator;
demod = comm.PAMModulator;
```

Calculate the constellation points.

```
refMod = constellation(mod)
```

```
refMod = 4×1

    -3
    -1
     1
     3
```

```
refDemod = constellation(demod)
```

```
refDemod = 4×1

    -3
    -1
     1
     3
```

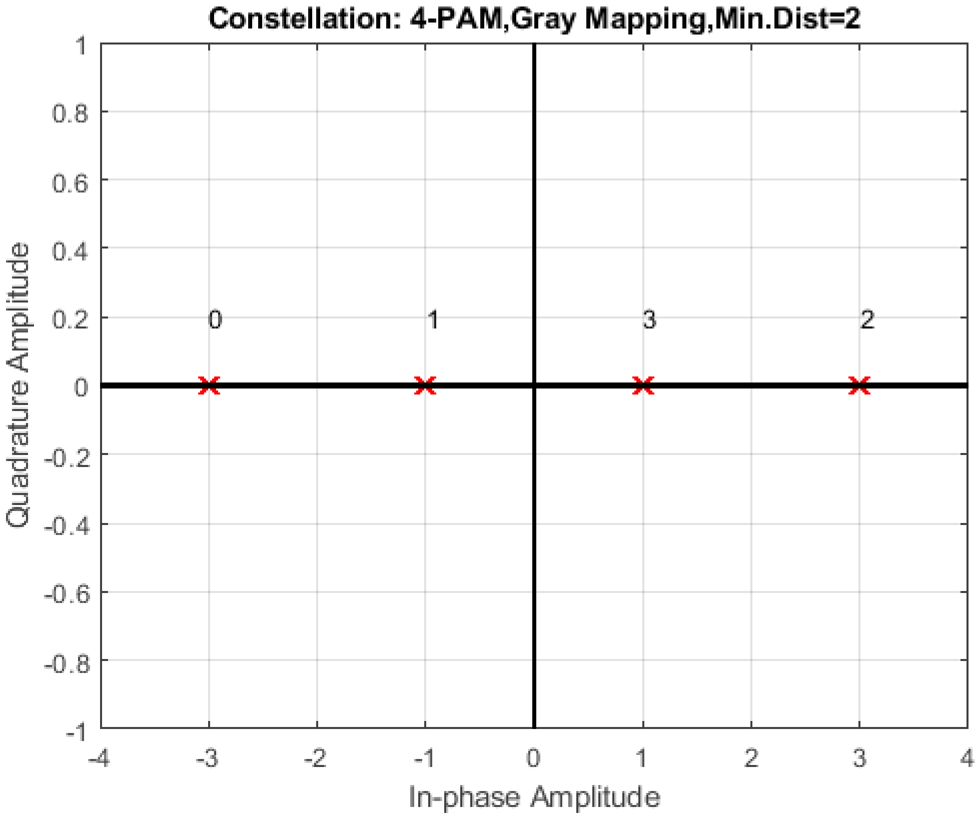
Verify that both objects produce the same points.

```
isequal(refMod,refDemod)
```

```
ans = logical
   1
```

Display the ideal signal constellation.

```
constellation(mod)
```



step

System object: comm.PAMModulator
Package: comm

(Not recommended) Modulate using M-ary PAM method

Note comm.PAMModulator is not recommended. Use pammod instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) modulates input data, X, with the M-PAM modulator System object, H. It returns the baseband modulated output, Y. Depending on the value of the BitInput property, input X can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.PhaseFrequencyOffset

Package: `comm`

Apply phase and frequency offsets to input signal

Description

The `PhaseFrequencyOffset` object applies phase and frequency offsets to an incoming signal.

To apply phase and frequency offsets to the input signal:

1. Define and set up your phase frequency offset object. See “Construction” on page 3-1075.
2. Call `step` to apply phase and frequency offsets to the input signal according to the properties of `comm.PhaseFrequencyOffset`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.PhaseFrequencyOffset` creates a phase and frequency offset System object, `H`. This object applies phase and frequency offsets to an input signal.

`H = comm.PhaseFrequencyOffset(Name,Value)` creates a phase and frequency offset object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`PhaseOffset`

Phase offset

Specify the phase offset in degrees. The default is `0`. If the `step` method input is an *M*-by-*N* matrix, the `PhaseOffset` on page 3-0 property can be set to a numeric scalar, an *M*-by-1, or 1-by-*N* numeric vector, or an *M*-by-*N* numeric matrix.

When you set the `PhaseOffset` property to a scalar value, the object applies the constant specified phase offset to each column of the input matrix.

When you set this property to an *M*-by-1 vector, the object applies time varying phase offsets, specified in the vector of this property, to each column of the input to the `step` method.

When you set this property to a 1-by-*N* vector, the object applies the *i*-th constant phase offset of this property to the *i*-th column of the input to the `step` method.

When you set this property to an *M*-by-*N* matrix, the object applies the *i*-th time varying phase offsets, specified in the *i*-th column of this property, to the *i*-th column of the input to the `step` method. This property is tunable.

`FrequencyOffsetSource`

Source of frequency offset

Specify the source of the frequency offset as one of `Property` | `Input port`. The default is `Property`. If you set this property to `Property`, you can specify the frequency offset using the `FrequencyOffset` on page 3-0 property. If you set this property to `Input port`, you specify the frequency offset as a step method input.

`FrequencyOffset`

Frequency offset

Specify the frequency offset in Hertz. The default is `0`. If the `step` method input is an *M*-by-*N* matrix, then the `FrequencyOffset` on page 3-0 property is a numeric scalar, an *M*-by-1, or 1-by-*N* numeric vector, or an *M*-by-*N* numeric matrix.

This property applies when you set the `FrequencyOffsetSource` on page 3-0 property to `Property`.

When you set this property to a scalar value, the object applies the constant specified frequency offset to each column of the input to the `step` method.

When you set this property to an *M*-by-1 vector, the object applies time-varying frequency offsets. These offsets are specified in the property, to each column of the input to the `step` method.

When you set this property to a 1-by-*N* vector, the object applies the *i*-th constant frequency offset in this property to the *i*-th column of the input to the `step` method.

When you set this property to an *M*-by-*N* matrix, the object applies the *i*-th time varying frequency offset. This offset is specified in the *i*-th column of this property and to the *i*-th column of input to the `step` method. This property is tunable.

`SampleRate`

Sample rate

Specify the sample rate of the input samples in seconds as a double-precision, real, positive scalar value. The default is `1`.

`SampleRate` = Input Vector Size / Simulink Sample Time

Methods

step	Apply phase and frequency offsets to input signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Introduce Phase Offset to 16-QAM Signal

Introduce a phase offset to a 16-QAM signal and view its effect on the constellation.

Create a phase frequency offset System object™. Set the phase offset to 30 degrees.

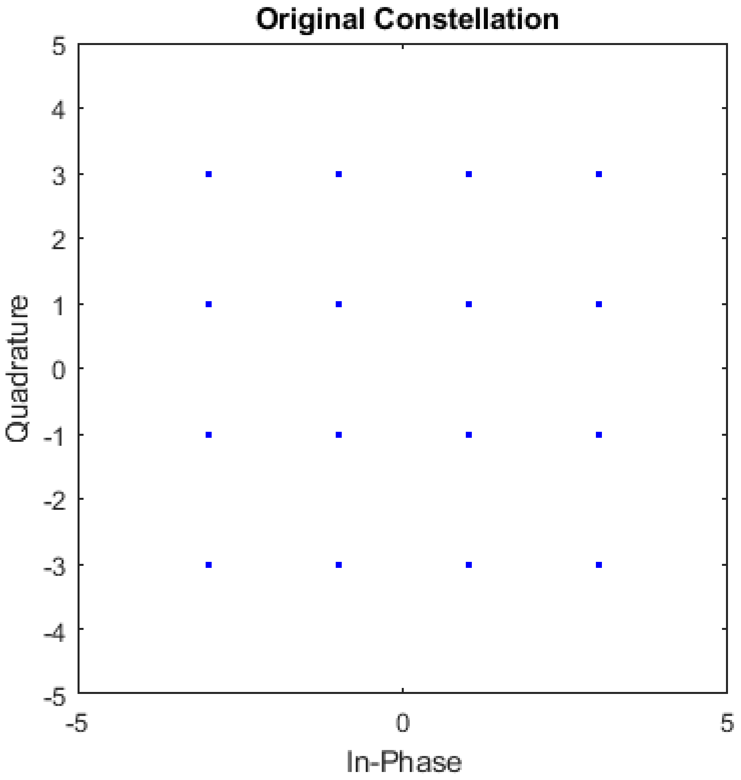
```
pfo = comm.PhaseFrequencyOffset('PhaseOffset',30);
```

Generate random symbols and apply 16-QAM modulation.

```
M = 16;
data = (0:M-1)';
modData = qammod(data,M);
```

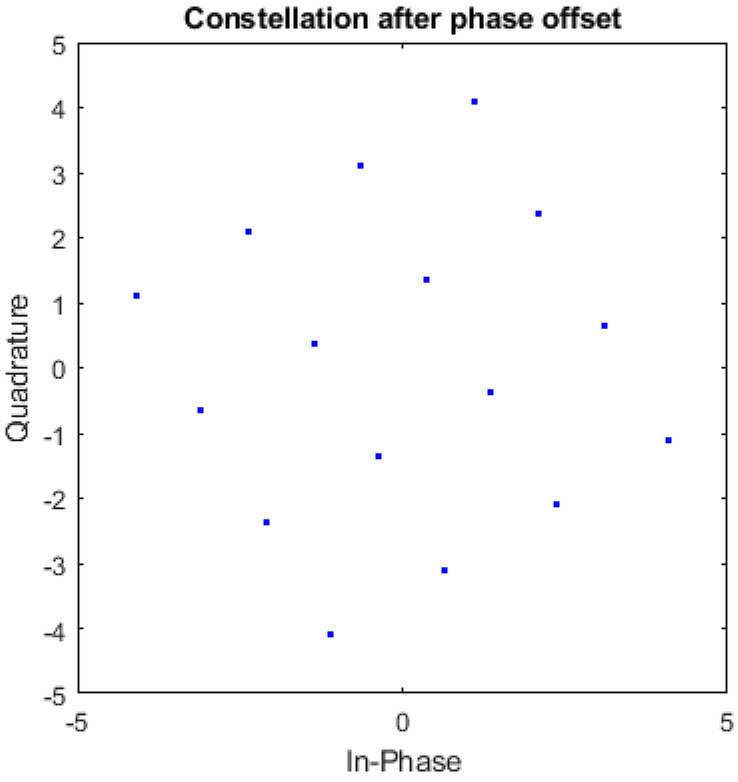
Plot the 16-QAM constellation.

```
scatterplot(modData);
title(' Original Constellation')
xlim([-5 5])
ylim([-5 5])
```



Introduce a phase offset using `pfo` and plot the offset constellation. Note that it has been shifted 30 degrees.

```
impairedData = pfo(modData);
scatterplot(impairedData);
title('Constellation after phase offset')
xlim([-5 5])
ylim([-5 5])
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the Phase/Frequency Offset block reference page. The object properties correspond to the block parameters, except: The object provides a `SampleRate` on page 3-0 property, which you must specify. The block senses the sample time of the signal and therefore does not have a corresponding parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

`comm.MemorylessNonlinearity` | `comm.PhaseNoise` | `comm.ThermalNoise`

Introduced in R2012a

step

System object: comm.PhaseFrequencyOffset
Package: comm

Apply phase and frequency offsets to input signal

Syntax

```
Y = step(H,X)
Y = step(H,X,FRQ)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) applies phase and frequency offsets to input X, and returns Y. The input X is a double or single precision matrix X, of dimensions MxN. M is the number of time samples in the input signals and N is number of channels. Both M and N can be equal to 1. The object adds phase and frequency offsets independently to each column of X. The data type and dimensions of X and Y are the same.

Y = step(H,X,FRQ) uses FRQ as the frequency offset that the object applies to input X when you set the FrequencyOffsetSource property to 'Input port'. When the X input is an MxN matrix, the value for FRQ can be a numeric scalar, an Mx1 or 1xN numeric vector, or an MxN numeric matrix. When the FRQ input is a scalar, the object applies a constant frequency offset, FRQ, to each column of X. When the FRQ input is an Mx1 vector, the object applies time varying frequency offsets, which are specified in the FRQ vector, to each column of X. When the FRQ input is a 1xN vector, the object applies the ith constant frequency offset in FRQ to the ith column of X. When the FRQ input is an MxN matrix, the object applies the ith time varying frequency offsets, specified in the ith column of FRQ, to the ith column of X.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.PhaseNoise

Package: `comm`

Apply phase noise to baseband signal

Description

The `comm.PhaseNoise` System object adds phase noise to a complex signal. This object emulates impairments introduced by the local oscillator of a wireless communication transmitter or receiver. The object generates filtered phase noise according to the specified spectral mask and adds it to the input signal. For a description of the phase noise modeling, see “Algorithms” on page 3-1087.

To add phase noise using a `comm.PhaseNoise` object:

1. Create the `comm.PhaseNoise` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
phznoise = comm.PhaseNoise
phznoise = comm.PhaseNoise(Name,Value)
phznoise = comm.PhaseNoise(level,offset,samplerate)
```

Description

`phznoise = comm.PhaseNoise` creates a phase noise System object with default property values.

`phznoise = comm.PhaseNoise(Name,Value)` creates a phase noise object with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`phznoise = comm.PhaseNoise(level,offset,samplerate)` creates a phase noise object with the phase noise level, frequency offset, and sample rate properties specified as value-only arguments. When specifying a value-only argument, you must specify all preceding value-only arguments.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Level` — Phase noise level

`[-60 -80]` (default) | vector of negative scalars

Phase noise level in decibels relative to carrier per hertz (dBc/Hz), specified as a vector of negative scalars. The `Level` and `FrequencyOffset` properties must have the same length.

Data Types: `double`

`FrequencyOffset` — Frequency offset

`[20 200]` (default) | vector of positive increasing values

Frequency offset in Hz, specified as a vector of positive increasing values. The `Level` and `FrequencyOffset` properties must have the same length.

Data Types: `double`

`SampleRate` — Sample rate

`1024` (default) | positive scalar

Sample rate in samples per second, specified as a positive scalar. To avoid aliasing, the sample rate must be greater than twice the largest value specified by `FrequencyOffset`.

Data Types: `double`

`RandomStream` — Source of random stream

`'Global stream'` (default) | `'mt19937ar with seed'`

Source of the random stream, specified as `'Global stream'` or `'mt19937ar with seed'`.If `RandomStream` is set to `'mt19937ar with seed'`, the mt19937ar algorithm is used for normally distributed random number generation, in which case the reset method reinitializes the random number stream to the value of the `Seed` property.

Data Types: `char` | `string`

`Seed` — Initial seed

`2137` (default) | positive scalar less than 2^{32}

Initial seed for `RandomStream`, specified as a positive scalar less than 2^{32}.

Dependencies

To enable this property, set `RandomStream` to `'mt19937ar with seed'`.

Data Types: `double`

Usage

Syntax

```
out = phznoise(in)
```

Description

`out = phznoise(in)` adds phase noise, specified by the `phznoise` System object, to the input signal. The result is returned in `out`.

Input Arguments

in — Input signal
complex column vector

Input signal, specified as an N_S-by-1 vector of complex values. N_S is the number of samples.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

out — Output signal
column vector

Output signal, returned as an N_S-by-1 vector of complex values. N_S equals the number of samples in the input signal.

Data Types: `double`
Complex Number Support: Yes

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to comm.PhaseNoise

visualize	Visualize spectrum mask of phase noise

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Phase Noise Effects on 16-QAM Signal

Add a phase noise vector and frequency offset vector to a 16-QAM signal. Then plot the signal.

Create a phase noise System object.

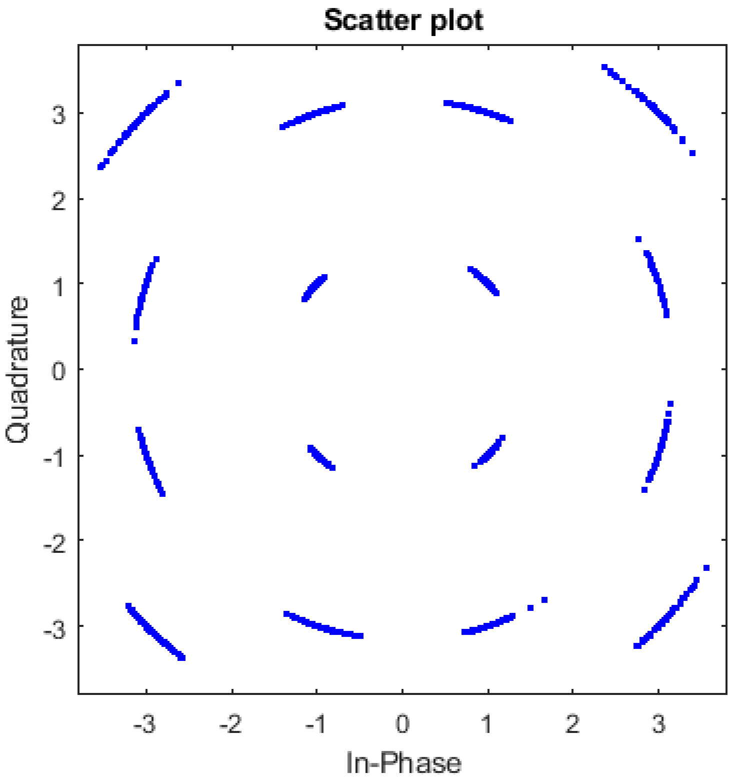
```
pnoise = comm.PhaseNoise('Level',-50,'FrequencyOffset',20);
```

Generate modulated symbols.

```
M = 16; % From 16-QAM
data = randi([0 M-1],1000,1);
modData = qammod(data,M);
```

Use `pnoise` to apply phase noise. Plot the impaired data.

```
y = pnoise(modData);
scatterplot(y)
```



View Phase Noise Effects on Signal Spectrum

View the effects of phase noise on a 10 MHz sine wave by using a spectrum analyzer. Adjust the resolution bandwidth of the spectrum analyzer to see its impact on the visualized spectral noise.

Initialize variables for the simulation.

```
fc = 1e6; % Carrier frequency in Hz
fs = 4e6; % Sample rate in Hz.
phNzLevel = [-85 -118 -125 -145]; % Phase noise level in dBc/Hz
phNzFreqOff = [1e3 9.5e3 19.5e3 195e3]; % Phase noise frequency offset in Hz
Nspf = 6e6; % Number of Samples per frame
freqSpan = 400e3; % Frequency span in Hz for spectrum computation
```

Create sine wave, phase noise, and spectrum analyzer objects.

```
sinewave = dsp.SineWave('Amplitude',1,'Frequency',fc,'SampleRate',fs, ...
    'SamplesPerFrame',Nspf,'ComplexOutput',true);
pnoise = comm.PhaseNoise('Level',phNzLevel, ...
    'FrequencyOffset',phNzFreqOff,'SampleRate',fs);
```

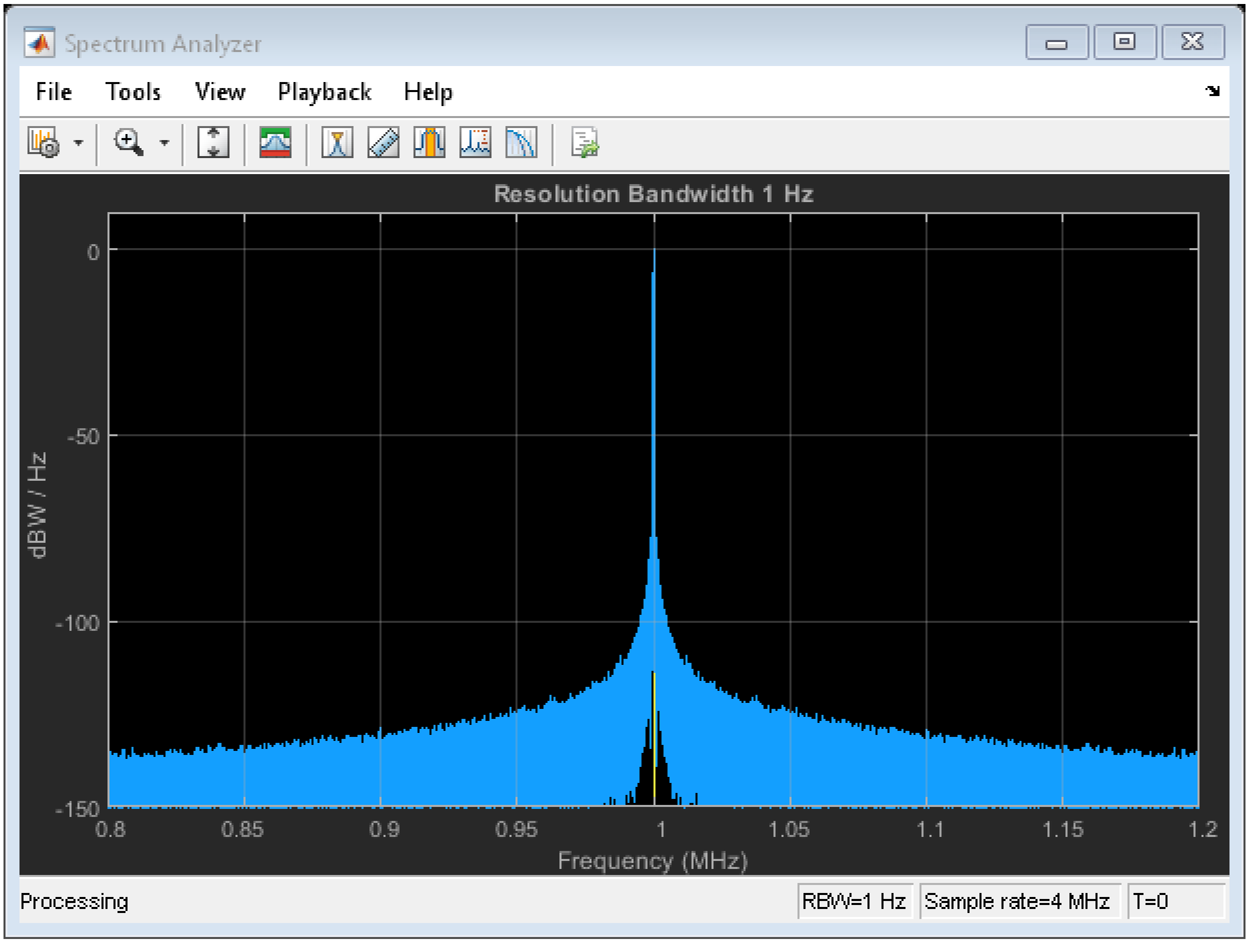
```
spectrumscopeRBW1 = dsp.SpectrumAnalyzer('NumInputPorts',2, ...
    'SampleRate',fs,'FrequencySpan','Span and center frequency', ...
    'CenterFrequency',fc,'Span',freqSpan,'RBWSource','Property', ...
    'RBW',1,'SpectrumType','Power density','SpectralAverages',10, ...
    'SpectrumUnits','dBW','YLimits',[-150 10], ...
    'Title','Resolution Bandwidth 1 Hz','Position',[79 147 605 374]);
spectrumscopeRBW10 = dsp.SpectrumAnalyzer('NumInputPorts',2, ...
    'SampleRate',fs,'FrequencySpan','Span and center frequency', ...
    'CenterFrequency',fc,'Span',freqSpan,'RBWSource','Property', ...
    'RBW',10,'SpectrumType','Power density','SpectralAverages',10, ...
    'SpectrumUnits','dBW','YLimits',[-150 10], ...
    'Title','Resolution Bandwidth 10 Hz','Position',[685 146 605 376]);
```

To analyze the spectrum and phase noise, the example includes two spectrum analyzer objects, with 1 Hz and 10 Hz resolution bandwidths, respectively. The spectrum analyzer objects use the default `Hann` windowing setting, the units are set to `dBW/Hz`, and the number of spectral averages is set to `10`.

```
x = sinewave();
y = pnoise(x);
```

When the resolution bandwidth is 1 Hz, the `dBW/Hz` view for the spectrum analyzer shows the tone at 0 dBW/Hz. The spectrum analyzer object corrects for the power spreading effect of the Hann windowing. Results show the visual average of the phase noise match the specified phase noise spectrum.

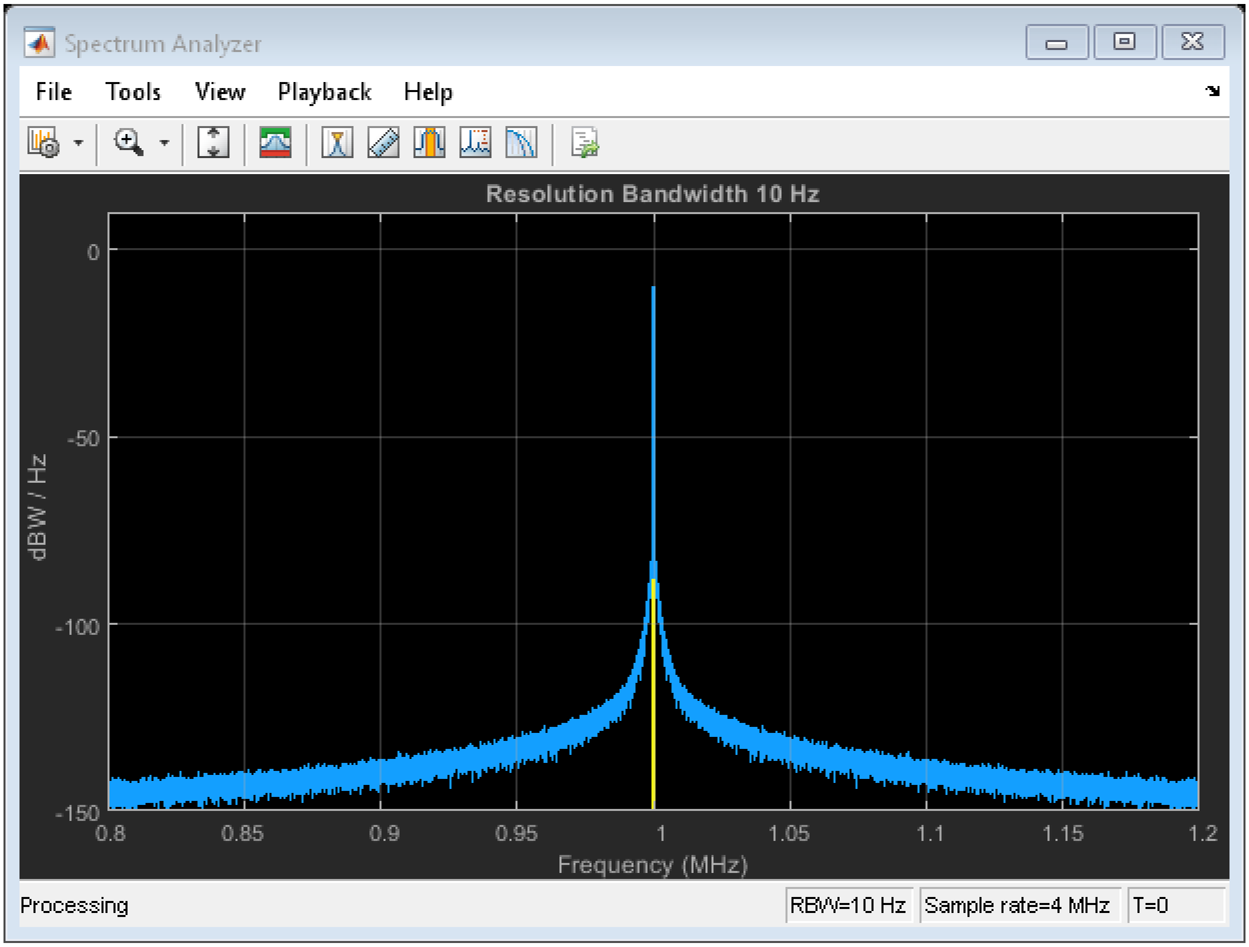
```
spectrumscopeRBW1(x,y)
```



When the resolution bandwidth is 10 Hz, the `dBW/Hz` view for the spectrum analyzer shows the tone at -10 dBW/Hz. The tone energy of the sine wave is now spread across 10 Hz instead of 1 Hz, so the sine wave PSD level reduces by 10 dB. With the resolution bandwidth at 10 Hz, the visual average of the phase noise still achieves the phase noise defined by the phase noise object.

With the resolution bandwidth increased from 1 Hz to 10 Hz, the spectrum analyzer object still corrects for the power spreading effect of the Hann window, and it achieves better spectral averaging with the wider resolution bandwidth. For more information, see “Why Use Windows?”.

```
spectrumscopeRBW10(x,y)
```



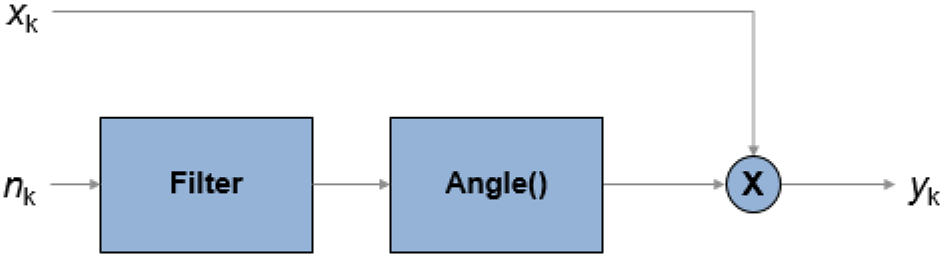
Calculate the RMS phase noise in degrees between the pure and noisy sine waves. In the general case, the pure signal must be time aligned with the noisy signal to accurately determine the phase error. However, in this case, the periodicity of the sine wave makes this step unnecessary.

```
ph_err = unwrap(angle(y) - angle(x));
rms_ph_nz_deg = rms(ph_err)*180/pi();
sprintf('The computed RMS phase noise is %3.2f degrees.\n',rms_ph_nz_deg)
```

```
ans = 
    'The computed RMS phase noise is 0.18 degrees.
     '
```

Algorithms

The output signal, y_k, is related to input sequence x_k by $y_k=x_k e^{j\varphi_k}$, where φ_k is the phase noise. The phase noise is filtered Gaussian noise such that $\varphi_k=f(n_k)$, where n_k is the noise sequence and f represents a filtering operation.



To model the phase noise, define the power spectrum density (PSD) mask characteristic by specifying scalar or vector values for the frequency offset and phase noise level.

- For a scalar frequency offset and phase noise level specification, an IIR digital filter computes the spectrum mask. The spectrum mask has a 1/*f* characteristic that passes through the specified point.
- For a vector frequency offset and phase noise level specification, an FIR filter computes the spectrum mask. The spectrum mask is interpolated across log10(*f*). It is flat from DC to the lowest frequency offset, and from the highest frequency offset to half the sample rate.

IIR Digital Filter

For the IIR digital filter, the numerator coefficient is

$$\lambda = \sqrt{2\pi f_{offset} 10^{L/10}}\,,$$

where f_{offset} is the frequency offset in Hz and L is the phase noise level in dBc/Hz. The denominator coefficients, γ_{i}, are recursively determined as

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1}\,,$$

where $\gamma_1 = 1$, $i = \{1, 2,..., N_{\text{t}}\}$, and N_{t} is the number of filter coefficients. N_{t} is a power of 2, from 2^7 to 2^{19}. The value of N_{t} grows as the phase noise offset decreases towards 0 Hz.

FIR Filter

For the FIR filter, the phase noise level is determined through log10(*f*) interpolation for frequency offsets over the range [df, f_{s} / 2], where df is the frequency resolution and f_{s} is the sample rate. The phase noise is flat from 0 Hz to the smallest frequency offset, and from the largest frequency offset to f_{s} / 2. The frequency resolution is equal to $\frac{f_s}{2}\left(\frac{1}{N_t}\right)$, where N_{t} is the number of coefficients, and is a power of 2 less than or equal to 2^{16}. If $N_{\text{t}} < 2^8$, a time domain FIR filter is used. Otherwise, a frequency domain FIR filter is used.

The algorithm increases N_{t} until these conditions are met:

- The frequency resolution is less than the minimum value of the frequency offset vector.
- The frequency resolution is less than the minimum difference between two consecutive frequencies in the frequency offset vector.

- The maximum number of FIR filter taps is 2^{16}.

References

[1] Kasdin, N. J., "Discrete Simulation of Colored Noise and Stochastic Processes and 1/(f^alpha); Power Law Noise Generation." *The Proceedings of the IEEE*. Vol. 83, No. 5, May, 1995, pp 802–827.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.MemorylessNonlinearity` | `comm.PhaseFrequencyOffset`

Blocks

Phase Noise

Introduced in R2012a

comm.PNSequence

Package: `comm`

Generate a pseudo-noise (PN) sequence

Description

The `PNSequence` object generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR). This object implements LFSR using a simple shift register generator (SSRG, or Fibonacci) configuration. Pseudo-noise sequences are typically used for pseudorandom scrambling and in direct-sequence spread-spectrum systems.

To generate a PN sequence:

1. Create the `comm.PNSequence` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
pnSequence = comm.PNSequence
pnSequence = comm.PNSequence(Name,Value)
```

Description

`pnSequence = comm.PNSequence` creates a pseudo-noise (PN) sequence generator System object. This object generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR).

`pnSequence = comm.PNSequence(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`Polynomial` — Generator polynomial
`'z^6 + z + 1'` (default) | polynomial character vector | binary row vector | integer vector

Generator polynomial that determines the feedback connections of the shift register , specified as one of these:

- A polynomial character vector that includes the number `1`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The first and last entries must be `1`. The length of this vector is $N + 1$, where N is the degree of the generator polynomial.
- A vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For more information, see "Character Representation of Polynomials".

Example: `'z^8 + z^2 + 1'`, `[1 0 0 0 0 0 1 0 1]`, and `[8 2 0]` represent the same polynomial, $p(z) = z^8 + z^2 + 1$.

Data Types: `double` | `char`

`InitialConditionsSource` — Source of initial conditions

`'Property'` (default) | `'Input port'`

Source of the initial conditions that determines the start of the PN sequence, specified as one of these:

- `'Property'` — Specify PN sequence generator initial conditions as a binary scalar or binary vector using the `InitialConditions` property.
- `'Input port'` — Specify PN sequence generator initial conditions by using an additional input argument, when calling the object. The object accepts a binary scalar or binary vector input. The length of the input must equal the degree of the generator polynomial that the `Polynomial` property specifies.

Data Types: `char`

`InitialConditions` — Initial conditions of shift register

`[0 0 0 0 0 1]` (default) | binary scalar | binary vector

Initial conditions of shift register when the simulation starts, specified as a binary scalar or binary vector.

If you set this property to a vector, each element of the vector corresponds to the initial value of the corresponding cell in the shift register. The length of the vector must equal the degree of the generator polynomial. If you set this property to a scalar, the initial conditions of all the cells of the shift register are the specified scalar value.

The scalar, or at least one element of the specified vector, must be nonzero for the object to generate a nonzero sequence.

Dependencies

This property is available when `InitialConditionsSource` is set to `'Property'`.

Data Types: `double`

`MaskSource` — Source of mask to shift PN sequence

`'Property'` (default) | `'Input port'`

Source of the mask that determines the shift of the PN sequence, specified as one of these:

- `'Property'` — Specify the mask as an integer scalar or binary vector using the `Mask` property.
- `'Input port'` — Specify the mask by using an additional input argument, when calling the object. The mask can only be specified as a binary vector, whose length must equal the degree of the generator polynomial that the `Polynomial` property specifies.

Data Types: `char`

`Mask` — Mask to shift PN sequence

`0` (default) | integer scalar | binary vector

Mask that determines how the PN sequence is shifted from its starting point, specified as an integer scalar or a binary vector.

When you set this property to an integer scalar, the value is the length of the shift. A scalar shift can be positive or negative. When the PN sequence has a period of $N = 2^n - 1$, where n is the degree of the generator polynomial that the `Polynomial` property specifies, the object wraps shift values that are negative or greater than N.

When you set this property to a binary vector, its length must equal the degree of the generator polynomial that the `Polynomial` specifies. For more information, see “Shifting PN Sequence Starting Point” on page 3-1100.

The mask vector can be calculated using the `shift2mask` function.

Dependencies

This property is available when `MaskSource` is set to `'Property'`.

`VariableSizeOutput` — Enable variable-size outputs

`false` (default) | `true`

Set this property to `true` to enable variable-size outputs by using an additional input argument when calling the object. The enabled input specifies the output size of the PN sequence. The input value must be less than or equal to the value of the `MaximumOutputSize` property.

When you set this property to `false`, the `SamplesPerFrame` property specifies the number of output samples.

`MaximumOutputSize` — Maximum output size

`[10 1]` (default) | two-element row vector

Maximum output size, specified as a positive integer two-element row vector that denotes the maximum output size of the PN sequence. The second element of the vector must be `1`.

Example: `[10 1]` gives a `10`-by-`1` maximum sized output signal.

Dependencies

This property is available when `VariableSizeOutput` is set to `true`.

`SamplesPerFrame` — Number of samples output per frame

`1` (default) | positive integer

Number of samples output per frame by the PN sequence object, specified as a positive integer. If you set this property to a value of M, then the object outputs M samples of a PN sequence that has a period of $N = 2^n - 1$, where n represents the degree of the generator polynomial that the `Polynomial` specifies.

If you set the `BitPackedOutput` property to `false`, the samples are bits from the PN sequence. If you set the `BitPackedOutput` property to `true`, then the output corresponds to `SamplesPerFrame` groups of bit-packed samples.

`ResetInputPort` — Enable generator reset input

`false` (default) | `true`

Set this property to `true` to enable PN sequence generator reset by using an additional input argument when calling the object. This input resets the states of the PN sequence generator to the initial conditions specified in the `InitialConditions` property. For more information, see "Resetting a Signal" on page 3-1100.

Dependencies

This property is available when `InitialConditionsSource` is set to `'Property'`.

`BitPackedOutput` — Output bit-packed words

`false` (default) | `true`

Option to output bit-packed words, specified as `false` or `true`. Set this property to `true` to enable bit-packed outputs.

When `BitPackedOutput` is `true`, the object outputs a column vector of length M, which contains most-significant-bit (MSB) first integer representations of bit words of length P. M is the number of samples per frame specified in the `SamplesPerFrame` property. P is the size of the bit-packed words specified in the `NumPackedBits` property.

Note The first bit from the left in the bit-packed word contains the most significant bit for the integer representation.

`NumPackedBits` — Number of bits per bit-packed word

8 (default) | integer in the range [1, 32]

Number of bits packed into each output data word, specified as an integer in the range [1, 32].

Dependencies

This property is available when `BitPackedOutput` is set to `true`.

`SignedOutput` — Output signed bit-packed words

`false` (default) | `true`

Set this property to `true` to obtain signed, bit-packed, output words. In this case, a `1` in the most significant bit (sign bit) indicates a negative value. The property indicates negative numbers in a two's complement format.

Dependencies

This property is available when `BitPackedOutput` is set to `true`.

`OutputDataType` — Data type of output

`'double'` (default) | `'logical'` | `'Smallest unsigned integer'` | `'Smallest integer'`

Output data type, specified as one of these:

- When BitPackedOutput is set to false, OutputDataType can be 'double', 'logical', or 'Smallest unsigned integer'.
- When BitPackedOutput is set to true, OutputDataType can be 'double' or 'Smallest integer'.

Note You must have a Fixed-Point Designer user license to use this property in 'Smallest unsigned integer' or 'Smallest integer' mode.

Dependencies

The valid settings for OutputDataType depends on the setting of BitPackedOutput.

Usage

Syntax

```
outSequence = pnSequence()
outSequence = pnSequence(initCond)
outSequence = pnSequence(maskVec)
outSequence = pnSequence(outputSize)
outSequence = pnSequence(reset)
outSequence = pnSequence(initCond,maskVec,outputSize)
outSequence = pnSequence(maskVec,outputSize,reset)
```

Description

outSequence = pnSequence() outputs a frame of the PN sequence. Specify the frame length with the SamplesPerFrame property. The PN sequence has a period of $N = 2^n - 1$, where n is the degree of the generator polynomial that you specify in the Polynomial property.

You can combine optional input arguments when you set their enabling properties. Optional inputs must be listed in the same order as the order of the enabling properties.

outSequence = pnSequence(initCond) provides an additional input with values specifying the initial conditions of the linear-feedback shift register.

This syntax applies when you set the InitialConditionsSource property of the object to 'Input port'.

outSequence = pnSequence(maskVec) provides an additional input specifying the mask vector that determines how the PN sequence is shifted from its starting point.

This syntax applies when you set the MaskSource property of the object to 'Input port'.

outSequence = pnSequence(outputSize) provides an additional input specifying the output size of the PN sequence.

This syntax applies when you set the VariableSizeOutput property of the object to true.

`outSequence = pnSequence(reset)` provides an additional input indicating whether to reset the PN sequence generator.

This syntax applies when you set `InitialConditionsSource` to `'Property'` and `ResetInputPort` to `true`.

`outSequence = pnSequence(initCond,maskVec,outputSize)`

`outSequence = pnSequence(maskVec,outputSize,reset)`

Using these syntaxes, you can combine optional input arguments when you set their enabling properties. Optional inputs must be listed in the same order as the order of the enabling properties.

Input Arguments

initCond — Initial register conditions

binary scalar | binary vector

Initial conditions of the shift register when the simulation starts, specified as a binary scalar or binary vector. When you set `initCond` to a binary vector, the length of the vector must equal the degree of the `Polynomial` property. The scalar, or at least one element of `initCond`, must be nonzero for the object to generate a nonzero sequence.

Example: `outSequence = pnSequence([1 1 0])` corresponds to possible initial register states for a PN sequence generator specified by a generator polynomial of degree 3.

Data Types: `double`

maskVec — Mask vector

binary vector

Mask that determines how the PN sequence is shifted from its starting point, specified as a binary vector. The length of the vector must equal the degree of the `Polynomial` property.

outputSize — Output size of the PN sequence

scalar | two-element row vector

Output size of the PN sequence, specified as a scalar or two-element row vector. When you set `outputSize` to a two-element row vector, the second element must be equal to `1`.

reset — Reset PN sequence generator

scalar | column vector

When you specify `reset` as a scalar, the object resets to the initial conditions specified in the `InitialConditions` property and generates a new output frame.

When you specify `reset` as a column vector, the length of the vector must equal the number of samples per frame specified in `SamplesPerFrame`. A column vector input allows multiple resets within an output frame. A non-zero value at the ith element of the vector will cause a reset at the ith element of the generated PN sequence.

Output Arguments

outSequence — PN Sequence

column vector

PN sequence generated by the object, returned as a column vector.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Generate Maximal Length PN Sequences

Generate a 14-sample frame of a maximal length PN sequence given generator polynomial, $x^3 + x^2 + 1$.

Generate PN sequence data by using the `comm.PNSequence` object. The sequence repeats itself as it contains 14 samples while the maximal sequence length is only 7 samples ($2^3 - 1$).

```
pnSequence = comm.PNSequence('Polynomial',[3 2 0], ...
    'SamplesPerFrame',14,'InitialConditions',[0 0 1]);
x1 = pnSequence();
[x1(1:7) x1(8:14)]
```

```
ans = 7×2

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

Create another maximal length sequence based on the generator polynomial, $x^4 + x + 1$. As it is a fourth order polynomial, the sequence repeats itself after 15 samples ($2^4 - 1$).

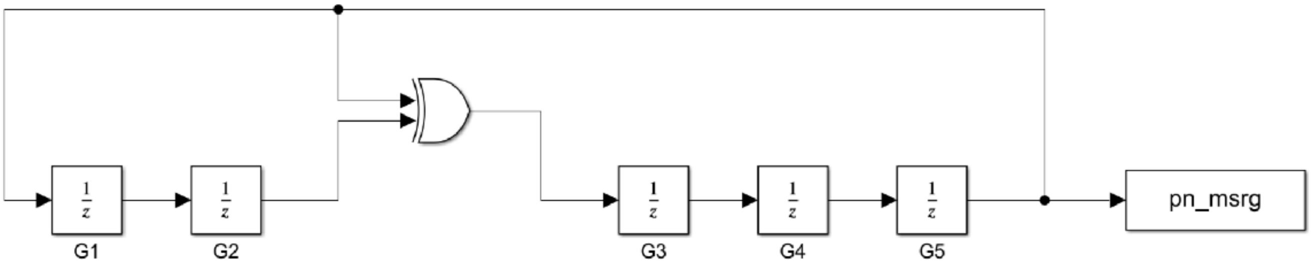
```
pnSequence2 = comm.PNSequence('Polynomial','x^4+x+1', ...
    'InitialConditions',[0 0 0 1],'SamplesPerFrame',30);
x2 = pnSequence2();
[x2(1:15) x2(16:30)]
```

```
ans = 15×2

     1     1
     0     0
     0     0
     0     0
     1     1
     0     0
     0     0
     1     1
     1     1
     0     0
       ⋮
```

Generate Galois Linear-Feedback Shift Register Output

The `comm.PNSequence` system object implements a linear-feedback shift register (LFSR) using a simple shift register generator (SSRG, or Fibonacci configuration). This configuration differs from the modular shift register generator (MSRG, or Galois configuration) by a phase difference, that can be determined empirically from the system object.

This phase difference can be specified as the `Mask` parameter for the `comm.PNSequence` system object to generate the equivalent MSRG configuration output. The block diagram represents the implementation of a 5-bit LFSR in the Galois (MSRG) configuration.



Load the file `GaloisLFSR`. The file contains the following variables that define the properties and output PN sequence of the 5-bit Galois LFSR:

- `polyVec`: Generator polynomial
- `polySize`: Degree of the generator polynomial
- `initStates`: Initial conditions of the shift register
- `maskVar`: Mask to shift the PN sequence
- `pn_msrg`: Output PN sequence of maximal length, from the 5-bit Galois LFSR

```
load GaloisLFSR
```

Generate PN sequence data by using the `comm.PNSequence` object with the same set of properties used to implement the 5-bit Galois LFSR. Compare this PN sequence with the output of the 5-bit Galois LFSR. The two sequences differ by a phase shift.

```
pnSequence = comm.PNSequence('Polynomial',polyVec,'InitialConditions',initStates,...
    'Mask',maskVar,'SamplesPerFrame',2^polySize-1);
```

```
pn = pnSequence();

isequal(pn,pn_msrg)

ans = logical
   0
```

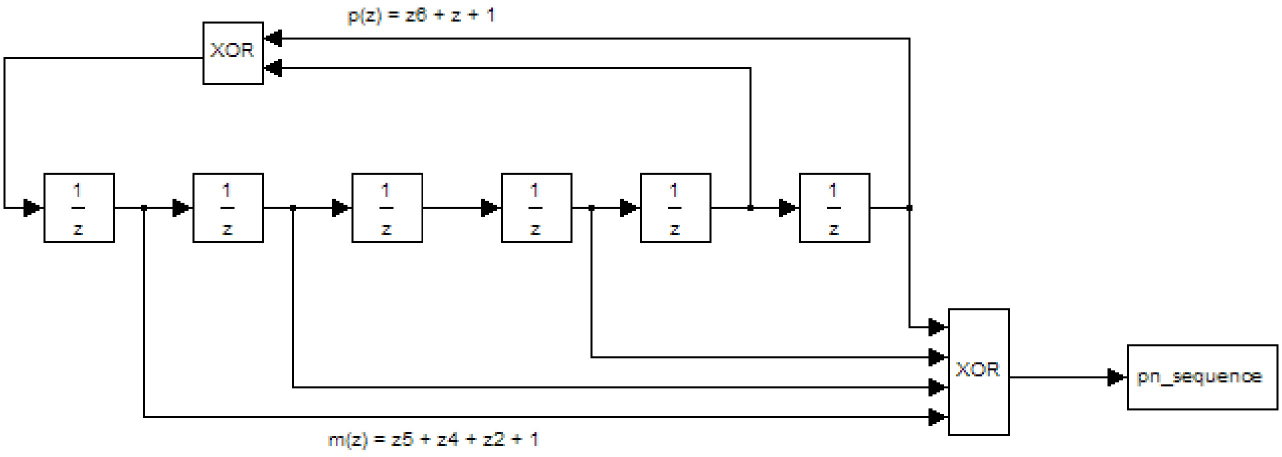
Compute the phase shift between the two configurations. Set the value of the `Mask` property based on this phase shift.

```
for i = 1:length(pn)
    exp_pn = [pn(i:end);pn(1:(i-1))];
    if isequal(exp_pn,pn_msrg)
        break
    end
end

maskVar = i-1;
```

Generate PN sequence data by using the `comm.PNSequence` system object with the modified `Mask` property value. Compare this sequence with the output of the 5-bit Galois LFSR. The two sequences are now equal.

```
pnSequence_mod = comm.PNSequence('Polynomial',polyVec,'InitialConditions',initStates,...
    'Mask',maskVar,'SamplesPerFrame',2^polySize-1);

pn_mod = pnSequence_mod();

isequal(pn_mod,pn_msrg)

ans = logical
   1
```

Setting up the PN sequence generator



This figure defines a PN sequence generator with a generator polynomial $p(z) = z^6 + z + 1$. You can set up the PN sequence generator by typing the following at the MATLAB command line.

```
h1 = comm.PNSequence('Polynomial', [1 0 0 0 0 1 1], 'InitialConditions', [1 1 0 1 0 1]);
h2 = comm.PNSequence('Polynomial', [1 0 0 0 0 1 1], 'Mask', 22);
mask2shift ([1 0 0 0 0 1 1],[1 1 0 1 0 1])
```

```
ans = 22
```

Alternatively you can input the polynomial exponents of z for the nonzero terms of the polynomial in descending order of powers.

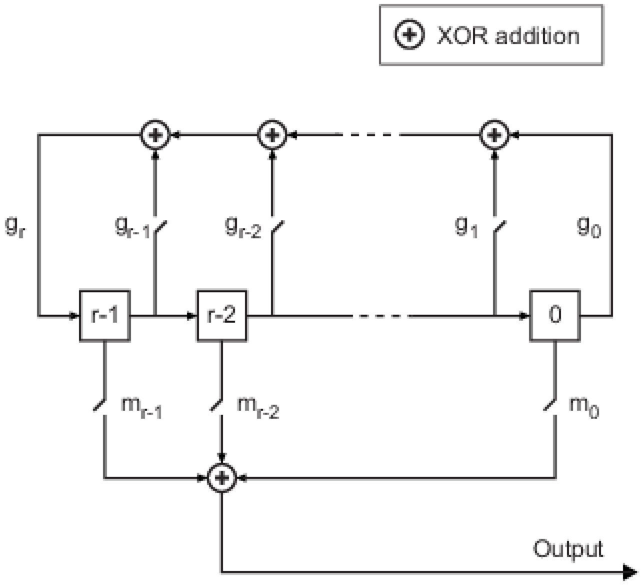
```
h = comm.PNSequence('Polynomial', [6 1 0], 'InitialConditions', [1 1 0 1 0 1])
```

```
h = 
  comm.PNSequence with properties:

                 Polynomial: [6 1 0]
    InitialConditionsSource: 'Property'
          InitialConditions: [1 1 0 1 0 1]
                 MaskSource: 'Property'
                       Mask: 0
         VariableSizeOutput: false
            SamplesPerFrame: 1
             ResetInputPort: false
            BitPackedOutput: false
             OutputDataType: 'double'
```

Algorithms

Simple Shift Register Generator

A linear-feedback shift register (LFSR), implemented as a simple shift register generator (SSRG), is used to generate PN sequences. This type of shift register is also known as a Fibonacci implementation.



The `Polynomial` property determines the feedback connections of the shift register. It is a primitive binary polynomial in *z*, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \ldots + g_0$. For the coefficient, $g_{k=0}$ to *r*, the coefficient g_k is 1 if there is a connection from the kth register to the adder. The leading term, g_r, and the constant term, g_0, of the `Polynomial` property must be `1` because the polynomial must be primitive. The `InitialConditions` property specifies the initial values of the registers. For example, the following table indicates two sets of parameter values that correspond to a generator polynomial of $p(z) = z^8 + z^2 + 1$.

Quantity	Example 1	Example 2
Polynomial	`g1 = [1 0 0 0 0 0 1 0 1]`	`g2 = [8 2 0]`
Degree of generator polynomial	8, which is `length(g1)-1`	8
InitialConditions	`[1 0 0 0 0 0 1 0]`	`[1 0 0 0 0 0 1 0]`

At each step, all *r* registers in the generator update their values according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The output of the LFSR reflects the sum of all connections in the *m* mask vector.

The `Mask` property, *m*, determines the shift of the PN sequence starting point. For more information, see “Shifting PN Sequence Starting Point” on page 3-1100.

Shifting PN Sequence Starting Point

To shift the starting point of the PN sequence, specify the `Mask` property as:

- An integer representing the length of the shift.

  The default `Mask` setting of `0` corresponds to no shift. As illustrated in the LFSR shift register diagram in “Simple Shift Register Generator” on page 3-1099, there is no shift when the only connection is along the arrow labeled m_0.

  This table shows the shift that occurs when you set `Mask` to `0` versus a positive integer *d*.

	T = 0	T = 1	T = 2	…	T = *d*	T = *d*+1
Shift = 0	x_0	x_1	x_2	…	x_d	x_{d+1}
Shift = *d*	x_d	x_{d+1}	x_{d+2}	…	x_{2d}	x_{2d+1}

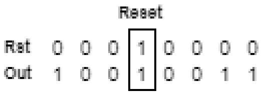
- A binary vector whose length is equal to the degree of the generator polynomial. The LFSR shift register diagram in “Simple Shift Register Generator” on page 3-1099 shows `Mask` specified as a mask vector, *m*. The binary vector must have *N* elements, where *N* is the degree of the generator polynomial. To calculate the mask vector, use the `shift2mask` function.

  The binary vector corresponds to a polynomial in *z*, $m_{r-1} z^{r-1} + m_{r-2} z^{r-2} + \ldots + m_1 z + m_0$, of degree at most $r - 1$. The mask vector that corresponds to a shift of *d* is the vector that represents $m(z) = z^d$ modulo $g(z)$, where $g(z)$ is the generator polynomial.

  For example, if the degree of the generator polynomial is 4, then the mask vector that corresponds to $d = 2$ is `[0 1 0 0]`, which represents the polynomial $m(z) = z^2$.

Resetting a Signal

To reset the PN generator sequence, you must first set the `ResetInputPort` property to `true`. Suppose that the system object generates a PN sequence of `[1 0 0 1 1 0 1 1]` when there is no reset. When the reset signal `[0 0 0 1]` is passed as an input argument to the object, the PN sequence is reset at the fourth bit, because the fourth bit of the reset signal is a `1`.



Rst	0	0	0	1	0	0	0	0
Out	1	0	0	1	0	0	1	1

Sequences of Maximum Length

To generate a maximum length sequence for a generator polynomial that has the degree *r*, set `Polynomial` to a value from the following table. The maximum sequence length is $2^r - 1$.

r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial
2	`[2 1 0]`	15	`[15 14 0]`	28	`[28 25 0]`	41	`[41 3 0]`
3	`[3 2 0]`	16	`[16 15 13 4 0]`	29	`[29 27 0]`	42	`[42 23 22 1 0]`
4	`[4 3 0]`	17	`[17 14 0]`	30	`[30 29 28 7 0]`	43	`[43 6 4 3 0]`
5	`[5 3 0]`	18	`[18 11 0]`	31	`[31 28 0]`	44	`[44 6 5 2 0]`
6	`[6 5 0]`	19	`[19 18 17 14 0]`	32	`[32 31 30 10 0]`	45	`[45 4 3 1 0]`
7	`[7 6 0]`	20	`[20 17 0]`	33	`[33 20 0]`	46	`[46 21 10 1 0]`
8	`[8 6 5 4 0]`	21	`[21 19 0]`	34	`[34 15 14 1 0]`	47	[47 14 0]
9	`[9 5 0]`	22	`[22 21 0]`	35	`[35 2 0]`	48	[48 28 27 1 0]
10	`[10 7 0]`	23	`[23 18 0]`	36	`[36 11 0]`	49	[49 9 0]
11	`[11 9 0]`	24	`[24 23 22 17 0]`	37	[37 12 10 2 0]	50	[50 4 3 2 0]
12	`[12 11 8 6 0]`	25	`[25 22 0]`	38	[38 6 5 1 0]	51	[51 6 3 1 0]
13	`[13 12 10 9 0]`	26	`[26 25 24 20 0]`	39	[39 8 0]	52	[52 3 0]
14	`[14 13 8 4 0]`	27	`[27 26 25 22 0]`	40	`[40 5 4 3 0]`	53	[53 6 2 1 0]

For more information about the shift register configurations that these polynomials represent, see *Digital Communications* by John Proakis.[1].

References

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

[2] Lee, J. S., and L. E. Miller. *CDMA Systems Engineering Handbook*. Boston and London. Artech House, 1998.

[3] Golomb, S.W. *Shift Register Sequences*. Laguna Hills. Aegean Park Press, 1967.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.GoldSequence` | `comm.KasamiSequence`

Blocks
PN Sequence Generator

Introduced in R2012a

comm.PreambleDetector

Package: `comm`

Detect preamble in data

Description

The `comm.PreambleDetector` System object detects a preamble in an input data sequence. A preamble is a set of symbols or bits used in packet-based communication systems to indicate the start of a packet. The preamble detector object finds the location corresponding to the end of the preamble.

To detect a preamble in an input data sequence:

1. Create a `comm.PreambleDetector` object and set the properties of the object.
2. Call `step` to detect the presence of a preamble.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`prbdet = comm.PreambleDetector` creates a preamble detector object, `prbdet`, using the default properties.

`prbdet = comm.PreambleDetector(Name,Value)` specifies additional properties using `Name,Value` pairs. Unspecified properties have default values.

`prbdet = comm.PreambleDetector(prb,Name,Value)` specifies the preamble, `prb` in addition to those properties specified by using `Name,Value` pairs.

Example:

```
prbdet = comm.PreambleDetector('Input','Bit','Detections','First');
```

Properties

`Input` — Type of input data

`'Symbol'` (default) | `'Bit'`

Type of input data, specified as `'Symbol'` or `'Bit'`. For binary inputs, set this parameter to `'Bit'`. For all other inputs, set this parameter to `'Symbol'`. Symbol data can be of data type `single` or `double` while bit data can, in addition, support the `Boolean`, `int8`, and `uint8` data types.

`Preamble` — Preamble sequence

`[1+1i; 1-1i]` (default) | column vector

Preamble sequence, specified as a real or complex column vector. The object uses this sequence to detect the presence of the preamble in the input data. If `Input` is `'Bit'`, the preamble must be a binary sequence. If `Input` is `'Symbol'`, the preamble can be any real or complex sequence.

Data Types: `double` | `single` | `logical` | `int8` | `uint8`

`Threshold` — Detection threshold

3 (default) | nonnegative scalar

Detection threshold, specified as a nonnegative scalar. When the computed detection metric is greater than or equal to `Threshold`, the preamble is detected. This property is available when `Input` is set to `'Symbol'`. Tunable.

`Detections` — Number of preambles to detect

`'All'` (default) | `'First'`

Number of preambles to detect, specified as `'All'` or `'First'`.

- `'All'` — Detects all the preambles in the input data sequence.
- `'First'` — Detect only the first preamble in the input data sequence.

Methods

reset	Reset states of preamble detector object
step	Detect preamble in data

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Detect Preamble from Binary Data Sequence

Specify a six-bit preamble.

```
prb = [1 0 0 1 0 1]';
```

Create a preamble detector object using preamble `prb` and taking bit inputs.

```
prbdet = comm.PreambleDetector(prb,'Input','Bit');
```

Generate a binary data sequence containing two preambles and using random bits to represent the data fields.

```
pkt = [prb; randi([0 1],10,1); prb; randi([0 1],10,1)];
```

Locate the indices of the two preambles. The indices correspond to the end of the preambles.

```
idx = prbdet(pkt)
```

```
idx = 2×1

     6
```

```
    22
```

The detector correctly identified indices 6 and 22 as the end of the two preambles inserted in the sequence.

Detect Preamble in Noisy QPSK Signal

Create a preamble and apply QPSK modulation.

```
p1 = [0 1 2 3 3 2 1 0]';
p = [p1; p1];
prb = pskmod(p,4,pi/4,'gray');
```

Create a `comm.PreambleDetector` object using preamble `prb`.

```
prbdet = comm.PreambleDetector(prb)
```

```
prbdet = 
  comm.PreambleDetector with properties:

         Input: 'Symbol'
      Preamble: [16x1 double]
     Threshold: 3
    Detections: 'All'
```

Generate a sequence of random symbols. The first sequence represents the last 20 symbols from a previous packet. The second sequence represents the symbols from the current packet.

```
d1 = randi([0 3],20,1);
d2 = randi([0 3],100,1);
```

Modulate the two sequences.

```
x1 = pskmod(d1,4,pi/4,'gray');
x2 = pskmod(d2,4,pi/4,'gray');
```

Create a sequence of modulated symbols consisting of the remnant of the previous packet, the preamble, and the current packet.

```
y = [x1; prb; x2];
```

Add white Gaussian noise.

```
z = awgn(y,10);
```

Determine the preamble index and the detection metric.

```
[idx,detmet] = prbdet(z);
```

Calculate the number of elements in `idx`. Because the number of elements is greater than one, the detection threshold is too low.

```
numel(idx)
```

```
ans = 80
```

Display the five largest detection metrics.

```
detmetSort = sort(detmet,'descend');
detmetSort(1:5)
```

```
ans = 5×1

   16.3115
   13.6900
   10.5698
    9.1920
    8.9706
```

Increase the threshold and determine the preamble index.

```
prbdet.Threshold = 15;
idx = prbdet(z)
```

```
idx = 36
```

The result of 36 corresponds to the sum of the preamble length (16) and the remaining samples in the previous packet (20). This indicates that the preamble has been successfully detected.

Algorithms

Bit Inputs

When the input data is composed of bits, the preamble detector uses an exact pattern match.

Symbol Inputs

When the input data is composed of symbols, the preamble detector uses a cross-correlation algorithm. A finite impulse response (FIR) filter, in which the coefficients are specified from the preamble, computes the cross-correlation between the input data and the preamble. When a sequence of input samples match the preamble, the filter output reaches its peak. The index of the peak corresponds to the end of the preamble sequence in the input data. See Discrete FIR Filter for further information on the FIR filter algorithm.

The cross-correlation values that are greater than or equal to the specified threshold are reported as peaks.

- If the detection threshold is too low, the algorithm will detect false peaks, or, in the extreme case, detect as many detected peaks as there are input samples.
- If the detection threshold is too high, the algorithm will miss detecting peaks, or, in the extreme case, detect no peaks.

Consequently, the selection of the detection threshold is critical.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CarrierSynchronizer` | `comm.CoarseFrequencyCompensator` | `comm.SymbolSynchronizer`

Introduced in R2016b

reset

System object: `comm.PreambleDetector`
Package: `comm`

Reset states of preamble detector object

Syntax

```
reset(prbdet)
```

Description

`reset(prbdet)` resets the states of the `PreambleDetector` object, `prbdet`.

Introduced in R2016b

step

System object: `comm.PreambleDetector`
Package: `comm`

Detect preamble in data

Syntax

```
idx = step(prbdet,x)
[idx,detmet] = step(prbdet,x)
idx = prbdet(x)
[idx,detmet] = prbdet(x)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`idx = step(prbdet,x)` returns the location of the end of the preamble in data sequence `x`, using preamble detector `prbdet`. The index is of data type `double`.

`[idx,detmet] = step(prbdet,x)` also returns the detection metric, `detmet`. This syntax is available when the `Input` property is `'Symbol'`. `detmet` has the same dimensions and data type as `x`.

The output, `detmet`, is determined by one of these algorithms:

- If either the preamble or input data is complex, the detection metric is the absolute value of the cross-correlation of the preamble and the input signal.
- If both the preamble and input data are real, the detection metric is the cross-correlation of the preamble and the input signal.

`idx = prbdet(x)` is equivalent to the first syntax.

`[idx,detmet] = prbdet(x)` is equivalent to the second syntax.

Note `prbdet` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

comm.PSKCoarseFrequencyEstimator

Package: `comm`

(To be removed) Estimate frequency offset for PSK signal

Note will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead.

Description

The `PSKCoarseFrequencyEstimator` System object estimates frequency offset for a PSK signal.

To estimate frequency offset for a PSK signal:

1. Define and set up your PSK coarse frequency estimator object. See “Construction” on page 3-1110.
2. Call `step` to estimate frequency offset for a PSK signal according to the properties of `comm.PSKCoarseFrequencyEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.PSKCoarseFrequencyEstimator` creates a PSK coarse frequency offset estimator object, H. This object uses an open-loop technique to estimate the carrier frequency offset in a received PSK signal.

`H = comm.PSKCoarseFrequencyEstimator(Name,Value)` creates a PSK coarse frequency offset estimator object, H, with the specified property `Name` set to the specified `Value`. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

`ModulationOrder`

Modulation order the object uses

Specify the modulation order of the PSK signal as a positive, real scalar of data type double. This value must be a positive power of 2. The default is `4`.

`Algorithm`

Estimation algorithm to object uses

Specify the estimation algorithm as one of `FFT-based` or `Correlation-based`. The default is `FFT-based`.

FrequencyResolution

Desired frequency resolution (Hz)

Specify the desired frequency resolution for offset frequency estimation as a positive, real scalar of data type double. This property establishes the FFT length used to perform spectral analysis, and must be less than or equal to half the SampleRate on page 3-0 property. This property applies only if the Algorithm property is FFT-based. The default is 0.001.

MaximumOffset

Maximum measurable frequency offset (Hz)

Specify the maximum measurable frequency offset as a positive, real scalar of data type double. The default is 0.05.

The value of this property must be less than SampleRate on page 3-0 / ModulationOrder on page 3-0 . It is recommended that MaximumOffset on page 3-0 be less than or equal to SampleRate on page 3-0 /(4*ModulationOrder on page 3-0). This property is active only if the Algorithm property is Correlation-based.

SampleRate

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type double. The default is 1.

Methods

reset	(To be removed) Reset states of the PSKCoarseFrequencyEstimator object
step	(To be removed) Estimate frequency offset for PSK signal

Common to All System Objects	
release	Allow System object property value changes

Examples

Compare Frequency Offset Estimation and Correction Methods for QPSK Signal

Estimate and correct for a frequency offset in a QPSK signal using the recommended comm.CoarseFrequencyCompensator System object. Compare frequency correction results to a workflow using the comm.PSKCoarseFrequencyEstimator System object.

Set example parameters.

```
nSym = 2048;     % Number of input symbols
M = 4;           % Modulation order
fs = 80000;      % Sampling frequency (Hz)
freqRez = 1;     % Frequency resolution (Hz)
freqOff = -2000; % Frequency offset
```

Create System objects for these operations:

```
% Square root raised cosine transmit filter
txfilter = comm.RaisedCosineTransmitFilter;
% Phase frequency offset - one to apply a frequency offset and a second
% that takes the frequency offset estimate as an input to correct the
% offset.
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',freqOff, ...
    'SampleRate',fs);
pfoCorrect = comm.PhaseFrequencyOffset(...
    'FrequencyOffsetSource','Input port', ...
    'SampleRate',fs);
% PSK coarse frequency estimator
frequencyEst = comm.PSKCoarseFrequencyEstimator(...
    'SampleRate',fs,'FrequencyResolution',freqRez);
```

```
Warning: comm.PSKCoarseFrequencyEstimator will be removed in a future release.
Use comm.CoarseFrequencyCompensator instead.
```

```
% Coarse frequency compensatator
freqComp = comm.CoarseFrequencyCompensator('Modulation','QPSK', ...
    'SampleRate',fs,'FrequencyResolution',freqRez);
```

Generate a QPSK signal, filter the signal, apply the frequency offset, and pass the signal through the AWGN channel.

```
data = randi([0 M-1],nSym,1);
modData = pskmod(data,M,pi/4);  % Generate QPSK signal
txFiltData = txfilter(modData); % Apply Tx filter
offsetData = pfo(txFiltData);   % Apply frequency offset
rxData = awgn(offsetData,25);   % Pass through AWGN channel
```

This example does not apply receive filtering. In general, when the frequency offset is high, it is beneficial to apply coarse frequency compensation prior to receive filtering because filtering suppresses energy in the useful spectrum.

Compare the results for estimating and correcting the frequency offset by:

- Using the `frequencyEst` object to estimate the frequency offset and `pfoCorrect` to compensate for the frequency offset.
- Using the `freqComp` object estimate and apply compensation to the signal.

Observe the frequency offset estimate returned by both estimation methods.

```
estFreqOffset1
estFreqOffset2
```

```
estFreqOffset1 =
  -2.0000e+03
estFreqOffset2 =
  -2.0000e+03
```

Confirm the maximum resulting difference between the two compensation methods is negligible.

```
max(compensatedData1-compensatedData2)
```

```
ans =

  1.4710e-13 + 9.1149e-14i
```

Selected Bibliography

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions”, *IEEE Transactions on Communications*, Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

Compatibility Considerations

`comm.PSKCoarseFrequencyEstimator` will be removed
Warns starting in R2020a

`comm.PSKCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead. For example, see “Compare Frequency Offset Estimation and Correction Methods for QPSK Signal” on page 3-1111.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also
`comm.CoarseFrequencyCompensator` | `comm.PhaseFrequencyOffset` | `dsp.FFT`

Introduced in R2013b

reset

System object: comm.PSKCoarseFrequencyEstimator
Package: comm

(To be removed) Reset states of the PSKCoarseFrequencyEstimator object

Note comm.PSKCoarseFrequencyEstimator will be removed in a future release. Use comm.CoarseFrequencyCompensator instead.

Syntax

```
reset(H)
```

Description

reset(H) resets the internal states of the PSKCoarseFrequencyEstimator object, H.

step

System object: `comm.PSKCoarseFrequencyEstimator`
Package: `comm`

(To be removed) Estimate frequency offset for PSK signal

Note `comm.PSKCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a complex column vector of data type double. The step method outputs the estimate `Y` as a scalar of type double.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.PSKDemodulator

Package: comm

Demodulate using M-ary PSK method

Description

The PSKDemodulator object demodulates an input signal using the M-ary phase shift keying (M-PSK) method.

To demodulate a signal that was modulated using phase shift keying:

1. Define and set up your PSK demodulator object. See “Construction” on page 3-1116.
2. Call step to demodulate the signal according to the properties of comm.PSKDemodulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.PSKDemodulator creates a demodulator System object, H. This object demodulates the input signal using the M-ary phase shift keying (M-PSK) method.

H = comm.PSKDemodulator(Name,Value) creates an M-PSK demodulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.PSKDemodulator(M,PHASE,Name,Value) creates an M-PSK demodulator object, H. This object has the ModulationOrder property set to M, the PhaseOffset property set to PHASE, and the other specified properties set to the specified values. M and PHASE are value-only arguments. To specify a value-only argument, you must also specify all preceding value-only arguments. You can specify name-value pair arguments in any order.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as a positive, integer scalar value. The default is 8.

PhaseOffset

Phase of zeroth point of constellation

Specify the phase offset of the zeroth point of the constellation, in radians, as a real scalar value. The default is `pi/8`.

BitOutput

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. The default is `false`. When you set this property to `true`, the `step` method outputs a column vector of bit values. The length of this vector equals **log2**(`ModulationOrder` on page 3-0) times the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector with a length equal to the input data vector. This vector contains integer symbol values between `0` and `ModulationOrder-1`.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol. Choose from `Binary` | `Gray` | `Custom`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-encoded signal constellation. When you set this property to `Binary`, the integer *m* must be in the range [`0`, (`ModulationOrder–1`)] and maps to the complex value **exp**(*j*□`PhaseOffset` on page 3-0 + *j*2π*m*/`ModulationOrder`). When you set this property to `Custom`, the object uses the signal constellation defined in the `CustomSymbolMapping` on page 3-0 property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is `0:7`. This property requires a row or column vector with a size of `ModulationOrder` on page 3-0 . This vector must have unique integer values in the range [`0`, `ModulationOrder–1`]. The values must be of data type `double`. The first element of this vector corresponds to the constellation point at an angle of `0` + `PhaseOffset` on page 3-0 , with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of – 2π/`ModulationOrder` + `PhaseOffset`. This property applies when you set the `SymbolMapping` on page 3-0 property to `Custom`.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as `Hard decision` | `Log-likelihood ratio` | `Approximate log-likelihood ratio`. The default is `Hard decision`. When you set the `BitOutput` on page 3-0 property to `false`, the object always performs hard decision demodulation. This property applies when you set the `BitOutput` property to `true`.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of `Property` | `Input port`. The default is `Property`. This property applies when you set the `BitOutput` on page 3-0 property to true and the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or –Inf. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, use approximate LLR instead because the algorithm for that option does not compute exponentials. This property applies when you set the BitOutput on page 3-0 property to true, the DecisionMethod on page 3-0 property to Log-likelihood ratio, or Approximate log-likelihood ratio, and the VarianceSource on page 3-0 property to Property. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32. The default is Full precision. This property applies when you set the BitOutput on page 3-0 property to false. It also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. In this second case, when the OutputDataType on page 3-0 property is set to Full precision, the input data type is single- or double-precision, the output data has the same data type as the input. . When the input data is of a fixed-point type, the output data type behaves as if you had set the OutputDataType property to Smallest unsigned integer.

When you set BitOutput to true and the DecisionMethod property to Hard Decision, then logical data type becomes a valid option. If you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data has the same data type as the input. In this case, the data type must be single- or double-precision.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as Same word length as input | Custom. The default is Same word length as input. This property applies when you set the BitOutput on page 3-0 property to false. It also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. The object uses the derotate factor in the computations only when the ModulationOrder on page 3-0 property is 2, 4, or 8. The step method input must also have a fixed-point type, and the PhaseOffset on page 3-0 property must have a nontrivial value. For ModulationOrder = 2, the phase offset is trivial if that value is a multiple of π/2. For ModulationOrder = 4, the phase offset is trivial if that value is an even multiple of π/4. For ModulationOrder = 8, there are no trivial phase offsets.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `DerotateFactorDataType` on page 3-0 property to `Custom`. The word length must be a value between 2 and 128.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using M-ary PSK method

Common to All System Objects	
`release`	Allow System object property value changes

Examples

16-PSK with Custom Symbol Mapping

Create 16-PSK modulator and demodulator System objects™ which use custom symbol mapping. Estimate the BER in an AWGN channel and compare the performance to a theoretical Gray-coded PSK system.

Create a custom symbol mapping for the 16-PSK modulation scheme. The 16 integer symbols must have values which fall between 0 and 15.

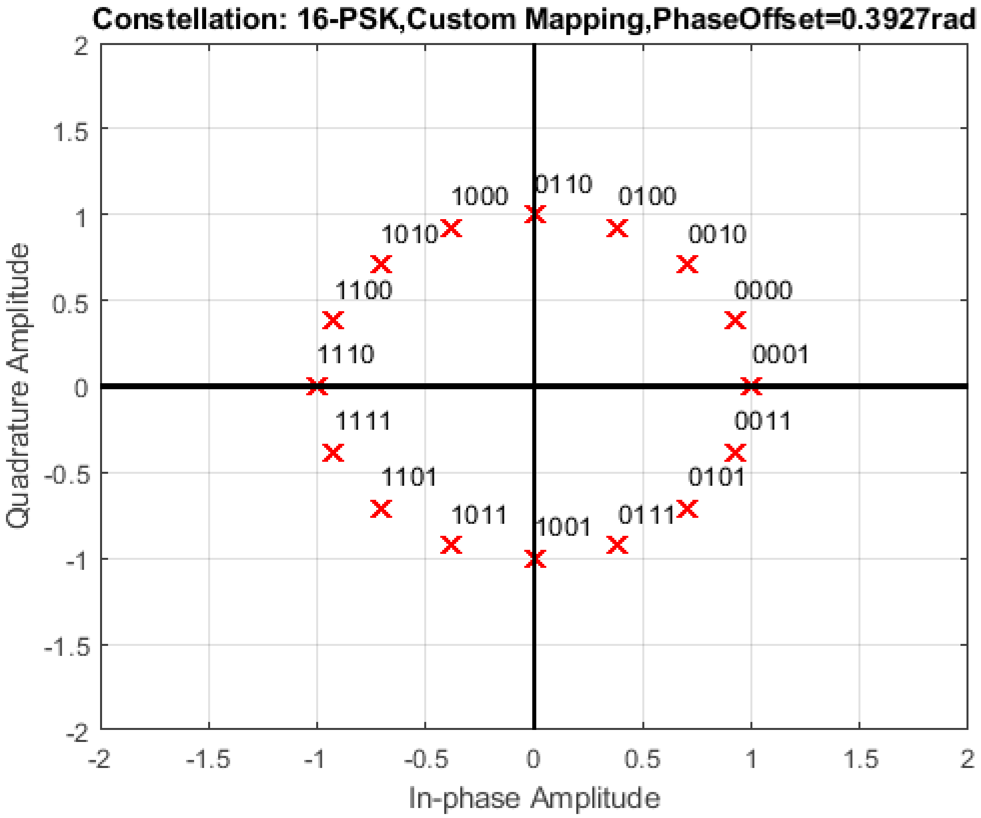
```
custMap = [0 2 4 6 8 10 12 14 15 13 11 9 7 5 3 1];
```

Create a 16-PSK modulator and demodulator pair having custom symbol mapping defined by the array, `custMap`.

```
pskModulator = comm.PSKModulator(16,'BitInput',true, ...
    'SymbolMapping','Custom','CustomSymbolMapping',custMap);
pskDemodulator = comm.PSKDemodulator(16,'BitOutput',true, ...
    'SymbolMapping','Custom','CustomSymbolMapping',custMap);
```

Display the modulator constellation.

```
constellation(pskModulator)
```



Create an AWGN channel System object for use with 16-ary data.

```
awgnChannel = comm.AWGNChannel('BitsPerSymbol',log2(16));
```

Create an error rate object to track the BER statistics.

```
errorRate = comm.ErrorRate;
```

Initialize the simulation vectors. The E_b/N_0 is varied from 6 to 18 dB in 1 dB steps.

```
ebnoVec = 6:18;
ber = zeros(size(ebnoVec));
```

Estimate the BER by modulating binary data, passing it through an AWGN channel, demodulating the received signal, and collecting the error statistics.

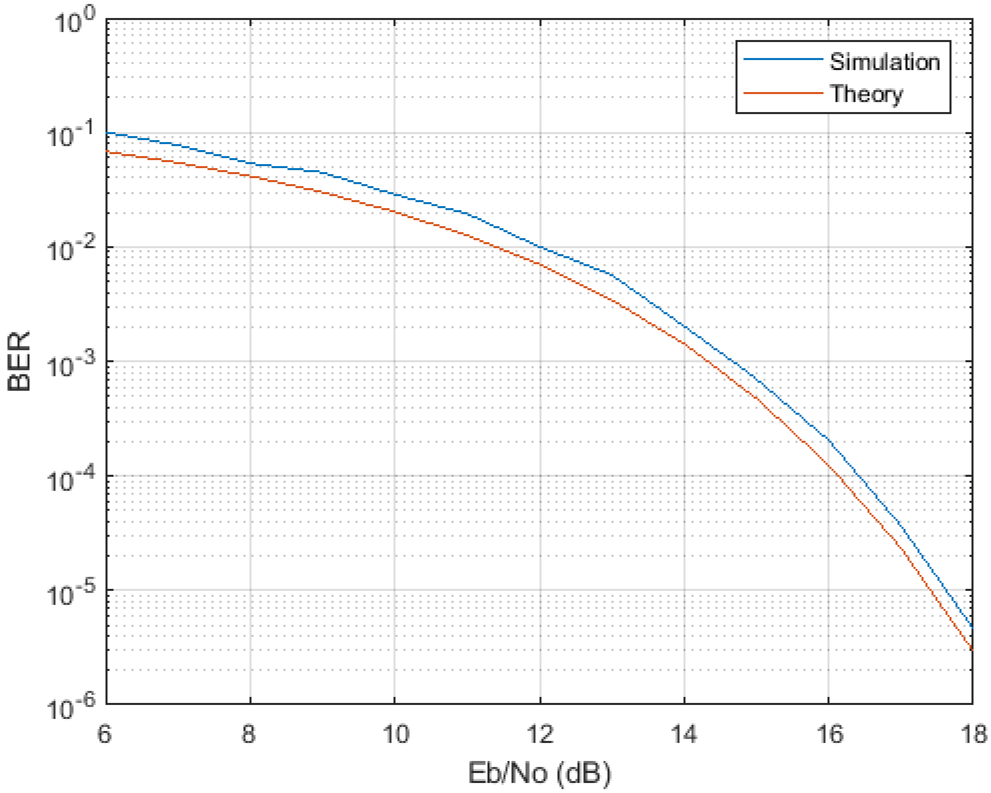
```
for k = 1:length(ebnoVec)

    % Reset the error counter for each Eb/No value
    reset(errorRate)
    % Reset the array used to collect the error statistics
    errVec = [0 0 0];
    % Set the channel Eb/No
    awgnChannel.EbNo = ebnoVec(k);

    while errVec(2) < 200 && errVec(3) < 1e7
        % Generate a 1000-symbol frame
        data = randi([0 1],4000,1);
        % Modulate the binary data
        modData = pskModulator(data);
        % Pass the modulated data through the AWGN channel
        rxSig = awgnChannel(modData);
        % Demodulate the received signal
        rxData = pskDemodulator(rxSig);
        % Collect the error statistics
        errVec = errorRate(data,rxData);
    end

    % Save the BER data
    ber(k) = errVec(1);
end
```

Generate theoretical BER data for an AWGN channel using the `berawgn` function.

```
berTheory = berawgn(ebnoVec,'psk',16,'nondiff');
```

Plot the simulated and theoretical results. The 16-PSK modulation BER performance of the simulated custom symbol mapping is not as good as the theoretical prediction curve for Gray codes.

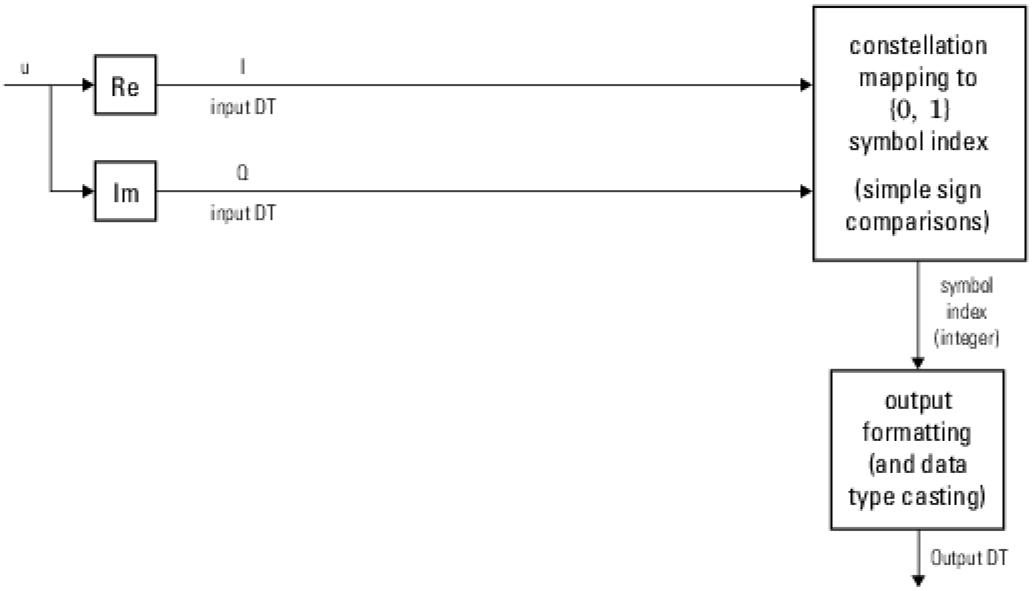
```
figure
semilogy(ebnoVec,[ber; berTheory])
xlabel('Eb/No (dB)')
ylabel('BER')
grid
legend('Simulation','Theory','location','ne')
```



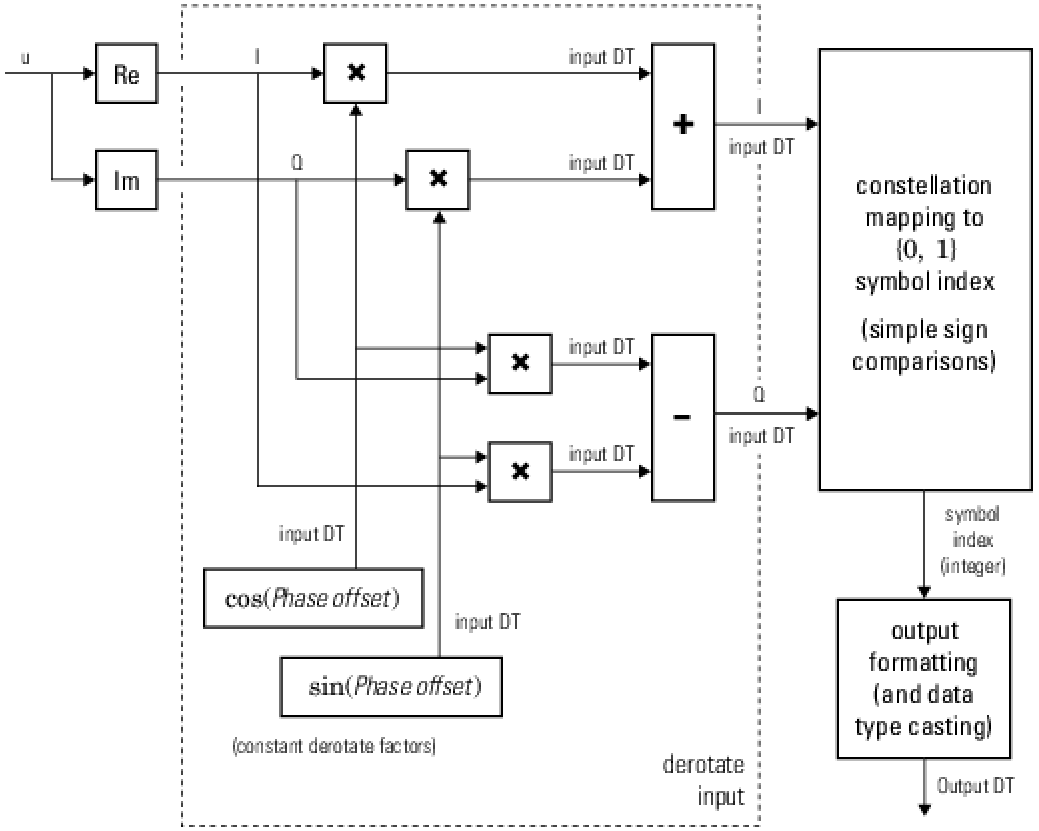
Algorithms

BPSK

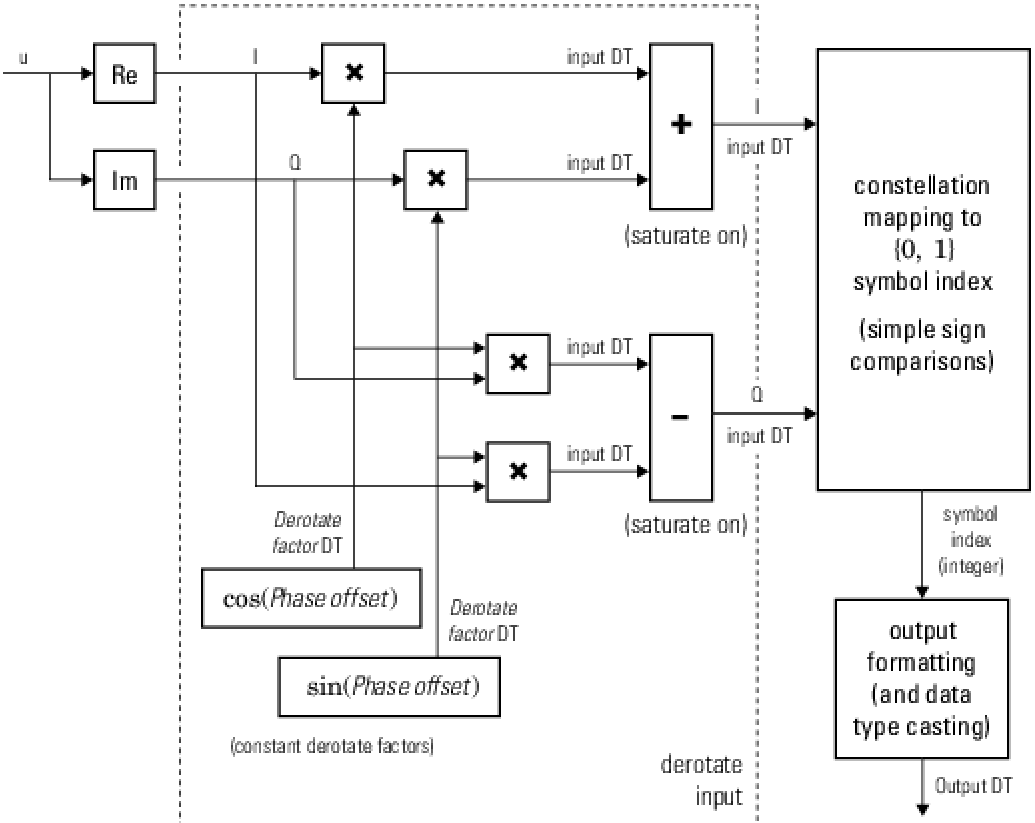
Diagrams for hard-decision demodulation of BPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)



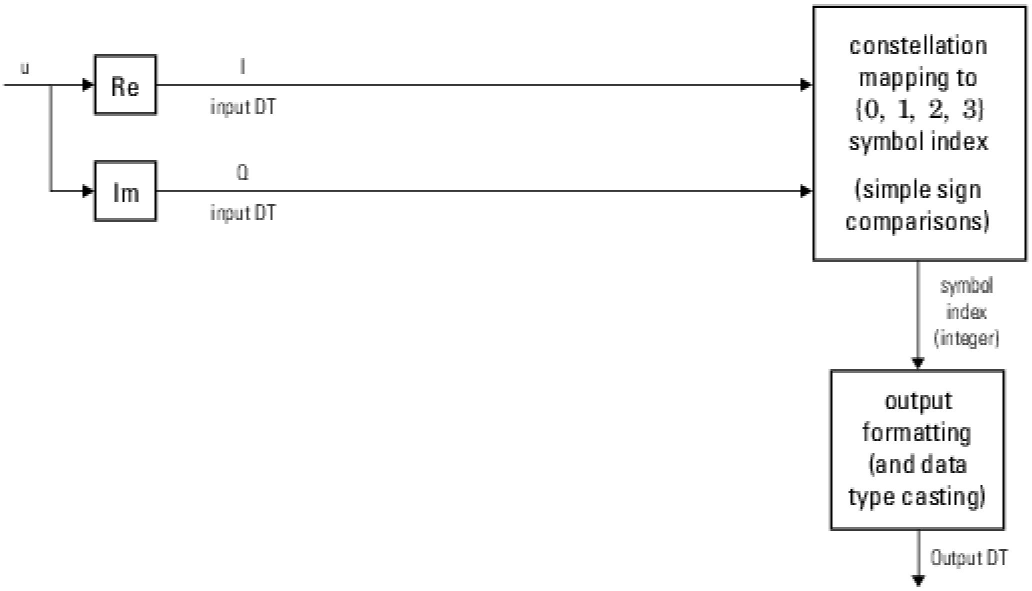
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



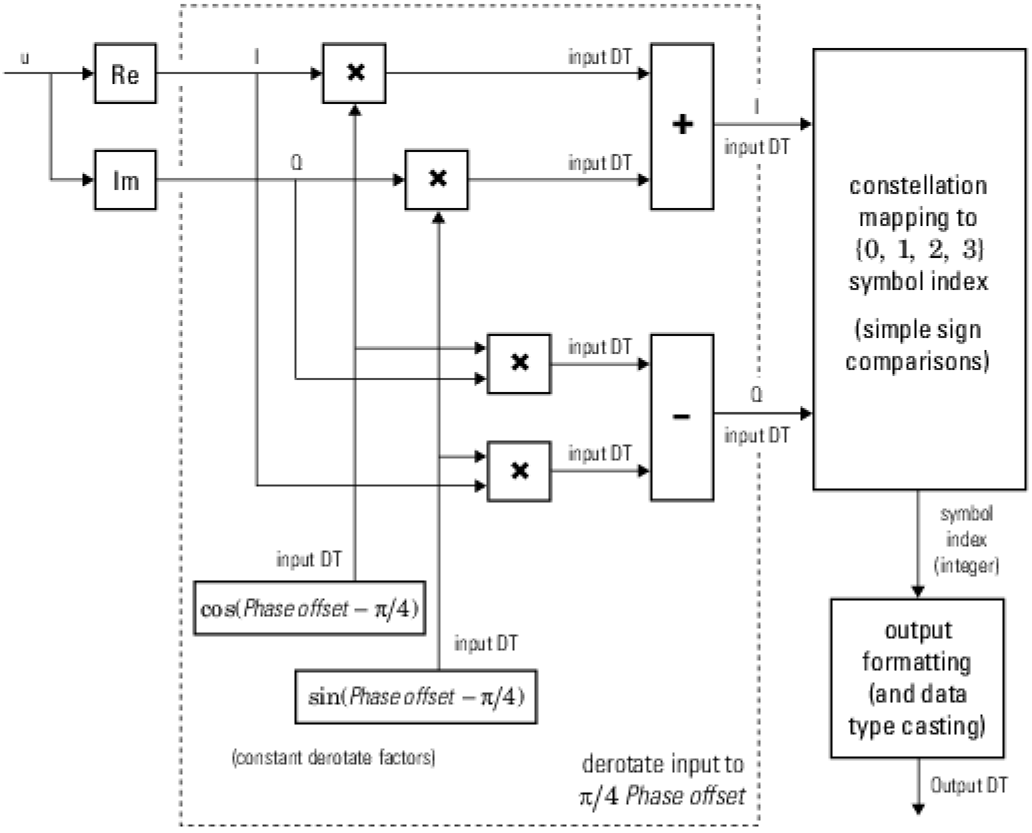
Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

QPSK

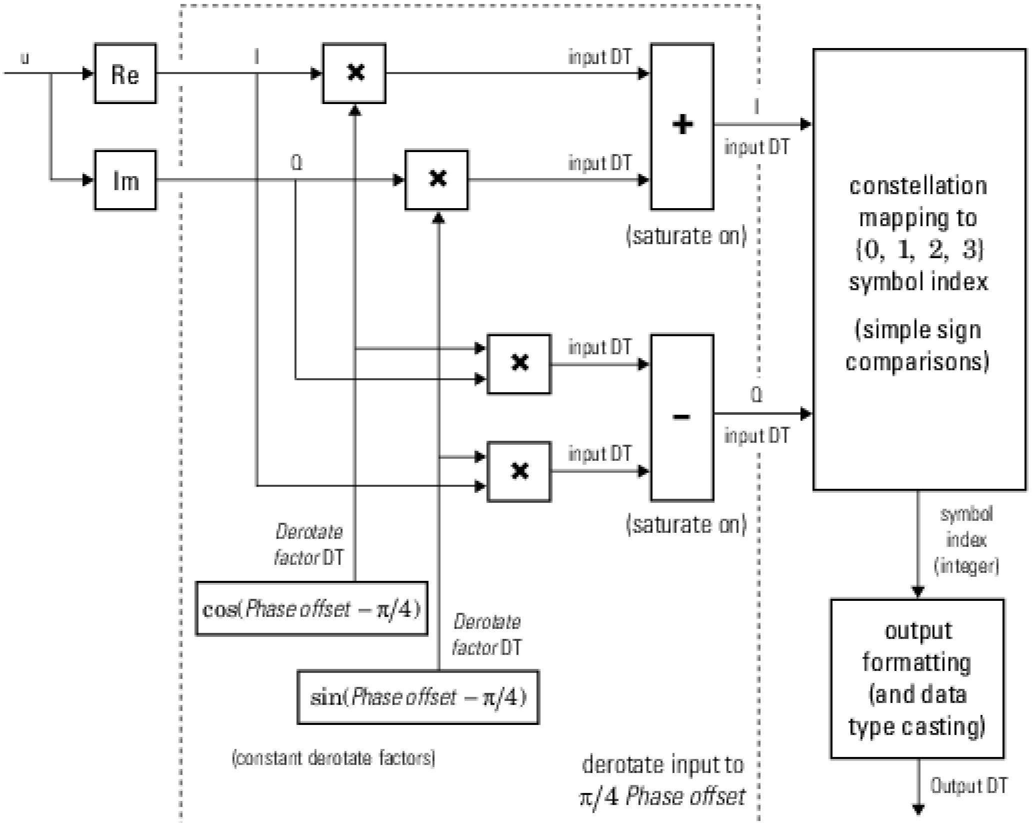
Diagrams for hard-decision demodulation of QPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of π/4)


u
Re
Im
I
Q
×
×
×
×
input DT
input DT
input DT
input DT
+
−
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
input DT
cos(Phase offset − π/4)
input DT
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset

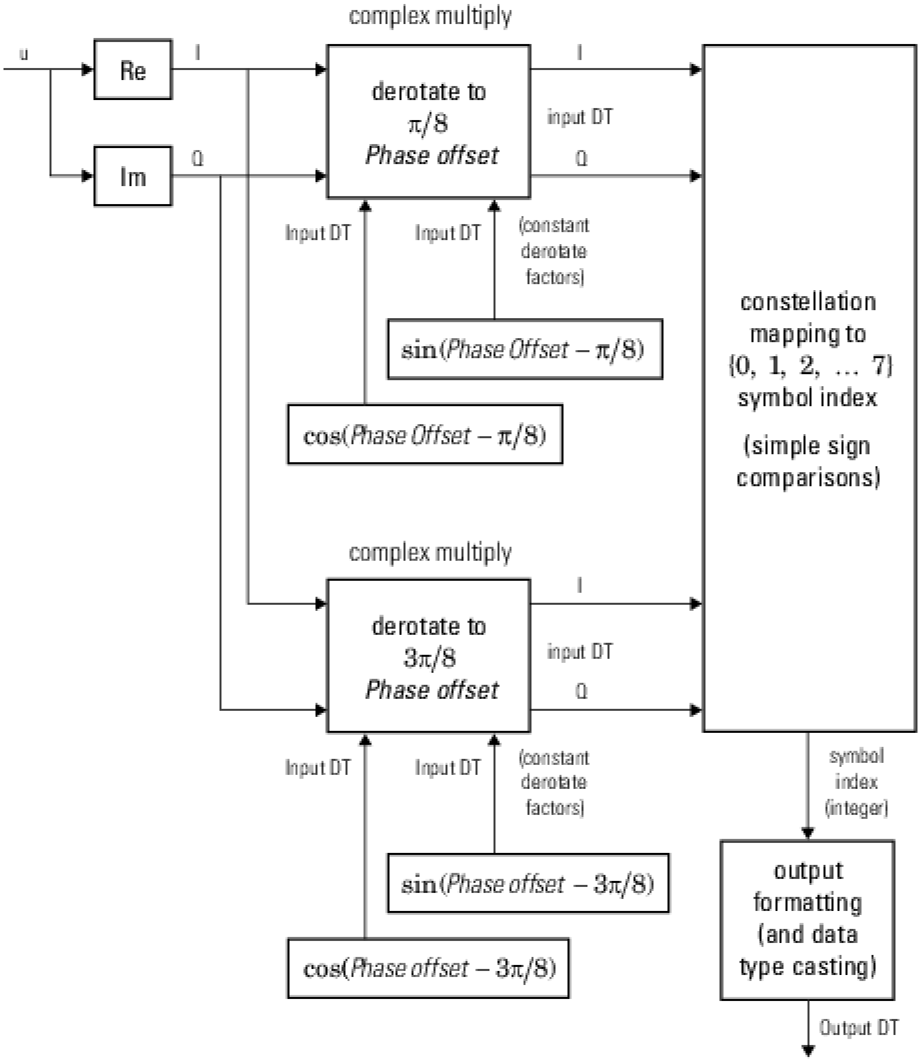
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



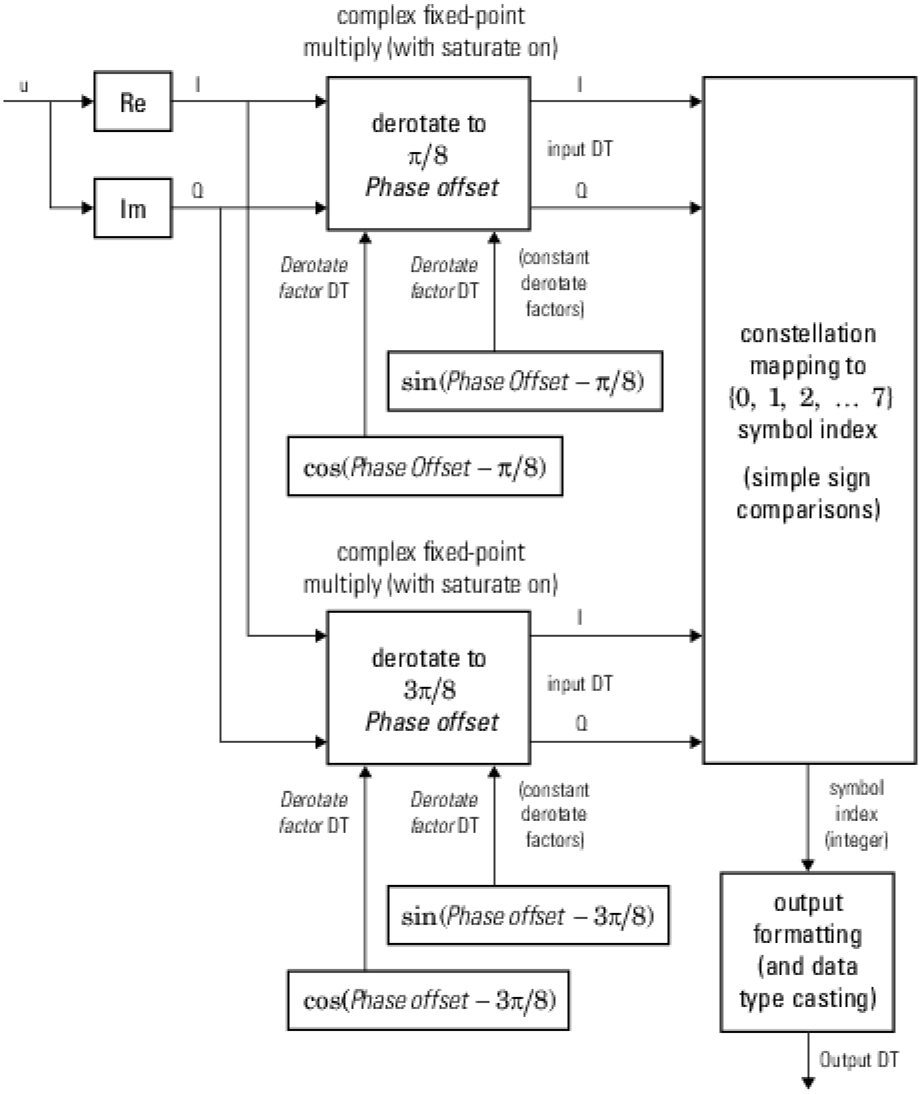
Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

Higher-Order PSK

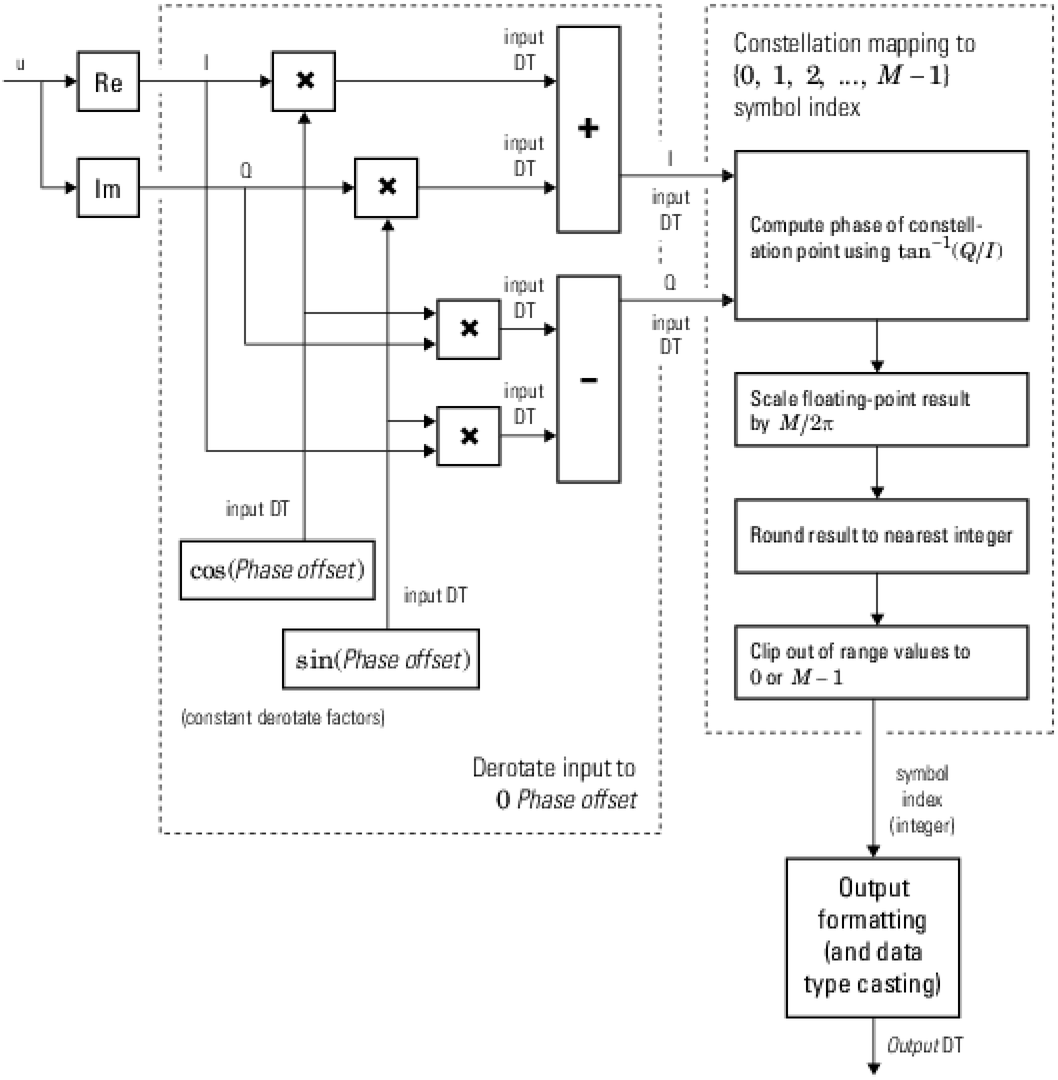
Diagrams for hard-decision demodulation of higher-order ($M \geq 8$) signals follow.



Hard-Decision 8-PSK Demodulator Floating-Point Signal Diagram



Hard-Decision 8-PSK Demodulator Fixed-Point Signal Diagram


u
Re
Im
I
Q
input DT
+
-
Constellation mapping to $\{0, 1, 2, ..., M-1\}$ symbol index
Compute phase of constellation point using $\tan^{-1}(Q/I)$
Scale floating-point result by $M/2\pi$
Round result to nearest integer
Clip out of range values to 0 or $M-1$
$\cos$(Phase offset)
$\sin$(Phase offset)
(constant derotate factors)
Derotate input to 0 Phase offset
symbol index (integer)
Output formatting (and data type casting)
Output DT

Hard-Decision M-PSK Demodulator (M > 8) Floating-Point Signal Diagram for Nontrivial Phase Offset

For $M > 8$, in order to improve speed and implementation costs, no derotation arithmetic is performed when `PhaseOffset` is 0, $\pi/2$, π, or $3\pi/2$ (i.e., when it is trivial).

Also, for $M > 8$, this block will only support inputs of type `double` and `single`.

Log-likelihood Ratio and Approximate Log-likelihood Ratio

The exact LLR and approximate LLR algorithms (soft-decision) are described in “Phase Modulation”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.DPSKDemodulator` | `comm.PSKModulator`

Introduced in R2012a

constellation

System object: `comm.PSKDemodulator`
Package: `comm`

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Plot PSK Reference Constellation

Create a PSK modulator.

```
mod = comm.PSKModulator;
```

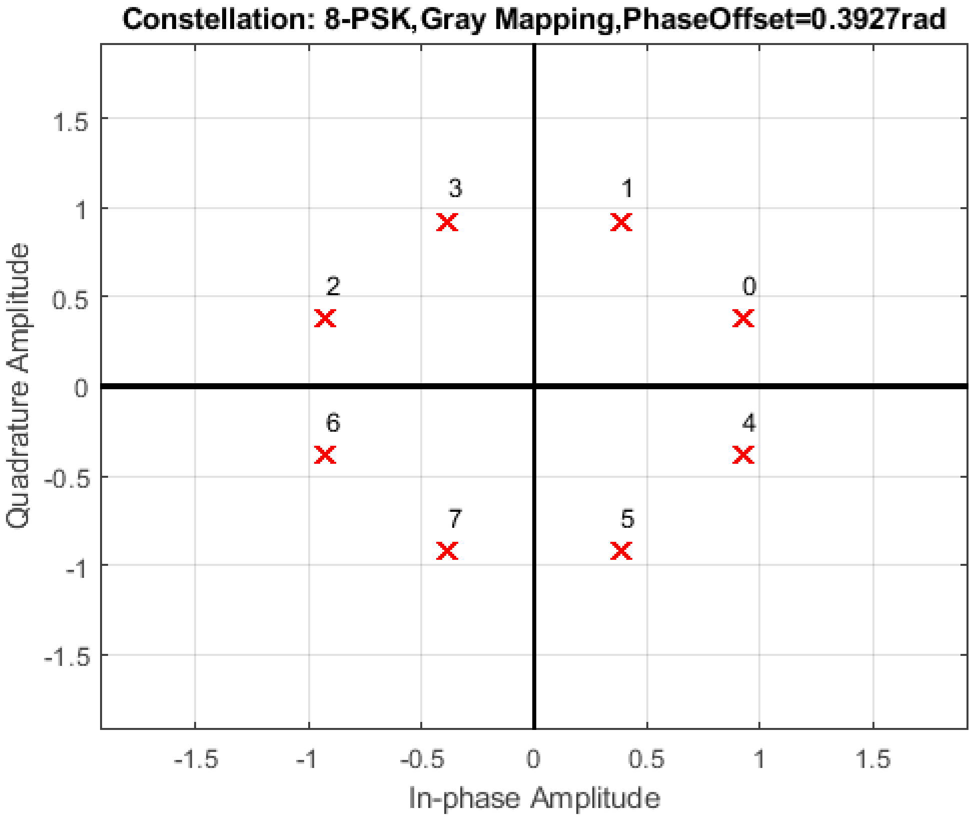
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 8×1 complex

   0.9239 + 0.3827i
   0.3827 + 0.9239i
  -0.3827 + 0.9239i
  -0.9239 + 0.3827i
  -0.9239 - 0.3827i
  -0.3827 - 0.9239i
   0.3827 - 0.9239i
   0.9239 - 0.3827i
```

Plot the constellation.

```
constellation(mod)
```

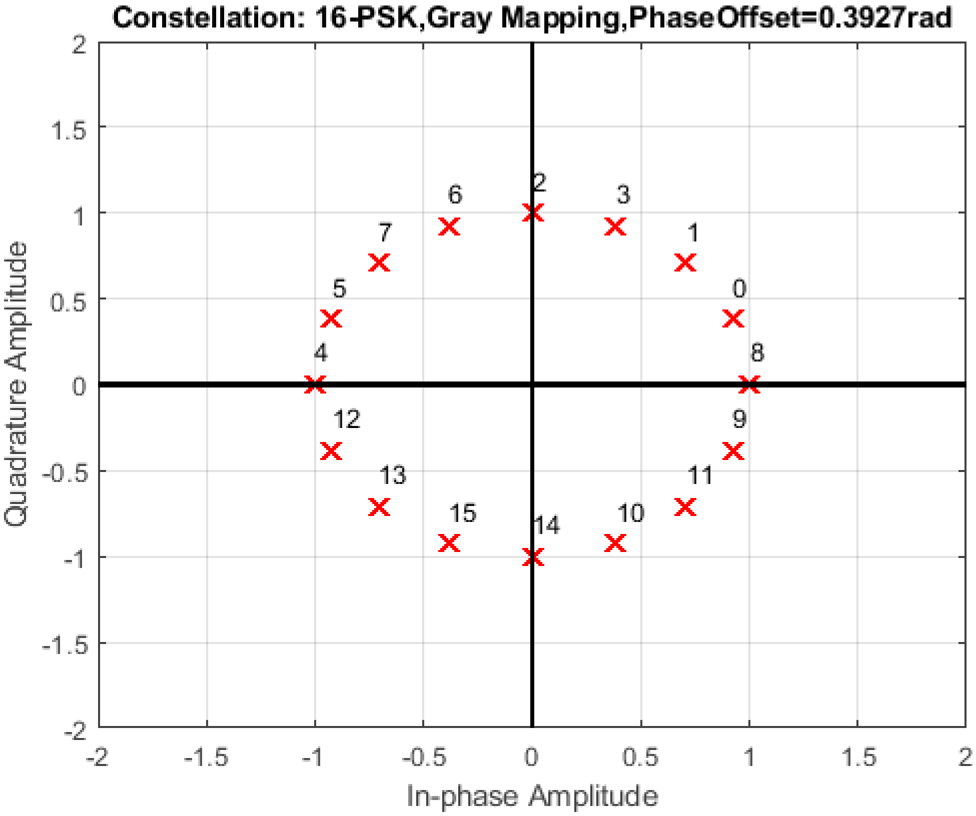


Create a PSK demodulator having modulation order 16.

```
demod = comm.PSKDemodulator(16);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```


Constellation: 16-PSK,Gray Mapping,PhaseOffset=0.3927rad
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

step

System object: `comm.PSKDemodulator`
Package: `comm`

Demodulate using M-ary PSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates data, `X`, with the PSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. If the value of the `ModulationOrder` property is less than or equal to `8` and you set `BitOutput` to `false`, or when you set the `DecisionMethod` property to `Hard Decision` and `BitOutput` to `true`, the object accepts an input with a signed integer data type or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y`, can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to true, the `DecisionMethod` property to `Approximate log-likelihood ratio` or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.PSKModulator

Package: `comm`

Modulate signal using M-PSK method

Description

The `PSKModulator` System object modulates using the M-ary phase shift keying (M-PSK) method. The output is a baseband representation of the modulated signal.

To modulate a signal by using the M-PSK method:

1. Create the `comm.PSKModulator` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
mpskmod = comm.PSKModulator
mpskmod = comm.PSKModulator(Name,Value)
mpskmod = comm.PSKModulator(M,phase,Name,Value)
```

Description

`mpskmod = comm.PSKModulator` creates a modulator System object `mpskmod`, that modulates the input signal using the M-ary phase shift keying (M-PSK) method.

`mpskmod = comm.PSKModulator(Name,Value)` creates an M-PSK modulator object `mpskmod`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`mpskmod = comm.PSKModulator(M,phase,Name,Value)` creates a M-PSK modulator object `mpskmod` using the modulation order specified in `M`. The object's `PhaseOffset` property is set to `phase`, and the other specified properties are set to the specified values.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`ModulationOrder` — Number of points in signal constellation

8 (default) | positive integer scalar

Number of points in the signal constellation, specified as a positive, integer scalar.

`PhaseOffset` — Phase of zeroth point of constellation

`pi/8` (default) | finite real-valued scalar

Phase of the zeroth point of the constellation in radians, specified as a finite real-valued scalar.

`BitInput` — Option to provide input in bits

`0` or `false` (default) | `1` or `true`

Option to provide input in bits, specified as a numeric or logical `0` (`false`) or `1` (`true`).

- If you set this property to `0` (`false`), the input values must be integer representations of two-bit input segments and range between 0 to 3.
- If you set this property to `1` (`true`), the input must be a binary vector of even length. Element pairs are binary representations of integers.

Data Types: `logical` | `char`

`SymbolMapping` — Signal constellation bit mapping

`'Gray'` (default) | `'Binary'` | `'Custom'`

Signal constellation bit mapping, specified as `'Gray'`, `'Binary'`, or `'Custom'`.

- `'Gray'` — Use this value to specify Gray-encoded signal constellation mapping.
- `'Binary'` — The integer *m* must be in the range [0, ModulationOrder – 1] and map to the complex value exp(*j*(PhaseOffset + 2π*m*/ModulationOrder))
- `'Custom'` — Use this value to specify the signal constellation mapping by using the CustomSymbolMapping property.

Data Types: `char` | `double`

`CustomSymbolMapping` — Custom constellation encoding

`[0 1 2 3 4 5 6 7]` (default) | row vector | column vector

Custom constellation encoding, specified as a row vector or column vector of values in the range [0, ModulationOrder – 1]. The length of this vector must be equal to the ModulationOrder property value. The first element of this vector corresponds to the constellation point at an angle of PhaseOffset, with subsequent elements running counterclockwise. The last element corresponds to the constellation point at an angle of – 2π/ModulationOrder + PhaseOffset.

Dependencies

To enable this property, set the SymbolMapping property to `'Custom'`.

Data Types: `double`

`OutputDataType` — Output datatype

`'double'` (default) | `'single'` | `'Custom'`

Output data type, specified as either `'double'`, `'single'` or `'Custom'`.

Fixed-Point Properties

`CustomOutputDataType` — Fixed-point data type of output
`numerictype([],16)` (default) | `numerictype` object

Fixed-point data type of the output signal, specified as a `numerictype` object with its `Signedness` property set to `Auto`. To create this type of object, use the `numerictype` function.

Dependencies

To enable this property, set the OutputDataType property to `'Custom'`.

Usage

Syntax

`mpsksignal = mpskmod(insignal)`

Description

`mpsksignal = mpskmod(insignal)` modulates the input signal by using the M-PSK method. The output is the modulated M-PSK baseband signal.

Input Arguments

`insignal` — Input signal
column vector

Input signal, specified as a column vector of integers or bits. The vector must be of length N_S, where N_S is the number of samples.

The setting of the BitInput property determines the interpretation of the input vector.

Data Types: `double`

Output Arguments

`mpsksignal` — M-PSK modulated baseband signal
vector

M-PSK modulated baseband signal, returned as a vector.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.PSKModulator`

constellation	Calculate or plot ideal signal constellation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Add White Gaussian Noise to 8-PSK Signal

Modulate an 8-PSK signal, add white Gaussian noise, and plot the signal to visualize the effects of the noise.

Create a M-PSK modulator System object™. The default modulation order for the object is 8.

```
pskModulator = comm.PSKModulator;
```

Modulate the signal.

```
modData = pskModulator(randi([0 7],2000,1));
```

Add white Gaussian noise to the modulated signal by passing the signal through an additive white Gaussian noise (AWGN) channel.

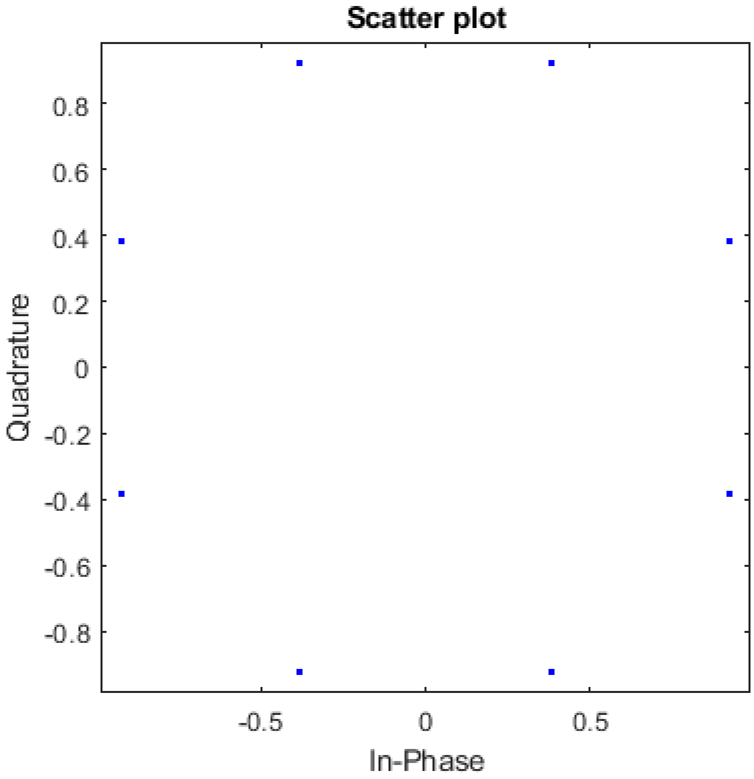
```
channel = comm.AWGNChannel('EbNo',20,'BitsPerSymbol',3);
```

Transmit the signal through the AWGN channel.

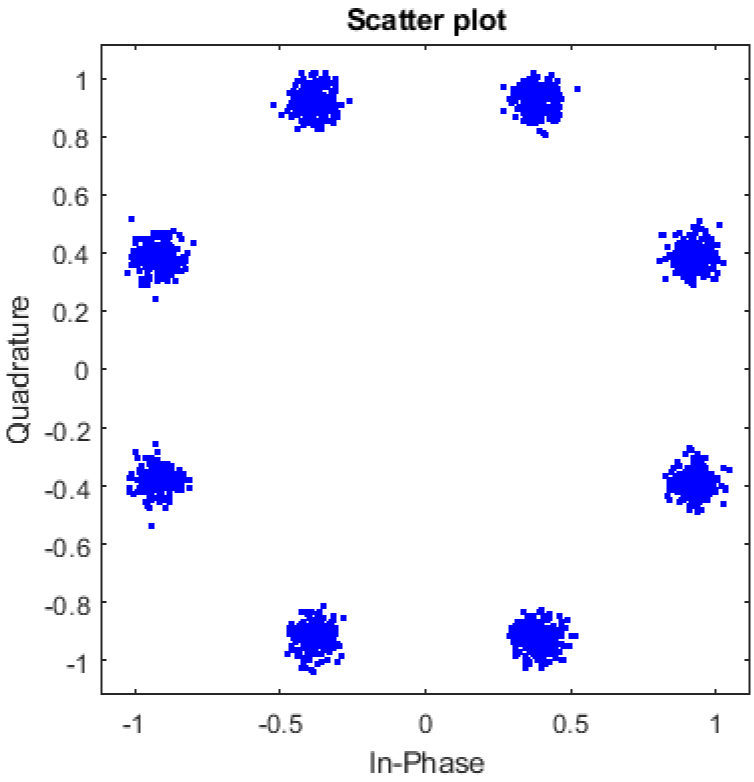
```
channelOutput = channel(modData);
```

Plot the noiseless and noisy data by using scatter plots to visualize the effects of the noise.

```
scatterplot(modData)
```



```
scatterplot(channelOutput)
```



Change the EbNo property to 10 dB to increase the noise.

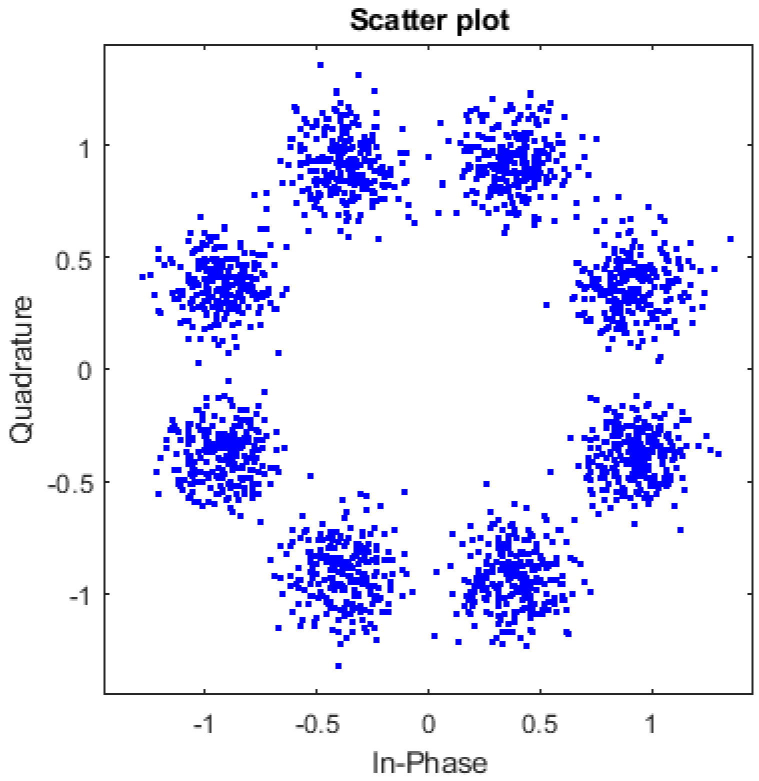
```
channel.EbNo = 10;
```

Pass the modulated data through the AWGN channel.

```
channelOutput = channel(modData);
```

Plot the channel output. You can see the effects of increased noise.

```
scatterplot(channelOutput)
```



16-PSK with Custom Symbol Mapping

Create 16-PSK modulator and demodulator System objects™ which use custom symbol mapping. Estimate the BER in an AWGN channel and compare the performance to a theoretical Gray-coded PSK system.

Create a custom symbol mapping for the 16-PSK modulation scheme. The 16 integer symbols must have values which fall between 0 and 15.

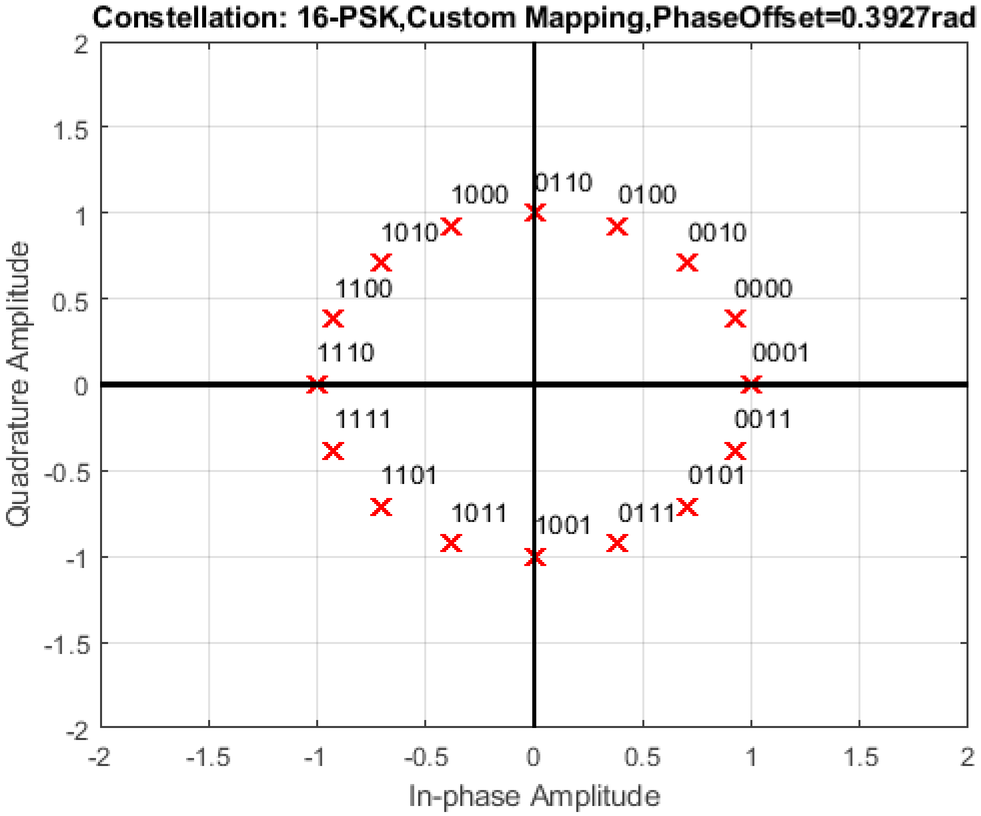
```
custMap = [0 2 4 6 8 10 12 14 15 13 11 9 7 5 3 1];
```

Create a 16-PSK modulator and demodulator pair having custom symbol mapping defined by the array, `custMap`.

```
pskModulator = comm.PSKModulator(16,'BitInput',true, ...
    'SymbolMapping','Custom','CustomSymbolMapping',custMap);
pskDemodulator = comm.PSKDemodulator(16,'BitOutput',true, ...
    'SymbolMapping','Custom','CustomSymbolMapping',custMap);
```

Display the modulator constellation.

```
constellation(pskModulator)
```



Create an AWGN channel System object for use with 16-ary data.

```
awgnChannel = comm.AWGNChannel('BitsPerSymbol',log2(16));
```

Create an error rate object to track the BER statistics.

```
errorRate = comm.ErrorRate;
```

Initialize the simulation vectors. The E_b/N_0 is varied from 6 to 18 dB in 1 dB steps.

```
ebnoVec = 6:18;
ber = zeros(size(ebnoVec));
```

Estimate the BER by modulating binary data, passing it through an AWGN channel, demodulating the received signal, and collecting the error statistics.

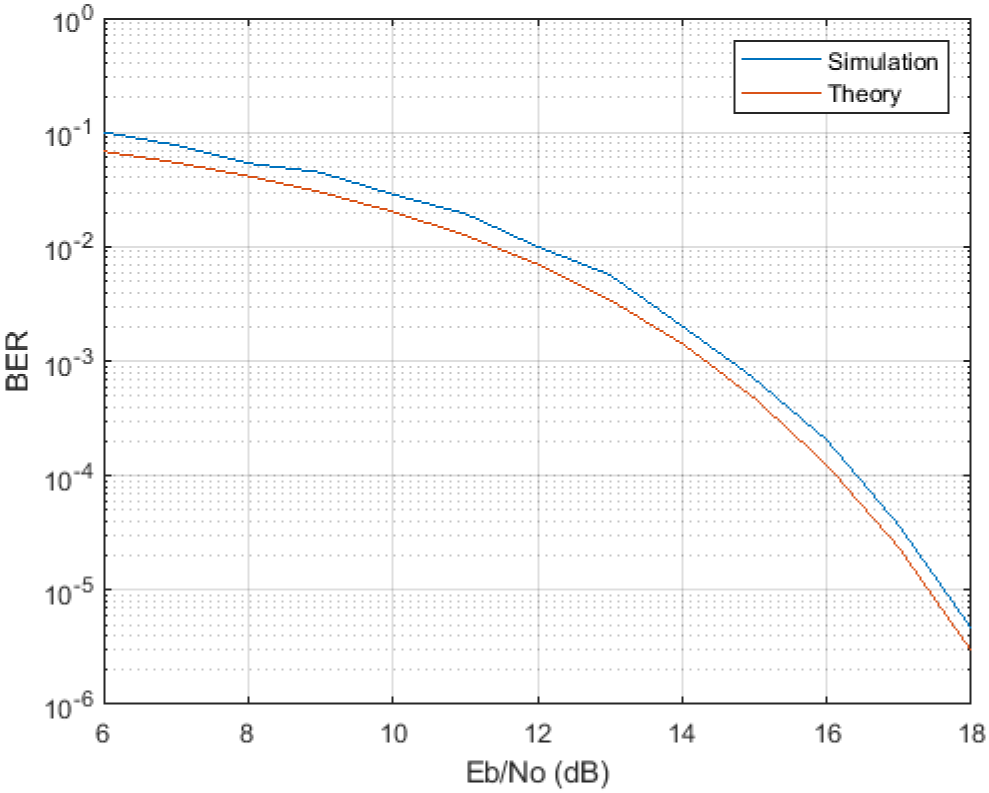
```
for k = 1:length(ebnoVec)

    % Reset the error counter for each Eb/No value
    reset(errorRate)
    % Reset the array used to collect the error statistics
    errVec = [0 0 0];
    % Set the channel Eb/No
    awgnChannel.EbNo = ebnoVec(k);

    while errVec(2) < 200 && errVec(3) < 1e7
        % Generate a 1000-symbol frame
        data = randi([0 1],4000,1);
        % Modulate the binary data
        modData = pskModulator(data);
        % Pass the modulated data through the AWGN channel
        rxSig = awgnChannel(modData);
        % Demodulate the received signal
        rxData = pskDemodulator(rxSig);
        % Collect the error statistics
        errVec = errorRate(data,rxData);
    end

    % Save the BER data
    ber(k) = errVec(1);
end
```

Generate theoretical BER data for an AWGN channel using the `berawgn` function.

```
berTheory = berawgn(ebnoVec,'psk',16,'nondiff');
```

Plot the simulated and theoretical results. The 16-PSK modulation BER performance of the simulated custom symbol mapping is not as good as the theoretical prediction curve for Gray codes.

```
figure
semilogy(ebnoVec,[ber; berTheory])
xlabel('Eb/No (dB)')
ylabel('BER')
grid
legend('Simulation','Theory','location','ne')
```



Algorithms

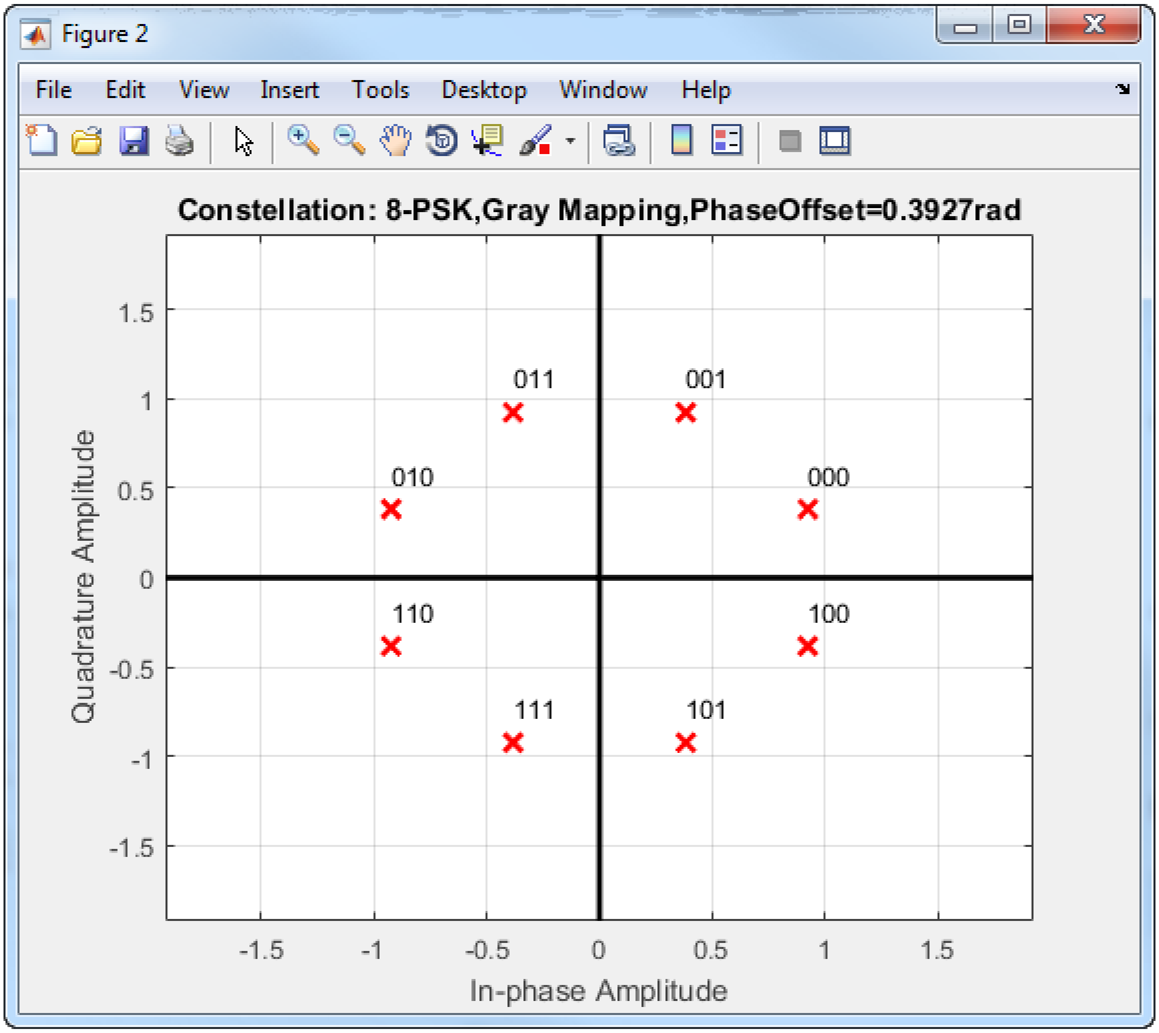
The block outputs a baseband signal by mapping input bits or integers to complex symbols according to the following:

$$s_n(t) = \exp\left(j\pi\left(\frac{2n+1}{M}\right)\right); \quad n \in \left\{0, 1, \ldots, M-1\right\}.$$

This applies when a natural binary ordering is used. Another common mapping is Gray coding, which has the advantage that only one bit changes between adjacent constellation points. This results in better bit error rate performance. For 8-PSK modulation with Gray coding, the mapping between the input and output symbols is shown.

Input	Output
0	0 *(000)*
1	1 *(001)*
2	3 *(011)*
3	2 *(010)*
4	6 *(110)*
5	7 *(111)*
6	5 *(101)*
7	4 *(100)*

The corresponding constellation diagram follows.



When the input signal is composed of bits, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $\log_2(M)$ bits.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See System Objects in MATLAB Code Generation (MATLAB Coder).

See Also

Objects

`comm.BPSKModulator` | `comm.QPSKDemodulator`

Functions

`pskmod`

Introduced in R2012a

constellation

System object: `comm.PSKModulator`
Package: `comm`

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Plot PSK Reference Constellation

Create a PSK modulator.

```
mod = comm.PSKModulator;
```

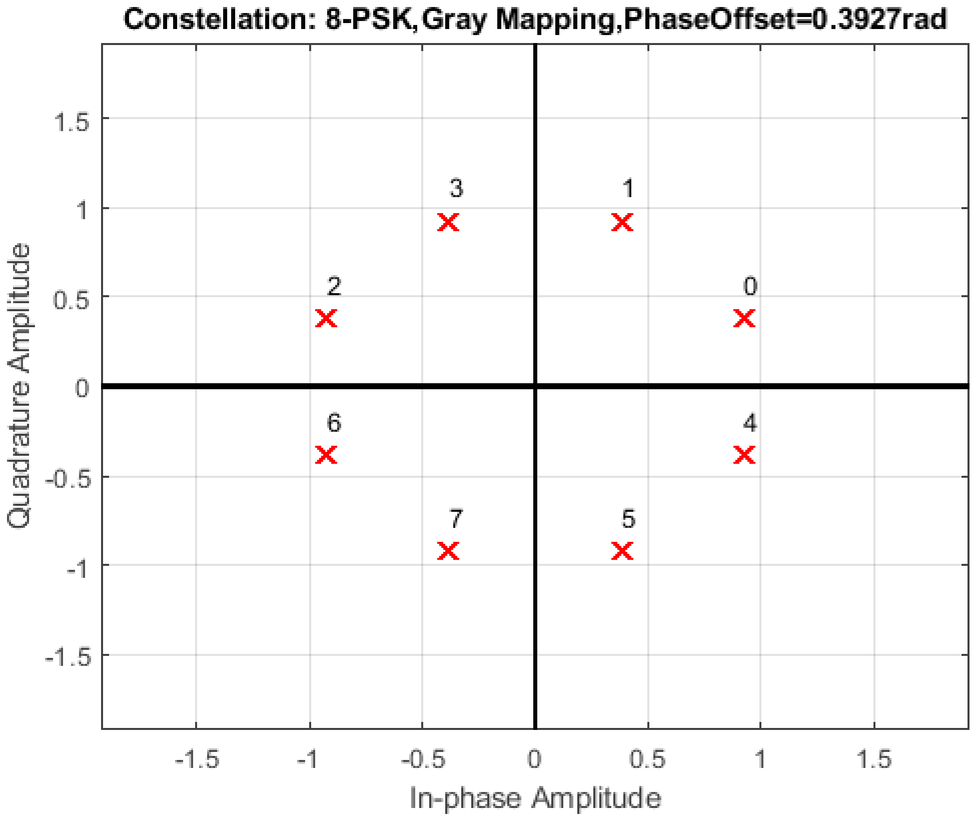
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 8×1 complex

   0.9239 + 0.3827i
   0.3827 + 0.9239i
  -0.3827 + 0.9239i
  -0.9239 + 0.3827i
  -0.9239 - 0.3827i
  -0.3827 - 0.9239i
   0.3827 - 0.9239i
   0.9239 - 0.3827i
```

Plot the constellation.

```
constellation(mod)
```

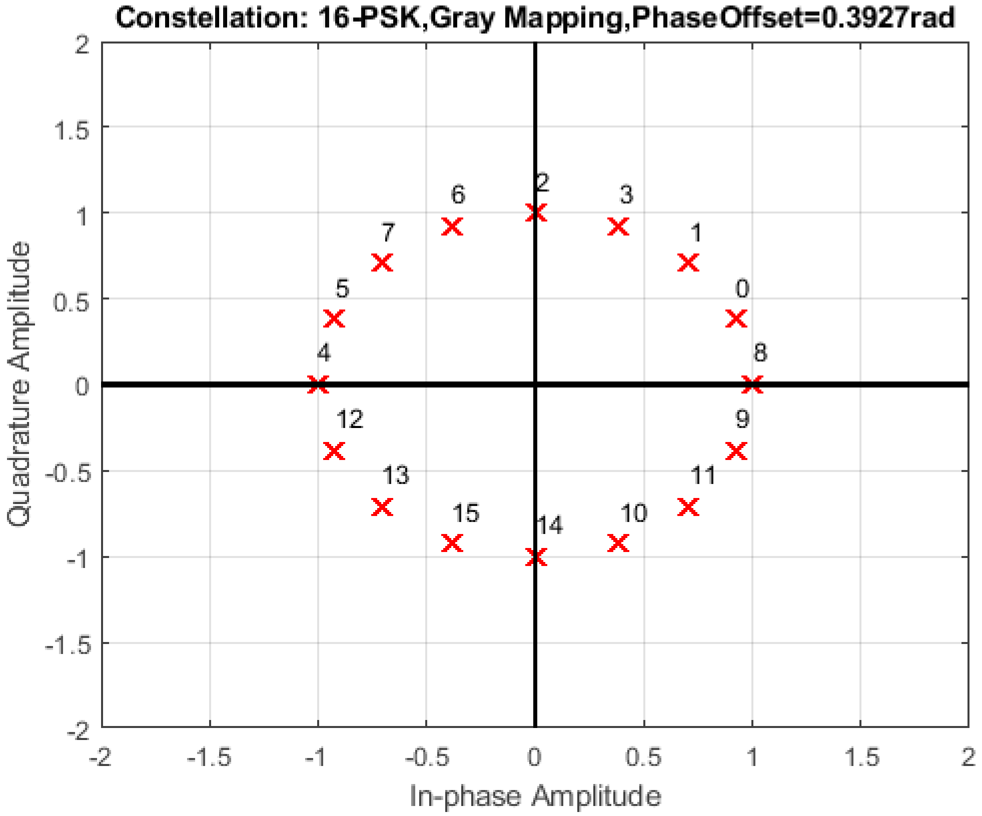


Create a PSK demodulator having modulation order 16.

```
demod = comm.PSKDemodulator(16);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```


Constellation: 16-PSK,Gray Mapping,PhaseOffset=0.3927rad
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
6
2
3
7
1
5
0
4
8
12
9
13
11
15
14
10

comm.PSKTCMDemodulator

Package: `comm`

Demodulate convolutionally encoded data mapped to M-ary PSK signal constellation

Description

The `PSKTCMDemodulator` object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a PSK signal constellation.

To demodulate a signal that was modulated using trellis-coded modulation:

1. Define and set up your PSK TCM demodulator object. See “Construction” on page 3-1150.
2. Call `step` to demodulate the signal according to the properties of `comm.PSKTCMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.PSKTCMDemodulator` creates a trellis-coded, M-ary phase shift, keying (PSK TCM) demodulator System object, `H`. This object demodulates convolutionally encoded data that has been mapped to an M-PSK constellation.

`H = comm.PSKTCMDemodulator(Name,Value)` creates a PSK TCM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.PSKTCMDemodulator(TRELLIS,Name,Value)` creates a PSK TCM demodulator System object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

`TrellisStructure`

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether the trellis structure is valid. The default is the result of `poly2trellis([1 3], [1 0 0; 0 5 2])`.

`TerminationMethod`

Termination method of encoded frame

Specify the termination method as one of `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

`TracebackDepth`

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The traceback depth influences the decoding accuracy and delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

When you set the `TerminationMethod` on page 3-0 property to `Continuous`, the decoding delay consists of TracebackDepth zero symbols or TracebackDepth×*K* zero bits for a rate *K*/*N* convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

`ResetInputPort`

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive, integer scalar value. The number of points must be `4`, `8`, or `16`. The default is `8`. The `ModulationOrder` on page 3-0 property value must equal the number of possible input symbols to the convolutional decoder of the PSK TCM demodulator object. The `ModulationOrder` property must equal 2^N for a rate *K*/*N* convolutional code.

`OutputDataType`

Data type of output

Specify output data type as `logical` | `double`. The default is `double`.

Methods

reset	Reset states of the PSK TCM demodulator object
step	Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Common to All System Objects	
release	Allow System object property value changes

Examples

Demodulate Noisy PSK QAM Data

Modulate and demodulate data using 8-PSK TCM modulation in an AWGN channel. Estimate the resulting error rate.

Define a trellis structure with four input symbols and eight output symbols.

```
t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
```

Create modulator and demodulator System objects™ using trellis, `t`, having modulation order 8.

```
hMod = comm.PSKTCMModulator(t,'ModulationOrder',8);
hDemod = comm.PSKTCMDemodulator(t,'ModulationOrder',8, ...
    'TracebackDepth',16);
```

Create an AWGN channel object.

```
hAWGN = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (SNR)', ...
    'SNR',7);
```

Create an error rate calculator with delay in bits equal to `TracebackDepth` times the number of bits per symbol.

```
hError = comm.ErrorRate('ReceiveDelay',...
    hDemod.TracebackDepth*log2(t.numInputSymbols));
```

Generate random binary data and modulate with 8-PSK TCM. Pass the modulated signal through the AWGN channel and demodulate. Calculate the error statistics.

```
for counter = 1:10
    % Transmit frames of 250 2-bit symbols
    data = randi([0 1],500,1);
    % Modulate
    modSignal = step(hMod,data);
    % Pass through AWGN channel
    noisySignal = step(hAWGN,modSignal);
    % Demodulate
    receivedData = step(hDemod,noisySignal);
    % Calculate error statistics
    errorStats = step(hError,data,receivedData);
end
```

Display the BER and the number of bit errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1),errorStats(2))
```

```
Error rate = 2.17e-02
Number of errors = 108
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK TCM Decoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMTCMDemodulator` | `comm.PSKTCMModulator` | `comm.RectangularQAMTCMDemodulator` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: `comm.PSKTCMDemodulator`
Package: `comm`

Reset states of the PSK TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `PSKTCMDemodulator` object, `H`.

step

System object: `comm.PSKTCMDemodulator`
Package: `comm`

Demodulate convolutionally encoded data mapped to M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates the PSK modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated, convolutionally encoded bits. `X` must be a complex, double or single precision column vector. The `step` method outputs a demodulated, binary data column vector, `Y`. When the convolutional encoder represents a rate *K*/*N* code, the length of the output vector is *K*× *L*, where *L* is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder to the all-zeros state when you input a reset signal, `R` that is non-zero. `R` must be a double precision or logical, scalar integer. This syntax applies when you set the `ResetInputPort` property to true.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.PSKTCMModulator

Package: comm

Convolutionally encode binary data and map using M-ary PSK signal constellation

Description

The PSKTCMModulator object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and then mapping the result to a PSK signal constellation.

To modulate a signal using trellis-coded modulation:

1. Define and set up your PSK TCM modulator object. See “Construction” on page 3-1156.
2. Call step to modulate the signal according to the properties of comm.PSKTCMModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.PSKTCMModulator creates a trellis-coded M-ary phase shift keying (PSK TCM) modulator System object, H. This object convolutionally encodes a binary input signal and maps the result to an M-PSK constellation.

H = comm.PSKTCMModulator(Name,Value) creates a PSK TCM encoder object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.PSKTCMModulator(TRELLIS,Name,Value) creates a PSK TCM encoder object, H. This object has the TrellisStructure property set to TRELLIS and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check whether a trellis structure is valid. The default is the result of poly2trellis([1 3], [1 0 0; 0 5 2]).

TerminationMethod

Termination method of encoded frame

Specify the termination method as one of `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. However, for each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate *K/N* code, the `step` method outputs the vector with a length given by $y = N \times {}^{(L+S)}/_{K}$, where $S = constraintLength–1$ (or, in the case of multiple constraint lengths, $S =$ **sum**(*constraintLength*(*i*)–1)). *L* indicates the length of the input to the `step` method.

`ResetInputPort`

Enable modulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive integer scalar value equal to `4`, `8`, or `16`. The default is `8`. The value of the `ModulationOrder` on page 3-0 property must equal the number of possible output symbols from the convolutional encoder of the PSK TCM modulator. Thus, the value for the `ModulationOrder` property must equal 2^N for a rate *K*/*N* convolutional code.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of the PSK TCM modulator object
step	Convolutionally encode binary data and map using M-ary PSK constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data Using 8-PSK TCM Modulation

Modulate random data using 8-PSK TCM modulation and display the constellation diagram.

Create binary data.

```
data = randi([0 1],1000,1);
```

Define a trellis structure with four input symbols and eight output symbols.

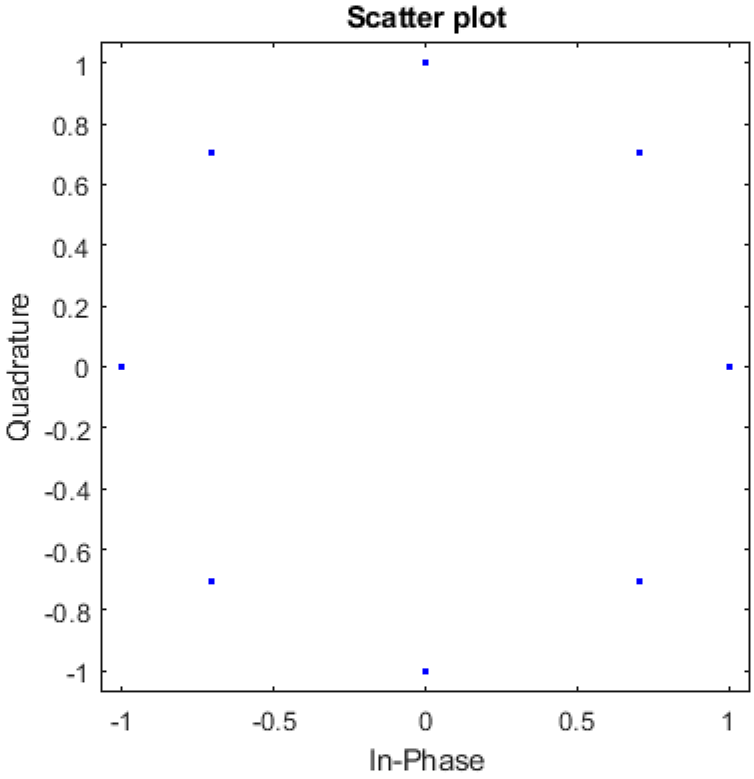
```
t =  poly2trellis([5 4],[23 35 0; 0 5 13]);
```

Create an 8-PSK TCM modulator object using the trellis structure variable, `t`.

```
hMod = comm.PSKTCMModulator(t,'ModulationOrder',8);
```

Modulate and plot the data.

```
modData = step(hMod,data);
scatterplot(modData);
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the M-PSK TCM Decoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMTCMModulator` | `comm.PSKTCMDemodulator` | `comm.RectangularQAMTCMModulator`

Introduced in R2012a

reset

System object: `comm.PSKTCMModulator`
Package: `comm`

Reset states of the PSK TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `PSKTCMModulator` object, `H`.

step

System object: `comm.PSKTCMModulator`
Package: `comm`

Convolutionally encode binary data and map using M-ary PSK constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` convolutionally encodes and modulates the input binary data column vector, `X`, and returns the encoded and modulated data, `Y`. `X` must be of data type numeric, logical, or unsigned fixed point of word length `1` (fi object). When the convolutional encoder represents a rate *K*/*N* code, the length of the input vector, `X`, must be *K*×*L*, for some positive integer *L*. The `step` method outputs a complex column vector, `Y`, of length *L*.

`Y = step(H,X,R)` resets the encoder of the PSK TCM modulator object to the all-zeros state when you input a reset signal, `R`, that is non-zero. `R` must be a double precision or logical scalar integer. This syntax applies when you set the `ResetInputPort` property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.QAMCoarseFrequencyEstimator

Package: `comm`

(To be removed) Estimate frequency offset for QAM signal

Note `comm.QAMCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead.

Description

The `QAMCoarseFrequencyEstimator` System object estimates frequency offset for a QAM signal.

To estimate frequency offset for a QAM signal:

1. Define and set up your QAM Coarse Frequency Estimator object. See "Construction" on page 3-1162.
2. Call `step` to estimate frequency offset for a QAM signal according to the properties of `comm.QAMCoarseFrequencyEstimator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.QAMCoarseFrequencyEstimator` creates a rectangular QAM coarse frequency offset estimator object, `H`. This object uses an open-loop, FFT-based technique to estimate the carrier frequency offset in a received rectangular QAM signal.

`H = comm.QAMCoarseFrequencyEstimator(Name,Value)` creates a rectangular QAM coarse frequency offset estimator object, `H`, with the specified property Name set to the specified Value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

`FrequencyResolution`

Desired frequency resolution (Hz)

Specify the desired frequency resolution for offset frequency estimation as a positive, real scalar of data type double. This property establishes the FFT length that the object uses to perform spectral analysis. The value for this property must be less than or equal to half the `SampleRate` on page 3-0 property. The default is `0.001`.

`SampleRate`

Sample rate (Hz)

Specify the sample rate in samples per second as a positive, real scalar of data type double. The default is 1.

Methods

reset	(To be removed) Reset states of the `QAMCoarseFrequencyEstimator` object
step	(To be removed) Estimate frequency offset for QAM signal

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Compare Frequency Offset Estimation and Correction Methods for 16-QAM Signal

Estimate and correct for a frequency offset in a 16-QAM signal using the recommended `comm.CoarseFrequencyCompensator` System object. Compare frequency correction results to a workflow using the `comm.QAMCoarseFrequencyEstimator` System object.

Set example parameters.

```
nSym = 2048;     % Number of input symbols
M = 16;          % Modulation order
fs = 80000;      % Sampling frequency (Hz)
freqRez = 1;     % Frequency resolution (Hz)
freqOff = -3000; % Frequency offset
```

Create System objects for these operations:

```
% Square root raised cosine transmit filter
txfilter = comm.RaisedCosineTransmitFilter;
% Phase frequency offset - one to apply a frequency offset and a
% second that takes the frequency offset estimate as an input to correct
% the offset.
pfo = comm.PhaseFrequencyOffset('FrequencyOffset',freqOff, ...
    'SampleRate',fs);
pfoCorrect = comm.PhaseFrequencyOffset(...
    'FrequencyOffsetSource','Input port', ...
    'SampleRate',fs);
% QAM coarse frequency estimator
frequencyEst = comm.QAMCoarseFrequencyEstimator(...
    'SampleRate',fs, ...
    'FrequencyResolution',freqRez);
```

```
Warning: comm.QAMCoarseFrequencyEstimator will be removed in a future release.
Use comm.CoarseFrequencyCompensator instead.
```

```
% Coarse frequency compensatator
freqComp = comm.CoarseFrequencyCompensator('Modulation','QAM', ...
    'SampleRate',fs,'FrequencyResolution',freqRez);
```

Generate a 16-QAM signal, filter the signal, apply the frequency offset, and pass the signal through the AWGN channel.

```
data = randi([0 M-1],nSym,1);
modData = qammod(data,M,'UnitAveragePower',true); % Generate QAM signal
txFiltData = txfilter(modData);                   % Apply Tx filter
offsetData = pfo(txFiltData);                     % Apply frequency offset
rxData = awgn(offsetData,25,'measured');          % Pass through AWGN channel
```

This example does not apply receive filtering. In general, when the frequency offset is high, it is beneficial to apply coarse frequency compensation prior to receive filtering because filtering suppresses energy in the useful spectrum.

Compare the results for estimating and correcting the frequency offset by:

- Using the `frequencyEst` object to estimate the frequency offset and `pfoCorrect` to compensate for the frequency offset.
- Using the `freqComp` object estimate and apply compensation to the signal.

Observe the frequency offset estimate returned by both estimation methods.

```
estFreqOffset1
estFreqOffset2
```

```
estFreqOffset1 =
  -3.0000e+03
estFreqOffset2 =
  -3.0000e+03
```

Confirm the maximum resulting difference between the two compensation methods is negligible.

```
max(compensatedData1-compensatedData2)
```

```
ans =

  -1.3051e-13 - 1.7614e-13i
```

Selected Bibliography

[1] Nakagawa, T., Matsui, M., Kobayashi, T., Ishihara, K., Kudo, R., Mizoguchi, M., and Y. Miyamoto. “Non-data-aided wide-range frequency offset estimator for QAM optical coherent receivers”, *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference* , March, 2011, pp. 1–3.

[2] Wang, Y., Shi. K., and E. Serpedin. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach”, *EURASIP Journal on Advances in Signal Processing*, Vol. 13, 2004, pp. 1993–2001.

Compatibility Considerations

`comm.QAMCoarseFrequencyEstimator` will be removed

Warns starting in R2020a

`comm.QAMCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead. For an example, see “Compare Frequency Offset Estimation and Correction Methods for 16-QAM Signal” on page 3-1163.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.CoarseFrequencyCompensator` | `comm.PhaseFrequencyOffset` | `dsp.FFT`

Introduced in R2013b

reset

System object: `comm.QAMCoarseFrequencyEstimator`
Package: `comm`

(To be removed) Reset states of the `QAMCoarseFrequencyEstimator` object

Note `comm.QAMCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead.

Syntax

```
reset(H)
```

Description

`reset(H)` resets the internal states of the `QAMCoarseFrequencyEstimator` object, `H`.

step

System object: `comm.PSKCoarseFrequencyEstimator`
Package: `comm`

(To be removed) Estimate frequency offset for QAM signal

Note `comm.QAMCoarseFrequencyEstimator` will be removed in a future release. Use `comm.CoarseFrequencyCompensator` instead.

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` estimates the carrier frequency offset of the input `X` and returns the result in `Y`. `X` must be a complex column vector of data type double. The step method outputs the estimate `Y` as a scalar of type double.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.QPSKDemodulator

Package: `comm`

Demodulate using QPSK method

Description

The `QPSKDemodulator` object demodulates a signal that was modulated using the quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

To demodulate a signal that was modulated using quadrature phase shift keying:

1. Define and set up your QPSK demodulator object. See “Construction” on page 3-1168.
2. Call `step` to demodulate the signal according to the properties of `comm.QPSKDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.QPSKDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the quadrature phase shift keying (QPSK) method.

`H = comm.QPSKDemodulator(Name,Value)` creates a QPSK demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.QPSKDemodulator(PHASE,Name,Value)` creates a QPSK demodulator object, `H`. This object has the `PhaseOffset` property set to `PHASE`, and the other specified properties set to the specified values.

Properties

`PhaseOffset`

Phase of zeroth point in constellation

Specify the phase offset of the zeroth point in the constellation, in radians, as a real scalar value. The default is `pi/4`.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values.

When you set this property to true, the step method outputs a column vector of bit values with length equal to twice the number of demodulated symbols.

When you set this property to false, the step method outputs a column vector with length equal to the input data vector. This vector contains integer symbol values between 0 and 3. The default is false.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of 2 bits to the corresponding symbol as one of Binary | Gray. The default is Gray.

When you set this property to Gray, the object uses a Gray-encoded signal constellation.

When you set this property to Binary, the integer m, between $0 \leq m \leq 3$ maps to the complex value **exp**(j×PhaseOffset on page 3-0 $+ j \times 2\pi \times {}^{m}\!/_{4}$).

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio. The default is Hard decision.

When you set the BitOutput on page 3-0 property to false, the object always performs hard decision demodulation. This property applies when you set the BitOutput property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as one of Property | Input port. The default is Property. This property applies when you set the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or -Inf. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, use approximate LLR is because that option's algorithm does not compute exponentials.

This property applies when you set the BitOutput on page 3-0 property to true, the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio, and the VarianceSource on page 3-0 property to Property.This property is tunable.

OutputDataType

Data type of output

Specify the output data type as Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32. The default is Full precision.

This property applies when you set the BitOutput on page 3-0 property to false. The property also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. In this second case, when the OutputDataType on page 3-0 property is set to Full precision, and the input data type is single or double precision, the output data has the same as that of the input.

When the input data is of a fixed-point type, the output data type behaves as if you had set the OutputDataType property to Smallest unsigned integer.

When you set BitOutput to true and the DecisionMethod property to Hard Decision, then logical data type becomes a valid option.

When you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data type is the same as that of the input. In this case, that data type can only be single or double precision.

Fixed-Point Properties

DerotateFactorDataType

Data type of derotate factor

Specify derotate factor data type as one of Same word length as input | Custom. The default is Same word length as input.

This property applies when you set the BitOutput on page 3-0 property to false. The property also applies when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. The object uses the derotate factor in the computations only when the step method input is a fixed-point type and the PhaseOffset on page 3-0 property has a value that is not an even multiple of pi/4.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the DerotateFactorDataType on page 3-0 property to Custom.

Methods

constellation	Calculate or plot ideal signal constellation
step	Demodulate using QPSK method

Common to All System Objects	
release	Allow System object property value changes

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

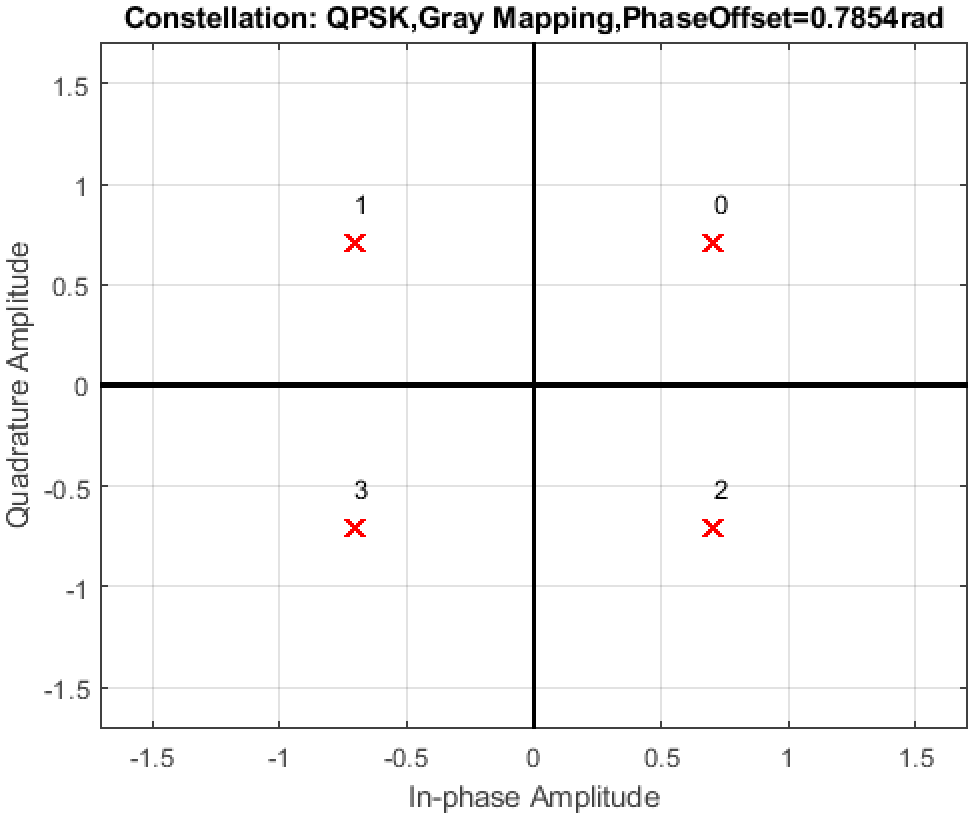
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

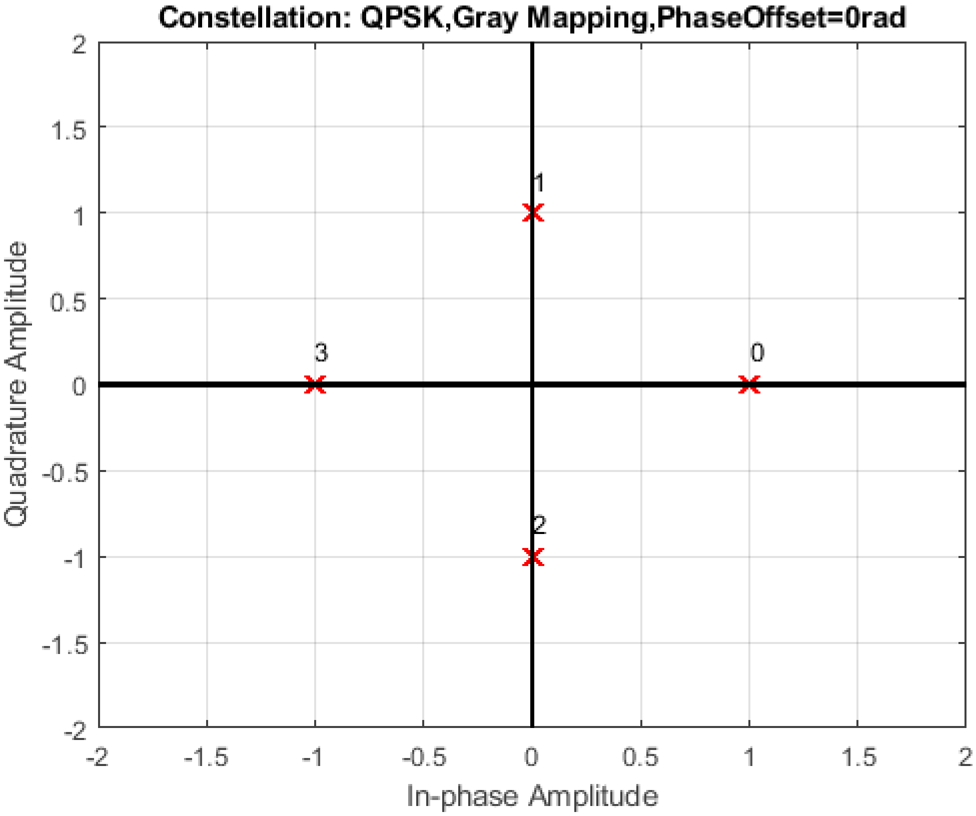


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



BER Estimate of QPSK Signal

Create a QPSK modulator and demodulator pair that operate on bits.

```
qpskModulator = comm.QPSKModulator('BitInput',true);
qpskDemodulator = comm.QPSKDemodulator('BitOutput',true);
```

Create an AWGN channel object and an error rate counter.

```
channel = comm.AWGNChannel('EbNo',4,'BitsPerSymbol',2);
errorRate = comm.ErrorRate;
```

Generate random binary data and apply QPSK modulation.

```
data = randi([0 1],1000,1);
txSig = qpskModulator(data);
```

Pass the signal through the AWGN channel and demodulate it.

```
rxSig = channel(txSig);
rxData = qpskDemodulator(rxSig);
```

Calculate the error statistics. Display the BER.

```
errorStats = errorRate(data,rxData);
```

```
errorStats(1)
```

```
ans = 0.0100
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the QPSK Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.PSKDemodulator` | `comm.QPSKModulator`

Topics

“Log-likelihood Ratio (LLR) Demodulation”

Introduced in R2012a

constellation

System object: comm.QPSKDemodulator
Package: comm

Calculate or plot ideal signal constellation

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

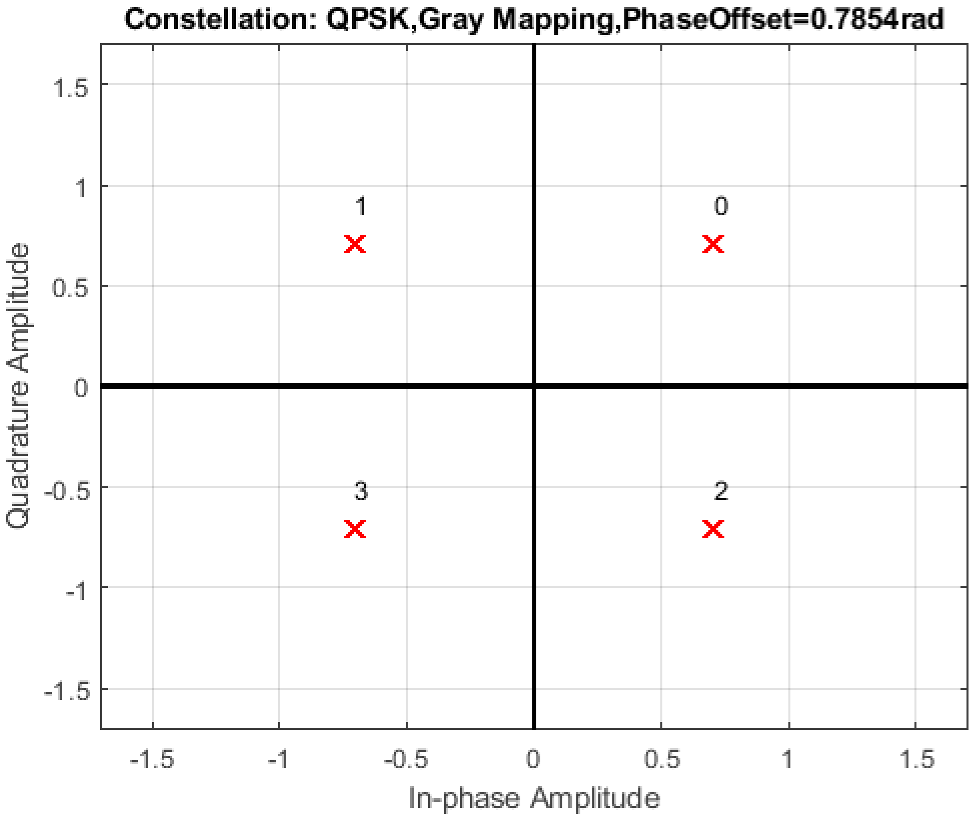
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

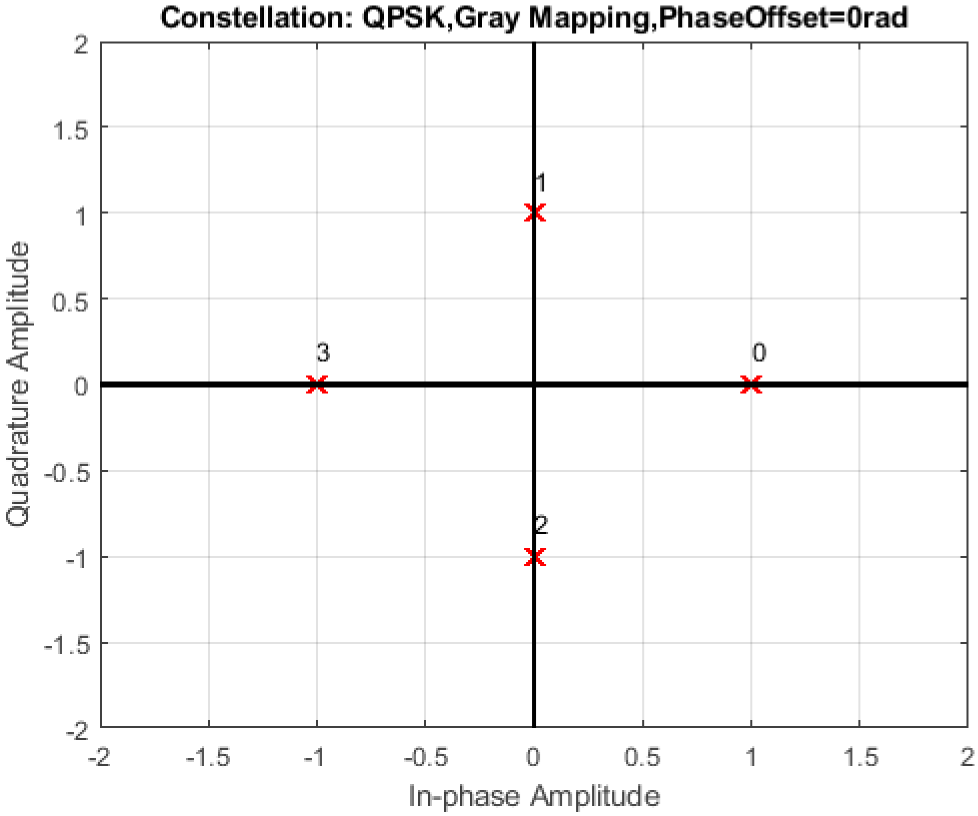


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```


Constellation: QPSK,Gray Mapping,PhaseOffset=0rad
Quadrature Amplitude
In-phase Amplitude
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
1
3
0
2

step

System object: `comm.QPSKDemodulator`
Package: `comm`

Demodulate using QPSK method

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates input data, `X`, with the QPSK demodulator System object, `H`, and returns `Y`. Input `X` must be a scalar or a column vector with double or single precision data type. When you set the `BitOutput` property to `false`, or when you set the `DecisionMethod` property to `Hard decision` and the `BitOutput` property to `true`, the data type of the input can also be signed integer, or signed fixed point (fi objects). Depending on the `BitOutput` property value, output `Y` can be integer or bit valued.

`Y = step(H,X,VAR)` uses soft decision demodulation and noise variance `VAR`. This syntax applies when you set the `BitOutput` property to `true`, the `DecisionMethod` property to `Approximate log-likelihood ratio`or `Log-likelihood ratio`, and the `VarianceSource` property to `Input port`. The data type of input `VAR` must be double or single precision.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.QPSKModulator

Package: comm

Modulate using QPSK method

Description

The comm.QPSKModulator object modulates using the quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

To apply QPSK modulation:

1. Create the comm.QPSKModulator object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
qpskmod = comm.QPSKModulator
qpskmod = comm.QPSKModulator(Name,Value)
qpskmod = comm.QPSKModulator(phase,Name,Value)
```

Description

qpskmod = comm.QPSKModulator creates a modulator System object. Use this object to modulate the input signal using the quadrature phase shift keying (QPSK) method.

qpskmod = comm.QPSKModulator(Name,Value) creates a QPSK modulator object, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

qpskmod = comm.QPSKModulator(phase,Name,Value) creates a QPSK modulator object, qpskmod. This object has the PhaseOffset property set to phase and the other specified properties set to the specified values. Specify phase in radians.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

PhaseOffset — Phase offset of zeroth point in constellation

pi/4 (default) | scalar

Phase of zeroth point of the signal constellation in radians, specified as a scalar.

Example: 'PhaseOffset',0 aligns the QPSK signal constellation points on the axes, {(1,0), (0,j), (-1,0), (0,-j)}.

Data Types: double

BitInput — Option to provide input in bits

false (default) | true

Option to provide input in bits, specified as false or true.

- When this property is set to false, the input values must be integer representations of two-bit input segments and range from 0 to 3.
- When this property is set to true, the input must be a binary vector of even length. Element pairs are binary representations of integers.

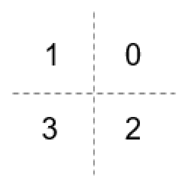
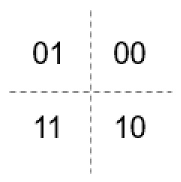
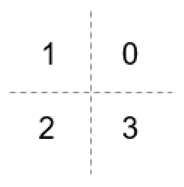
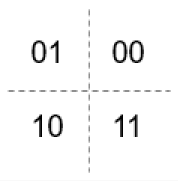
Data Types: logical

SymbolMapping — Signal constellation bit mapping

'Gray' (default) | 'Binary'

Constellation encoding

Signal constellation bit mapping, specified as 'Gray' or 'Binary'.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
Gray			The signal constellation mapping is Gray-encoded.
Binary			The signal constellation mapping for the input integer m (0 ≤ m ≤ 3) is the complex value $e^{(j*(PhaseOffset) + j*2*\pi*m/4)}$.

Data Types: char

OutputDataType — Data type assigned to output

'double' (default) | 'single' | 'Custom'

Data type assigned to output, specified as 'double', 'single', or 'Custom'.

Data Types: char

Fixed-Point Properties

`CustomOutputDataType` — Fixed-point data type of output
`numerictype([],16)` (default) | `numerictype` object

Fixed-point data type of output, specified as a `numerictype` object with a signedness of Auto. This property applies when you set the `OutputDataType` on page 3-0 property to `Custom`.

Dependencies

This property applies when you set the `OutputDataType` property to `'Custom'`.

Usage

Syntax

```
waveform = qpskmod(insignal)
```

Description

`waveform = qpskmod(insignal)` returns baseband-modulated output.

Input Arguments

`insignal` — Input signal
integer column vector | bit column vector

Input signal, specified as an N_S-element column vector of integers or bits, where N_S is the number of samples.

The setting of the BitInput property determines the interpretation of the input vector.

Data Types: `double` | `int8` | `logical` | `fi`

Output Arguments

`waveform` — Output waveform
vector

Output waveform, returned as a complex-valued vector.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.QPSKModulator`

constellation	Calculate or plot ideal signal constellation

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

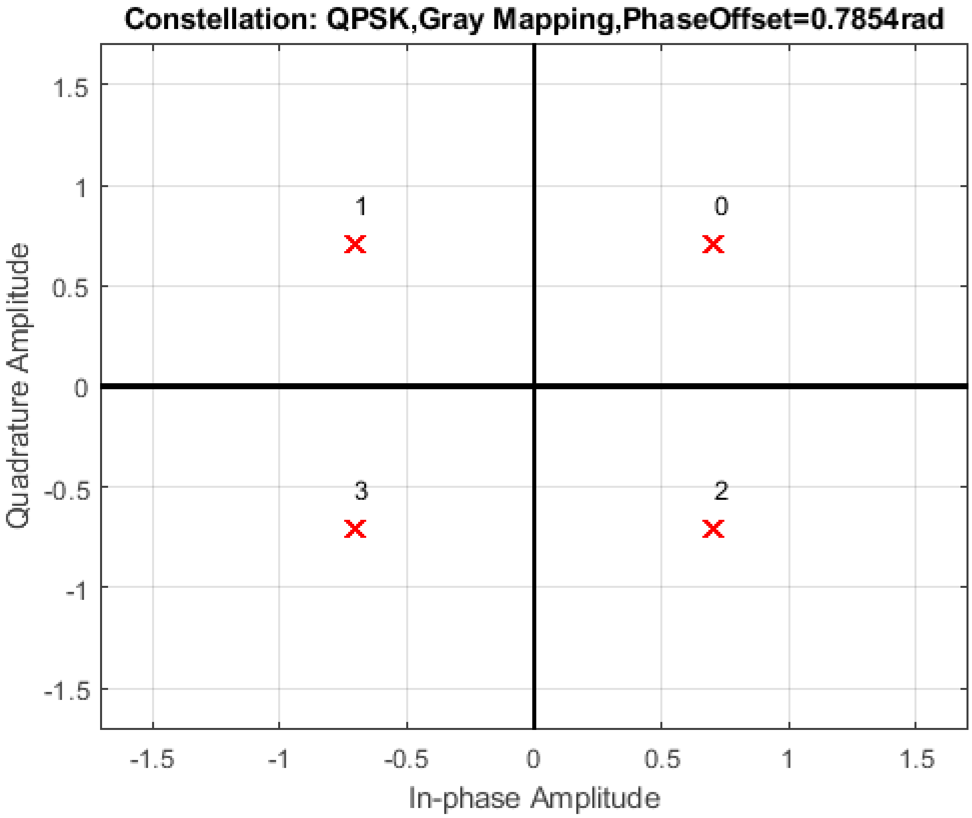
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

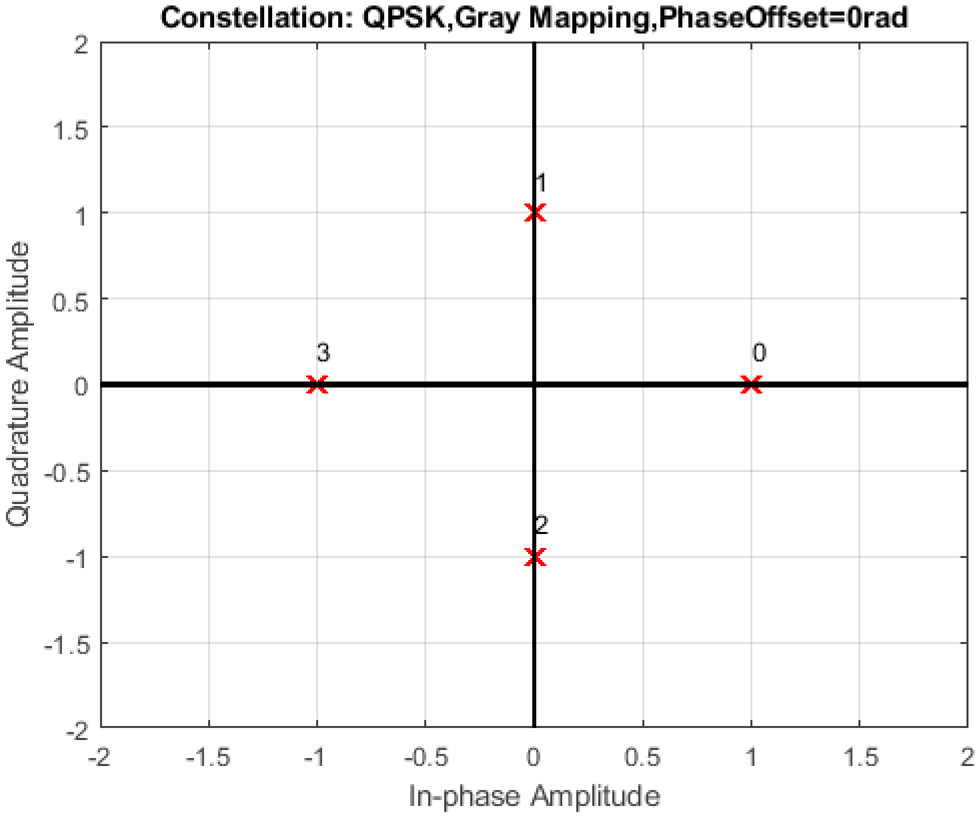


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



Phase Noise on QPSK Signal

Create a QPSK modulator object and a phase noise object.

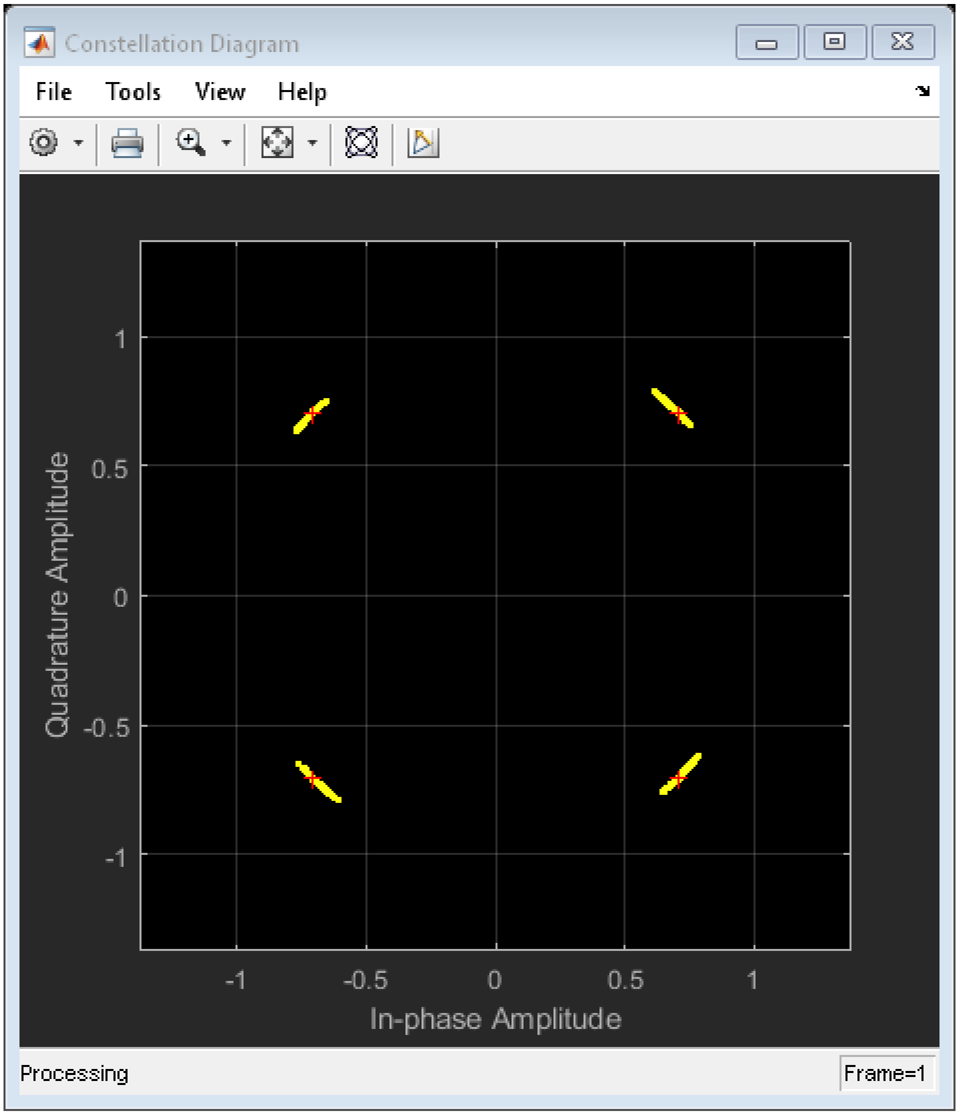
```
qpskModulator = comm.QPSKModulator;
phNoise = comm.PhaseNoise('Level',-55,'FrequencyOffset',20,'SampleRate',1000);
```

Generate random QPSK data. Pass the signal through the phase noise object.

```
d = randi([0 3],1000,1);
x = qpskModulator(d);
y = phNoise(x);
```

Display the constellation diagram of the QPSK signal. The phase noise has introduced a rotational distortion on the constellation diagram.

```
constDiagram = comm.ConstellationDiagram;

constDiagram(y)
```



Create QPSK Modulator Object with Bit Input

Create QPSK modulator object setting the `BitInput` property to `true`. Display the properties.

```
qpskmod = comm.QPSKModulator('BitInput',true)
```

```
qpskmod = 
  comm.QPSKModulator with properties:

        PhaseOffset: 0.7854
           BitInput: true
      SymbolMapping: 'Gray'
     OutputDataType: 'double'
```

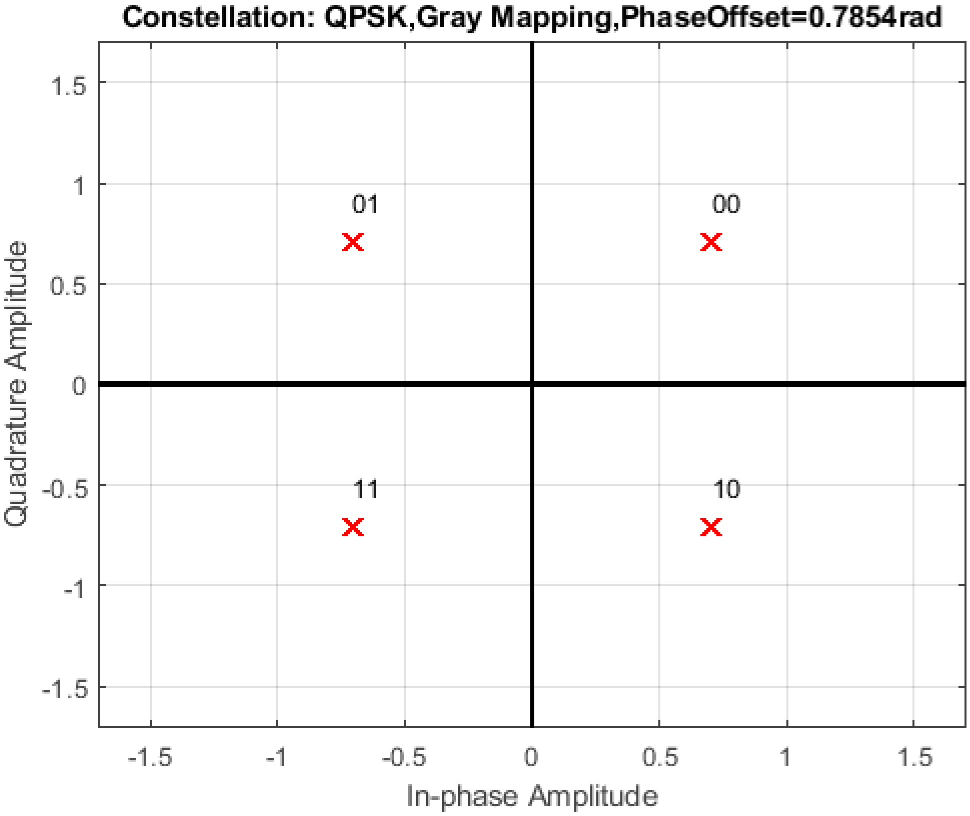
Determine the reference constellation points.

```
refC = constellation(qpskmod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation. Since `BitInput` is `true`, the constellation symbols are label with bit values.

```
constellation(qpskmod)
```



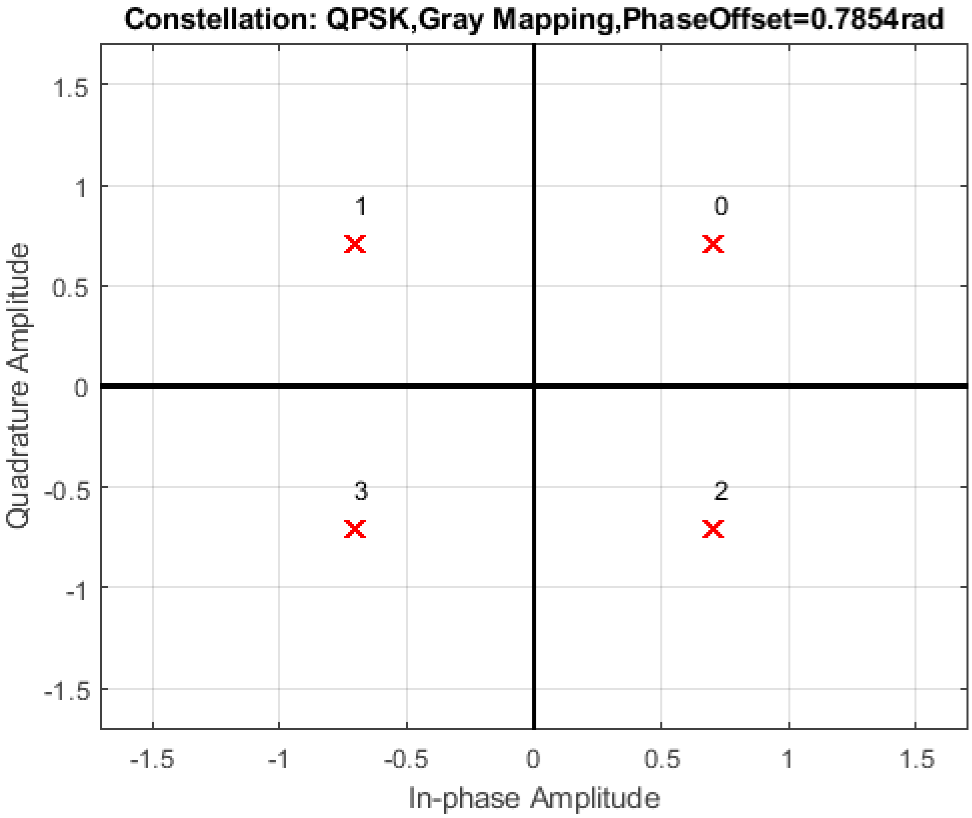
Create QPSK modulator object with default properties settings. Display the properties.

```
qpskmod2 = comm.QPSKModulator
```

```
qpskmod2 = 
  comm.QPSKModulator with properties:

        PhaseOffset: 0.7854
           BitInput: false
      SymbolMapping: 'Gray'
     OutputDataType: 'double'
```

Plot constellation with default settings. Since `BitInput` is `false`, the constellation symbols are label with integer values.

```
constellation(qpskmod2)
```



Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
`pskmod`

Objects
`comm.PSKModulator` | `comm.QPSKDemodulator`

Blocks
QPSK Modulator Baseband

Introduced in R2012a

comm.RaisedCosineReceiveFilter

Package: `comm`

Apply pulse shaping by decimating signal using raised-cosine FIR filter

Description

The `comm.RaisedCosineReceiveFilter` System object applies pulse shaping by decimating an input signal using a raised-cosine finite impulse response (FIR) filter. The FIR filter has (`FilterSpanInSymbols` x `InputSamplesPerSymbol` + 1) tap coefficients.

To apply pulse shaping by decimating an input signal using a raised-cosine FIR filter:

1. Create the `comm.RaisedCosineReceiveFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
rxfilter = comm.RaisedCosineReceiveFilter
rxfilter = comm.RaisedCosineReceiveFilter(Name,Value)
```

Description

`rxfilter = comm.RaisedCosineReceiveFilter` returns a raised-cosine FIR receive filter System object, which decimates the input signal using a raised-cosine FIR filter. The filter uses an efficient polyphase FIR decimation structure and has unit energy.

`rxfilter = comm.RaisedCosineReceiveFilter(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `comm.RaisedCosineReceiveFilter('RolloffFactor',0.3)` configures a raised-cosine receive filter System object with the roll-off factor set to `0.3`.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Shape — Filter shape
`Square root` (default) | `Normal`

Filter shape, specified as 'Square root' or 'Normal'.

Data Types: char | string

RolloffFactor — Roll-off factor
0.2 (default) | scalar in the range [0, 1]

Roll-off factor, specified as a scalar in the range [0, 1].

Data Types: double

FilterSpanInSymbols — Filter span in symbols
10 (default) | positive integer

Filter span in symbols, specified as a positive integer. The object truncates the infinite impulse response (IIR) of an ideal raised-cosine filter to an impulse response that spans the number of symbols specified by this property.

Data Types: double

InputSamplesPerSymbol — Input samples per symbol
8 (default) | positive integer

Input samples per symbol, specified as a positive integer.

Data Types: double

DecimationFactor — Decimation factor
8 (default) | integer

Decimation factor, specified as an integer in the range [1, InputSamplesPerSymbol]. This value must evenly divide into InputSamplesPerSymbol. The sampling rate of the output signal is reduced by the decimation factor such that length(y)/length(x) is equal to DecimationFactor. For a matrix input signal, the number of input rows must be a multiple of the decimation factor.

Data Types: double

DecimationOffset — Decimation offset
0 (default) | integer

Decimation offset, specified as an integer in the range [0, (DecimationFactor − 1)]. This property specifies the number of filtered samples the object discards before downsampling.

Data Types: double

Gain — Linear filter gain
1 (default) | positive scalar

Linear filter gain, specified as a positive scalar. The object designs a raised-cosine filter that has unit energy and then applies the linear filter gain to obtain final tap coefficient values.

Data Types: double

Usage

Syntax

```
y = rxfilter(x)
```

Description

`y = rxfilter(x)` applies pulse shaping by decimating an input signal using a raised cosine FIR filter. The output consists of decimated signal values.

Input Arguments

x — Input signal
column vector | matrix

Input signal, specified as a column vector or a K_i-by-N matrix. K_i is the number of input samples per signal channel, and N is the number of signal channels.

For a K_i-by-N matrix input, the object processes columns of the input matrix as N independent channels.

Data Types: `double` | `single`

Output Arguments

y — Output signal
column vector | matrix

Output signal, returned as a column vector or a K_o-by-N matrix. K_o is equal to K_i / `DecimationFactor`. K_i is the number of input samples per signal channel, and N is the number of signal channels.

The System object filters each channel over time and generates a K_o-by-N output matrix. The output signal is the same data type as the input signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.RaisedCosineReceiveFilter`

coeffs	Coefficients for filters
info	Information about filter System object
order	Order of discrete-time filter System object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics

reset	Reset internal states of System object

Examples

Filter Signal Using Square-Root-Raised-Cosine Receive Filter

Filter the output of a square-root-raised-cosine (SRRC) transmit filter by using a matched SRRC receive filter. The input signal has eight samples per symbol.

Create an SRRC transmit filter object, setting the number of output samples per symbol to 8.

```
 txfilter = comm.RaisedCosineTransmitFilter('OutputSamplesPerSymbol',8);
```

Create an SRRC receive filter, setting the number of input samples per symbol to 8 and the decimation factor to 8.

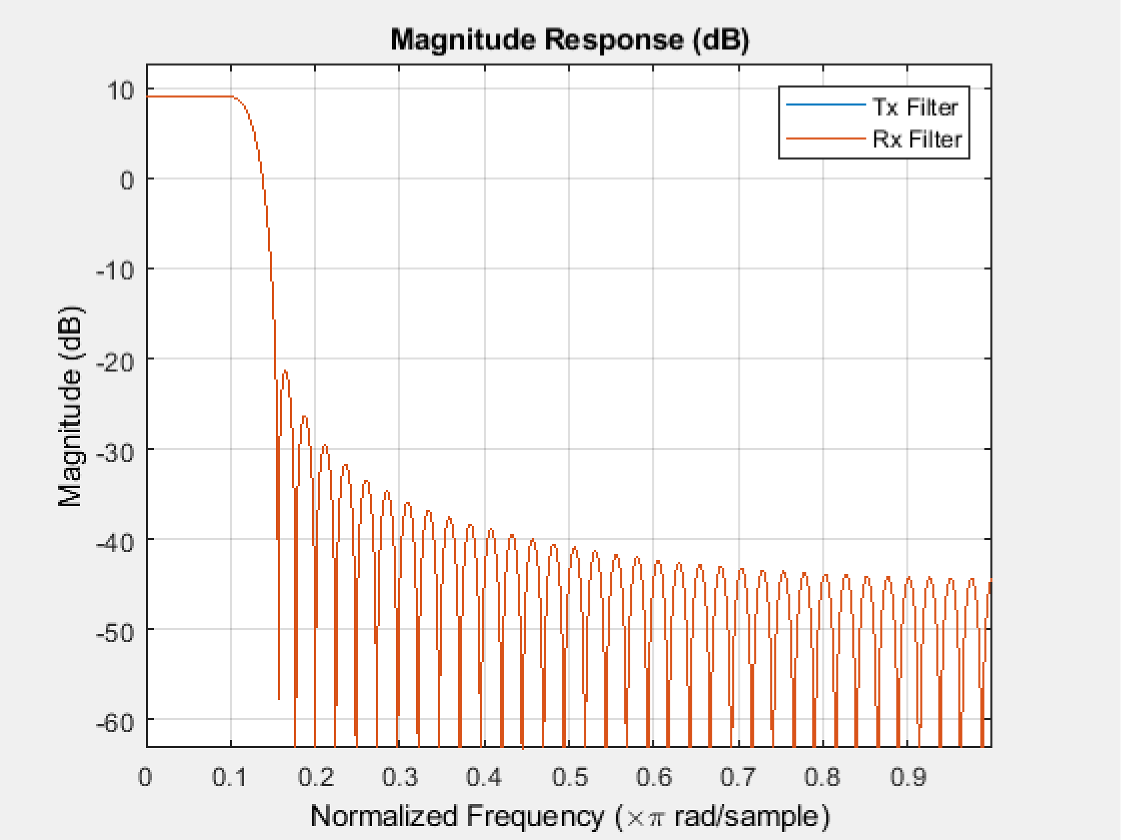
```
rxfilter = comm.RaisedCosineReceiveFilter('InputSamplesPerSymbol',8, ...
    'DecimationFactor',8);
```

Use the `coeffs` function to determine the filter coefficients for both filters.

```
txCoef = coeffs(txfilter);
rxCoef = coeffs(rxfilter);
```

Launch the filter visualization tool and display the magnitude responses of the two filters. The results show that the responses are the same.

```
 fvtool(txCoef.Numerator,1,rxCoef.Numerator,1);
 legend('Tx Filter','Rx Filter')
```



Generate a random bipolar signal. Interpolate the signal by using the SRRC transmit filter object.

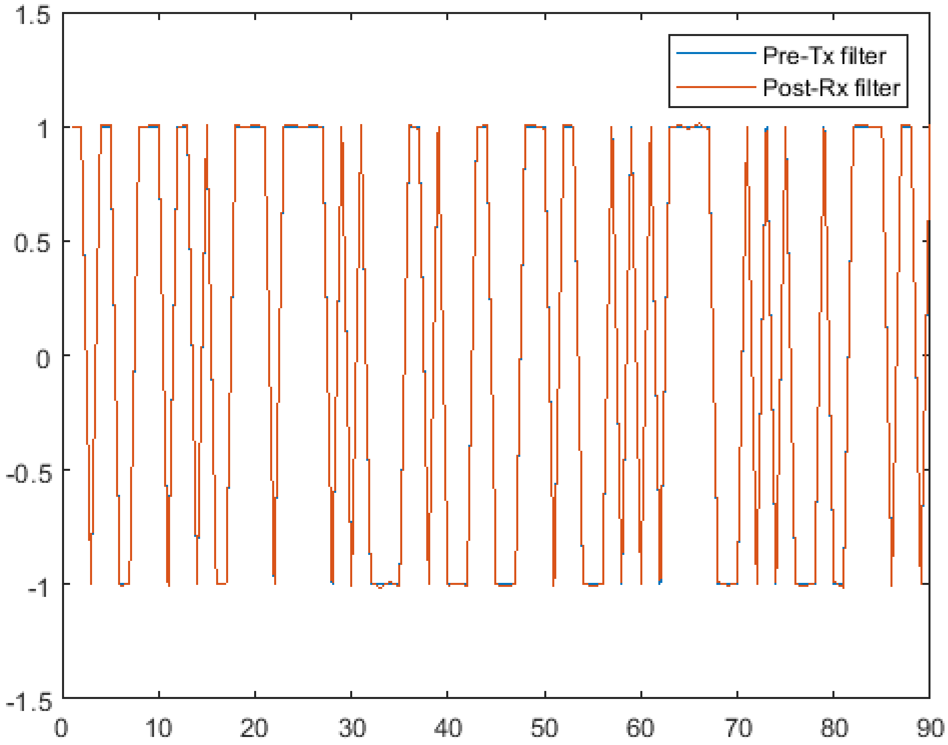
```
preTx = 2*randi([0 1],100,1) - 1;
y = txfilter(preTx);
```

Decimate the signal by using the SRRC receive filter object.

```
postRx = rxfilter(y);
```

The filter delay is equal to the filter span. Accounting for the filter delay, adjust the plotted samples to compare the pre-Tx filter signal with the post-Rx filter signal. Because the combined receive and the transmit RRC filters generate a matched filter pair, the two signals overlap one another.

```
delay = txfilter.FilterSpanInSymbols;
x = (1:(length(preTx)-delay));
plot(x,preTx(1:end-delay),x,postRx(delay+1:end))
legend('Pre-Tx filter','Post-Rx filter')
```



Specify Filter Span of Square-Root-Raised-Cosine Receive Filter

Decimate a bipolar signal using a square-root-raised-cosine (SRRC) filter whose impulse response is truncated to filter a span of six symbol durations.

Create a SRRC transmit FIR filter, setting the filter span to six symbols. The object truncates the impulse response to six symbols.

```
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',6);
```

Generate a random bipolar signal . Filter the signal by using the SRRC transmit FIR filter object.

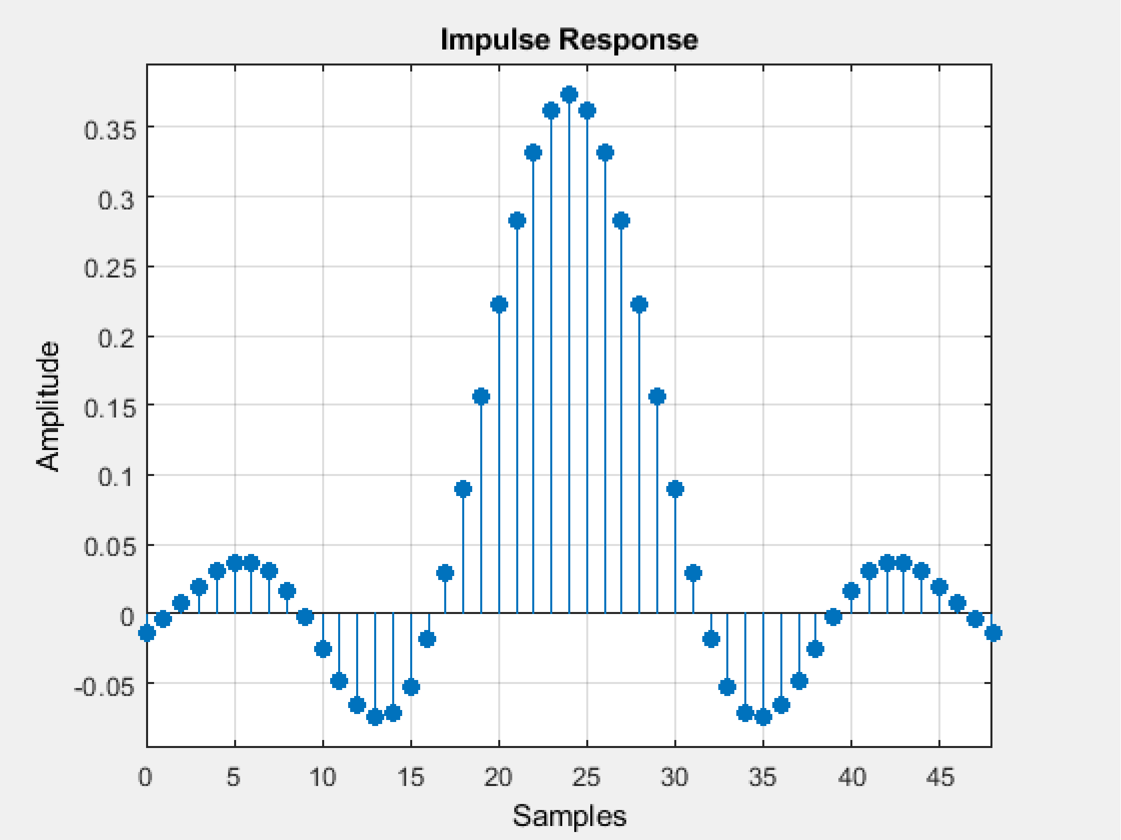
```
x = 2*randi([0 1],25,1) - 1;
y = txfilter(x);
```

Create a matched SRRC receive filter object.

```
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',6);
```

Launch the filter visualization tool to show the impulse response of the receive filter.

```
fvtool(rxfilter,'Analysis','impulse')
```

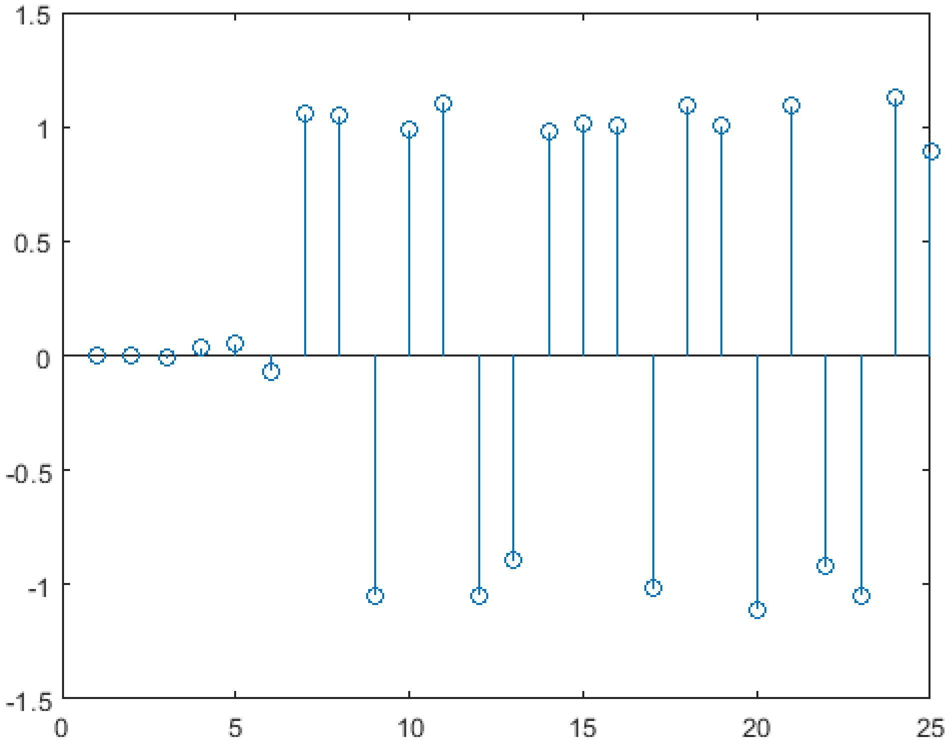


Filter the output signal from the transmit filter by using the matched SRRC receive filter object.

```
r = rxfilter(y);
```

Plot the interpolated signal. The results show a delay equal to the filter span (six symbols) before data passes through the filter.

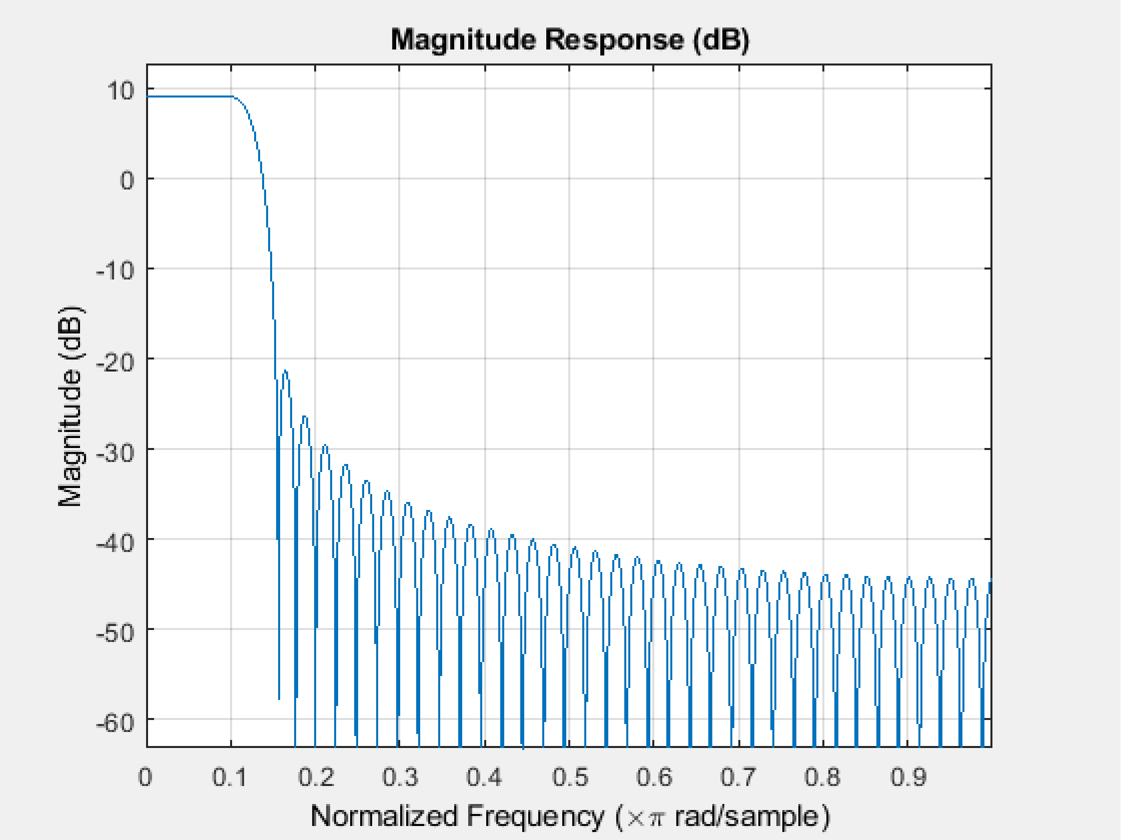
```
stem(r)
```



Create Square-Root-Raised-Cosine Receive Filter with Unity Passband Gain

Create a square-root-raised-cosine (SRRC) receive filter object. Use FVTool to plot the filter response. The results show that the linear filter gain is greater than unity. Specifically, the passband gain is more than 0 dB.

```
rxfilter = comm.RaisedCosineReceiveFilter;
fvtool(rxfilter)
```



Use the `coeffs` object function to obtain the filter coefficients and adjust the filter gain to unit energy.

```
b = coeffs(rxfilter);
```

Because a filter with unity passband gain must have filter coefficients that sum to 1, set the linear filter gain to the inverse of the sum of the filter tap coefficients, `b.Numerator`.

```
rxfilter.Gain = 1/sum(b.Numerator);
```

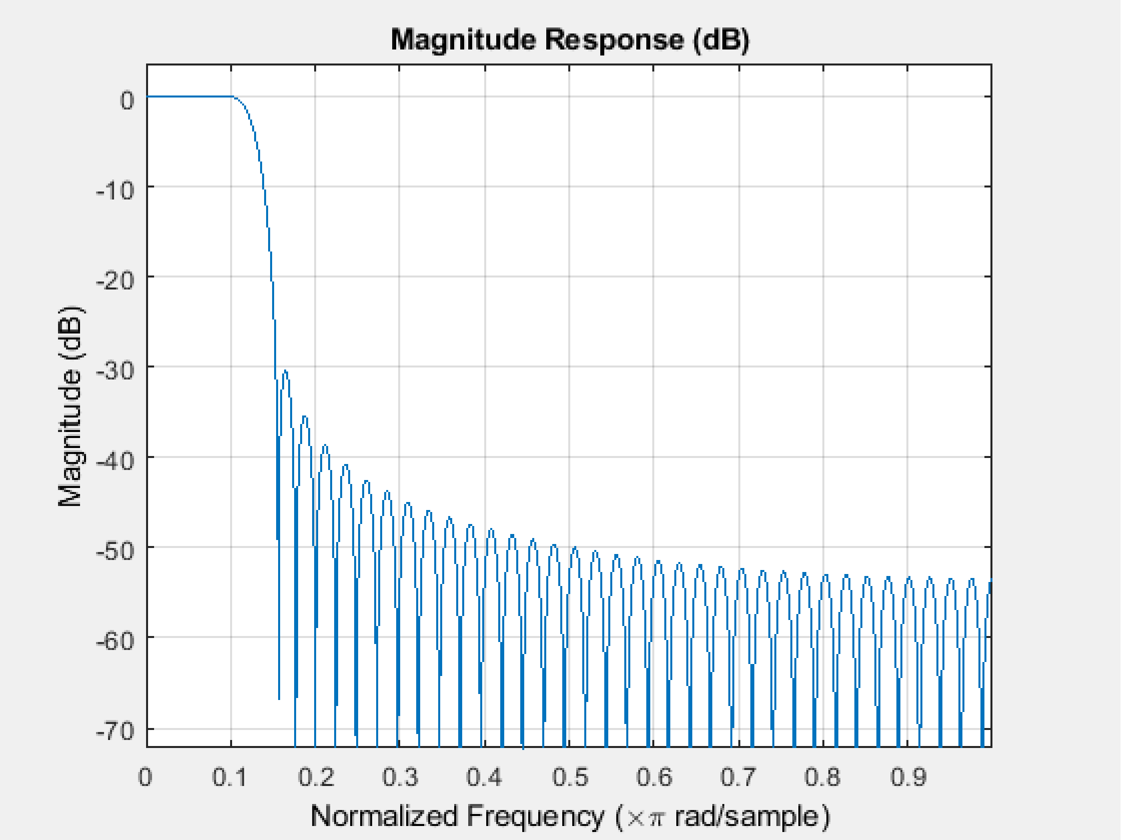
Verify that the resulting filter coefficients sum to 1.

```
bNorm = coeffs(rxfilter);
sum(bNorm.Numerator)
```

```
ans = 1.0000
```

Plot the filter frequency response. The results now show that the passband gain is 0 dB, which is unity gain.

```
fvtool(rxfilter)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

See Also

Objects

`comm.RaisedCosineTransmitFilter`

Functions

`rcosdesign`

Introduced in R2013b

comm.RaisedCosineTransmitFilter

Package: `comm`

Apply pulse shaping by interpolating signal using raised-cosine FIR filter

Description

The `comm.RaisedCosineTransmitFilter` System object applies pulse shaping by interpolating an input signal using a raised cosine finite impulse response (FIR) filter. The FIR filter has (`FilterSpanInSymbols` x `OutputSamplesPerSymbol` + 1) tap coefficients.

To apply pulse shaping by interpolating an input signal using a raised cosine FIR filter:

1. Create the `comm.RaisedCosineTransmitFilter` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
txfilter = comm.RaisedCosineTransmitFilter
txfilter = comm.RaisedCosineTransmitFilter(Name,Value)
```

Description

`txfilter = comm.RaisedCosineTransmitFilter` returns a raised cosine transmit FIR filter System object, which interpolates an input signal using a raised cosine FIR filter. The filter uses an efficient polyphase FIR interpolation structure and has unit energy.

`txfilter = comm.RaisedCosineTransmitFilter(Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, `comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',15)` configures a raised cosine transmit filter System object with the filter span set to 15 symbols.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Shape — Filter shape
`'Square root'` (default) | `'Normal'`

Filter shape, specified as `'Square root'` or `'Normal'`.

Data Types: `char` | `string`

`RolloffFactor` — Roll-off factor

`0.2` (default) | scalar in the range [0, 1]

Roll-off factor, specified as a scalar in the range [0, 1].

Data Types: `double`

`FilterSpanInSymbols` — Filter span in symbols

`10` (default) | positive integer

Filter span in symbols, specified as a positive integer. The object truncates the infinite impulse response (IIR) of an ideal raised-cosine filter to an impulse response that spans the number of symbols specified by this property.

Data Types: `double`

`OutputSamplesPerSymbol` — Output samples per symbol

`8` (default) | positive integer

Output samples per symbol, specified as a positive integer.

Data Types: `double`

`Gain` — Linear filter gain

`1` (default) | positive scalar

Linear filter gain, specified as a positive scalar. The object designs a raised-cosine filter that has unit energy and then applies the linear filter gain to obtain final tap coefficient values.

Data Types: `double`

Usage

Syntax

```
y = txfilter(x)
```

Description

`y = txfilter(x)` applies pulse shaping by interpolating an input signal using a raised cosine FIR filter. The output consists of interpolated signal values.

Input Arguments

`x` — Input signal

column vector | matrix

Input signal, specified as a column vector or a K_i-by-N matrix. K_i is the number of input samples per signal channel, and N is the number of signal channels.

For a K_i-by-N matrix input, the object processes columns of the input matrix as N independent channels.

Data Types: `double` | `single`

Output Arguments

y — Output signal
column vector | matrix

Output signal, returned as a column vector or a K_o-by-N matrix. K_o is equal to K_i × `OutputSamplesPerSymbol`. K_i is the number of input samples per signal channel, and N is the number of signal channels.

The object interpolates and filters each channel over the first dimension and then generates a K_o-by-N output matrix. The output signal is the same data type as the input signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.RaisedCosineTransmitFilter`

coeffs	Coefficients for filters
info	Information about filter System object
order	Order of discrete-time filter System object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Interpolate Signal Using Square-Root-Raised-Cosine Filter

Interpolate a signal using square-root-raised-cosine (SRRC) transmit filter object and display the spectrum of the filtered signal.

Create random bipolar symbols at a symbol rate of 1e6 symbols per second.

```
data = 2*randi([0 1],1e6,1) - 1;
```

Create a SRRC transmit filter object. The default sets the filter to a square-root shape and the number of samples per symbol to 8.

```
txfilter = comm.RaisedCosineTransmitFilter
```

```
txfilter = 
  comm.RaisedCosineTransmitFilter with properties:

                   Shape: 'Square root'
```

```
            RolloffFactor: 0.2000
        FilterSpanInSymbols: 10
      OutputSamplesPerSymbol: 8
                       Gain: 1
```

Filter the data by using the SRRC filter.

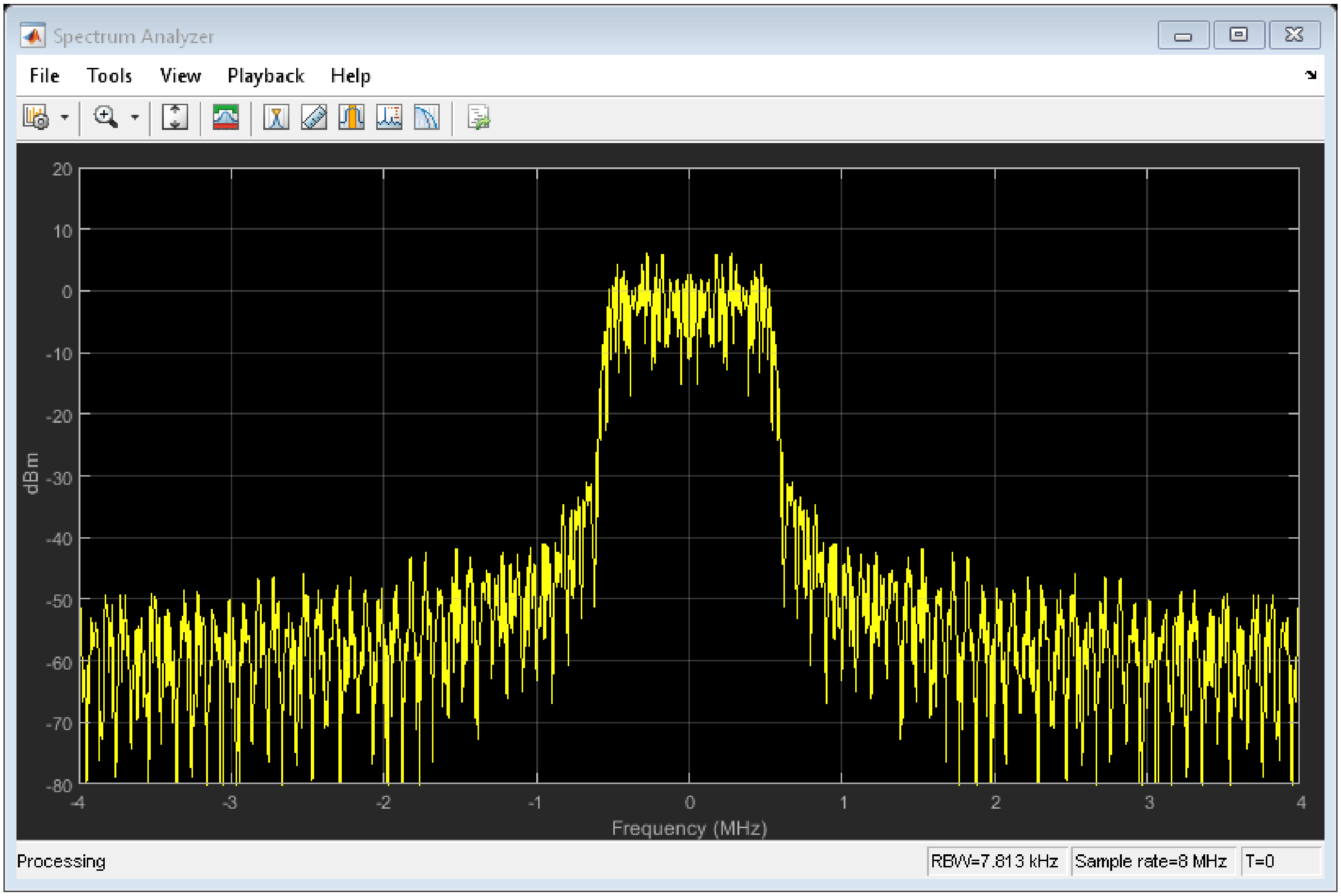
```
filteredData = txfilter(data);
```

Create a spectrum analyzer object with an 8e6 sampling rate. This sampling rate matches the sampling rate of the filtered signal.

```
spectrumAnalyzer = dsp.SpectrumAnalyzer('SampleRate',8e6);
```

View the spectrum of the filtered signal by using the spectrum analyzer object.

```
spectrumAnalyzer(filteredData)
```



Examine Effect of Filter Span on Magnitude Response

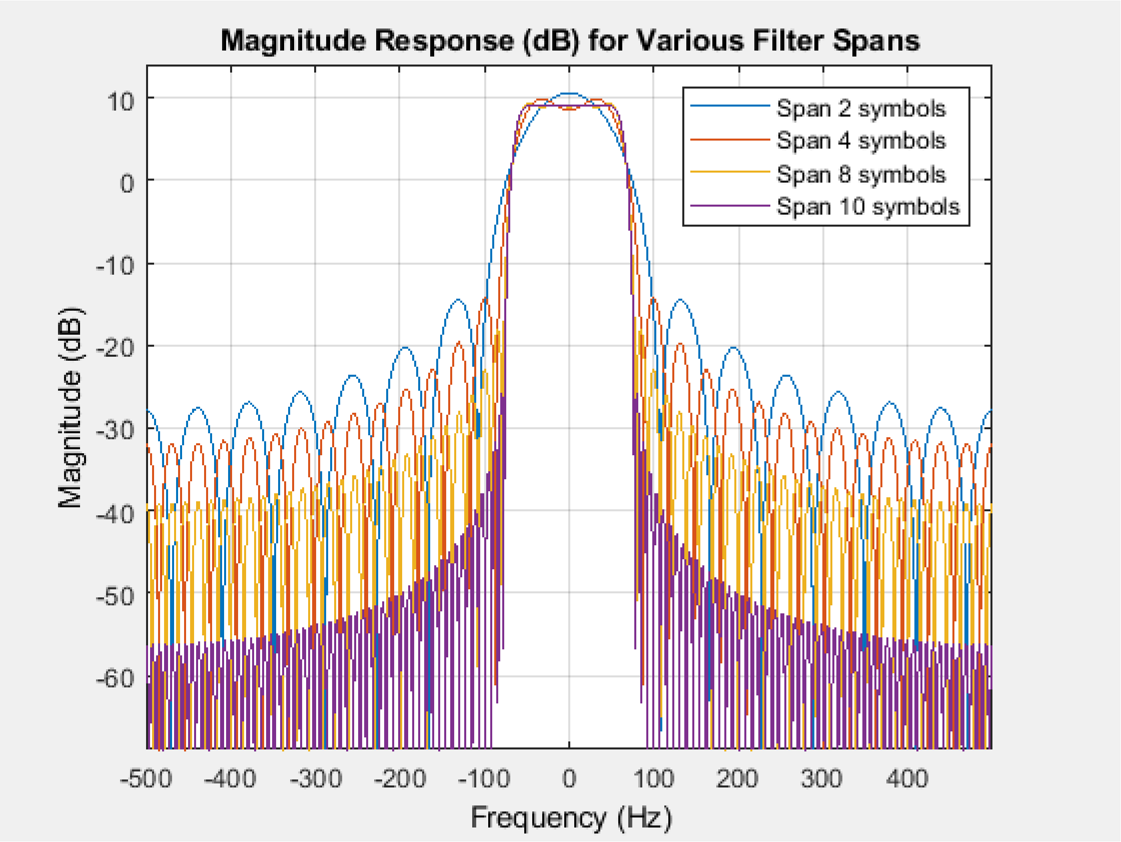
Create interpolated signals from a square-root-raised-cosine (SRRC) filter with various filter spans. Examine the magnitude response of the various filter designs.

Create SRRC filter objects setting various filter spans. Use the `coeffs` object function to obtain the filter coefficients.

```
txfilt2 = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',2);
txfilt4 = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',4);
txfilt6 = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',6);
txfilt8 = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',8);
txfilt16 = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',16);

taps2 = coeffs(txfilt2).Numerator;
taps4 = coeffs(txfilt4).Numerator;
taps6 = coeffs(txfilt6).Numerator;
taps8 = coeffs(txfilt8).Numerator;
taps16 = coeffs(txfilt16).Numerator;
```

Launch the filter visualization tool to show the impulse response. Specify a sample rate of 1 kHz. Display the two-sided centered response.

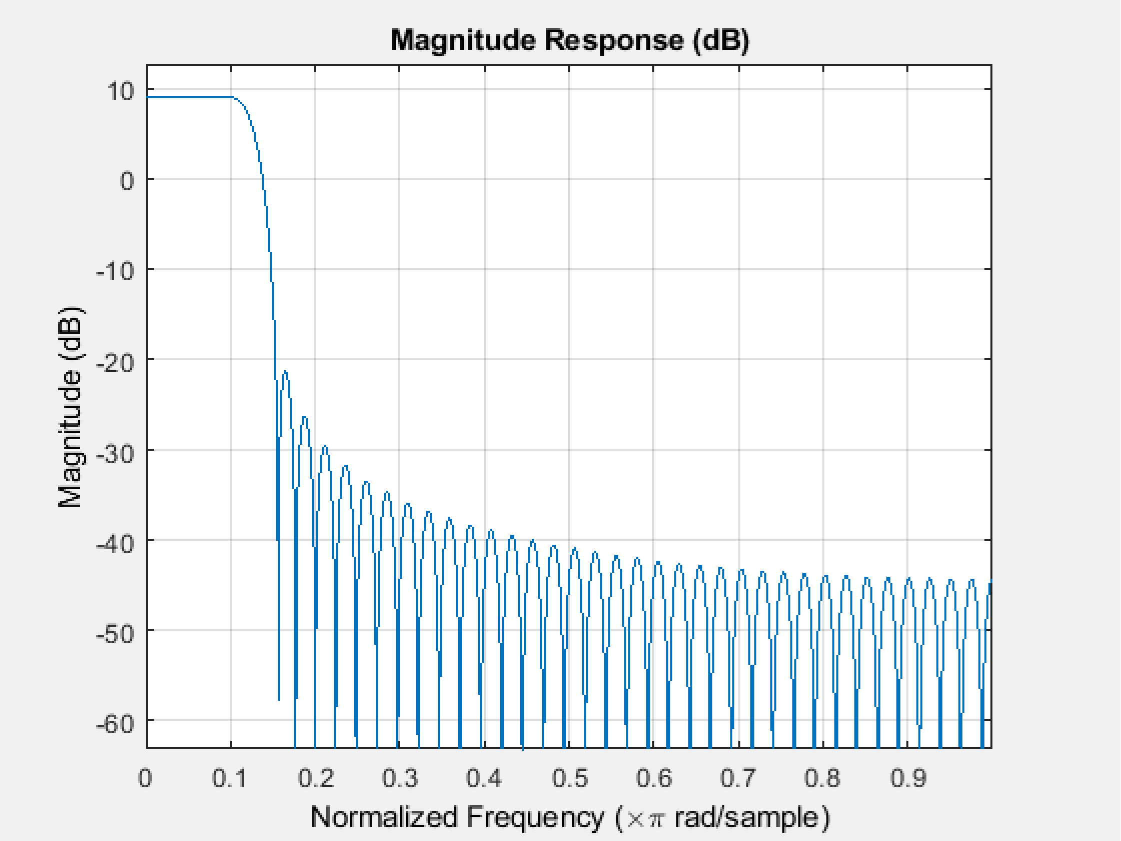
```
h = fvtool(taps2,1,taps4,1,taps8,1,taps16,1);
h.Fs = 1e3;
h.FrequencyRange = '[-Fs/2, Fs/2)';
legend('Span 2 symbols','Span 4 symbols','Span 8 symbols','Span 10 symbols')
title('Magnitude Response (dB) for Various Filter Spans')
```



Create Square-Root-Raised-Cosine Transmit Filter with Unity Passband Gain

Create a square-root-raised-cosine (SRRC) transmit filter object. Use FVTool to plot the filter response. The results show that the linear filter gain is greater than unity. Specifically, the passband gain is more than 0 dB.

```
txfilter = comm.RaisedCosineTransmitFilter;
fvtool(txfilter)
```



Use the `coeffs` object function to obtain the filter coefficients and adjust the filter gain to unit energy.

```
b = coeffs(txfilter);
```

Because a filter with unity passband gain must have filter coefficients that sum to 1, set the linear filter gain to the inverse of the sum of the filter tap coefficients, `b.Numerator`.

```
txfilter.Gain = 1/sum(b.Numerator);
```

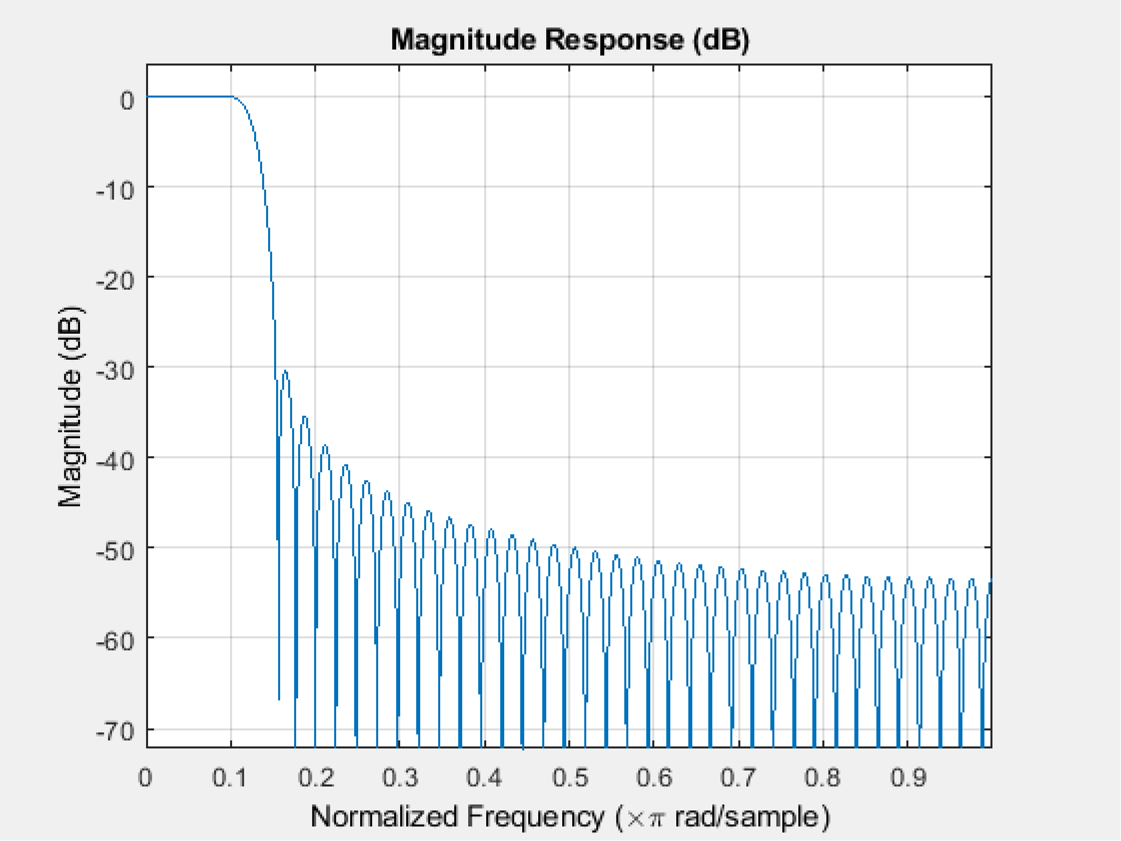
Verify that the resulting filter coefficients sum to 1.

```
bNorm = coeffs(txfilter);
sum(bNorm.Numerator)
```

```
ans = 1.0000
```

Plot the filter frequency response. The results now show that the passband gain is 0 dB, which is unity gain.

```
fvtool(txfilter)
```



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.RaisedCosineReceiveFilter`

Functions

`rcosdesign`

Introduced in R2013b

comm.RayleighChannel

Package: comm

Filter input signal through multipath Rayleigh fading channel

Description

The comm.RayleighChannel System object filters an input signal through a multipath Rayleigh fading channel. For more information on fading model processing, see Methodology for Simulating Multipath Fading Channels.

To filter an input signal using a multipath Rayleigh fading channel:

1. Create the comm.RayleighChannel object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
rayleighchan = comm.RayleighChannel
rayleighchan = comm.RayleighChannel(Name,Value)
```

Description

rayleighchan = comm.RayleighChannel creates a frequency-selective or frequency-flat multipath Rayleigh fading channel System object. This object filters a real or complex input signal through the multipath channel to obtain the channel-impaired signal.

rayleighchan = comm.RayleighChannel(Name,Value) sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, comm.RayleighChannel('SampleRate',2) sets the input signal sample rate to 2.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SampleRate — Input signal sample rate
1 (default) | positive scalar

Input signal sample rate in Hz, specified as a positive scalar.

Data Types: `double`

`PathDelays` — Discrete path delay

`0` (default) | scalar | row vector

Discrete path delay in seconds, specified as a scalar or row vector.

- When `PathDelays` is a scalar, the channel is frequency flat.
- When `PathDelays` is a vector, the channel is frequency selective.

Data Types: `double`

`AveragePathGains` — Average of the discrete paths

`0` (default) | scalar | row vector

Average gains of the discrete paths in decibels, specified as a scalar or row vector. `AveragePathGains` must be of the same size as the `PathDelays` property.

Data Types: `double`

`NormalizePathGains` — Normalize average path gains to 0 dB

`true` or `1` (default) | `false` or `0`

Normalize average path gains to 0 dB, specified as a logical `1` (`true`) or `0` (`false`).

- When `NormalizePathGains` is `true`, the fading processes are normalized so that the total power of the path gains, averaged over time, is 0 dB.
- When `NormalizePathGains` is `false`, the total power of the path gains is not normalized.

Data Types: `logical`

`MaximumDopplerShift` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths, specified as a nonnegative scalar. Units are in hertz.

The maximum Doppler shift limit applies to each channel path. When you set this property to `0`, the channel remains static for the entire input. You can use the `reset` object function to generate a new channel realization. The `MaximumDopplerShift` property value must be smaller than `SampleRate`/10/f_c for each path. f_c represents the cutoff frequency factor of the path. For most Doppler spectrum types, the value of f_c is 1. For Gaussian and biGaussian Doppler spectrum types, f_c is dependent on the Doppler spectrum structure fields. For more details about how f_c is defined, see “Cutoff Frequency Factor” on page 3-1218.

Data Types: `double`

`DopplerSpectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | Doppler spectrum structure | 1-by-N_P cell array of Doppler spectrum structures

Doppler spectrum shape for all channel paths, specified as a Doppler spectrum structure or a 1-by-N_P cell array of Doppler spectrum structures. These Doppler spectrum structures must be outputs of the form returned from the `doppler` function. N_P is the number of discrete delay paths, that is, the length of the `PathDelays` property.

- When DopplerSpectrum is defined by a single Doppler spectrum structure, all paths have the same specified Doppler spectrum.
- When DopplerSpectrum is defined by a cell array of Doppler spectrum structures, each path has the Doppler spectrum specified by the corresponding structure in the cell array.

Options for the spectrum type are specified by the specType input to the doppler function. If you set the FadingTechnique property to 'Sum of sinusoids', you must set DopplerSpectrum to doppler('Jakes').

Dependencies

To enable this property, set the MaximumDopplerShift property to a positive value. The MaximumDopplerShift property defines the maximum Doppler shift value permitted when specifying the Doppler spectrum.

Data Types: struct | cell

FadingTechnique — Channel model fading technique

'Filtered Gaussian noise' (default) | 'Sum of sinusoids'

Channel model fading technique, specified as 'Filtered Gaussian noise' or 'Sum of sinusoids'.

Data Types: char | string

NumSinusoids — Number of sinusoids used

48 (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

To enable this property, set the FadingTechnique property to 'Sum of sinusoids'.

Data Types: double

InitialTimeSource — Source to control start time of fading process

'Property' (default) | 'Input port'

Source to control the start time of the fading process, specified as 'Property' or 'Input port'.

- When InitialTimeSource is set to 'Property', use the InitialTime property to set the initial time offset.
- When InitialTimeSource is set to 'Input port', specify the start time of the fading process by using the itime input to the System object. The input value can change in consecutive calls to the System object.

Dependencies

To enable this property, set the FadingTechnique property to 'Sum of sinusoids'.

Data Types: string | char

InitialTime — Initial time offset

0 (default) | nonnegative scalar

Initial time offset, in seconds, for the fading model, specified as a nonnegative scalar. `InitialTime` must be greater than end time of the last frame. `InitialTime` is rounded up to the nearest sample position when it is not a multiple of 1/`SampleRate`.

Dependencies

To enable this property, set the `FadingTechnique` property to `'Sum of sinusoids'` and the `InitialTimeSource` property to `'Property'`.

Data Types: `double`

`RandomStream` — Source of random number stream

`'Global stream'` (default) | `'mt19937ar with seed'`

Source of random number stream, specified as `'Global stream'` or `'mt19937ar with seed'`.

- `'Global stream'` — The current global random number stream is used for normally distributed random number generation. In this case, the `reset` object function resets the channel filters only.
- `'mt19937ar with seed'` — The mt19937ar algorithm is used for normally distributed random number generation. In this case, the `reset` object function resets the channel filters and reinitializes the random number stream to the value of the `Seed` property.

Data Types: `string` | `char`

Seed — Initial seed of mt19937ar random number stream

`73` (default) | nonnegative integer

Initial seed of mt19937ar random number stream generator algorithm, specified as a nonnegative integer. When the `reset` object function is called, the mt19937ar random number stream is reinitialized to the `Seed` property value.

Dependencies

To enable this property, set the `RandomStream` property to `'mt19937ar with seed'`.

Data Types: `double`

`PathGainsOutputPort` — Option to output path gains

`false` or `0` (default) | `true` or `1`

Option to output path gains, specified as `0` (`false`) or `1` (`true`). Set this property to `1` (`true`) to output the channel path gains of the underlying fading process.

Data Types: `logical`

`Visualization` — Channel visualization

`'Off'` (default) | `'Impulse response'` | `'Frequency response'` | `'Impulse and frequency responses'` | `'Doppler spectrum'`

Channel visualization, specified as `'Off'`, `'Impulse response'`, `'Frequency response'`, `'Impulse and frequency responses'`, or `'Doppler spectrum'`. For more information, see “Channel Visualization” on page 3-1219.

Data Types: `string` | `char`

`SamplesToDisplay` — Percentage of samples to display

`'25%'` (default) | `'10%'` | `'50%'` | `'100%'`

Percentage of samples to display, specified as '25%', '10%', '50%', or '100%'. Displaying fewer samples improves (decreases) the display update rate at the expense of decreasing the visualized precision.

Dependencies

To enable this property, set the Visualization property to 'Impulse response', 'Frequency response', or 'Impulse and frequency responses'.

Data Types: string | char

PathsForDopplerDisplay — Path for Doppler display

1 (default) | positive integer

Path for Doppler display, specified as a positive integer. Use this property to select the discrete path used in constructing a Doppler spectrum plot. The specified path must be an element of {1, 2, ..., N_P}. In this set, N_P is the number of discrete delay paths, that is, the length of the PathDelays property value.

Dependencies

To enable this property, set the Visualization property to 'Doppler spectrum'.

Data Types: double

Usage

Syntax

```
y = rayleighchan(x)
[y,pathgains] = rayleighchan(x)
___ = rayleighchan(x,itime)
```

Description

y = rayleighchan(x) filters input signal x through a multipath Rayleigh fading channel and returns the output signal in y.

[y,pathgains] = rayleighchan(x) returns the channel path gains of the underlying multipath Rayleigh fading process in pathgains. To enable this syntax set the PathGainsOutputPort property set to 1 (true).

___ = rayleighchan(x,itime) passes data through the multipath Rayleigh fading channel beginning at the initial time specified by itime. To enable this syntax, set the FadingTechnique property to 'Sum of sinusoids' and the InitialTimeSource property to 'Input port'.

Input Arguments

x — Input signal

N_S-by-1 vector

Input signal, specified as an N_S-by-1 vector, where N_S is the number of samples.

Data Types: single | double
Complex Number Support: Yes

`itime` — Initial time

`0` | nonnegative scalar

Initial time in seconds, specified as a nonnegative scalar. The `itime` input must be greater than the end time of the last frame. When `itime` is not a multiple of 1/`SampleRate`, it is rounded up to the nearest sample position.

Data Types: `single` | `double`

Output Arguments

y — Output signal

N_S-by-1 vector

Output signal, returned as an N_S-by-1 vector of complex values with the same data precision as the input signal `x`. N_S is the number of samples.

`pathgains` — Path gains

N_S-by-N_P array

Path gains, returned as an N_S-by-N_P array. N_S is the number of samples. N_P is the number of discrete delay paths, that is, the length of the `PathDelays` property value. `pathgains` contains complex values with the same precision as the input signal `x`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.RayleighChannel`

info	Characteristic information about fading channel object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Produce Same Rayleigh Channel Outputs Using Two Random Number Generation Methods

This example shows how to produce the same multipath Rayleigh fading channel response by using two different methods for random number generation. The multipath Rayleigh fading channel System object™ includes two methods for random number generation. You can use the current global stream or the mt19937ar algorithm with a specified seed. By interacting with the global stream, the System object can produce the same outputs from these two methods.

Create a PSK modulator System object to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
channelInput = pskModulator(randi([0 pskModulator.ModulationOrder-1],1024,1));
```

Create a multipath Rayleigh fading channel System object, specifying the random number generation method as the my19937ar algorithm and the random number seed as 22.

```
rayleighchan = comm.RayleighChannel(...
    'SampleRate',10e3, ...
    'PathDelays',[0 1.5e-4], ...
    'AveragePathGains',[2 3], ...
    'NormalizePathGains',true, ...
    'MaximumDopplerShift',30, ...
    'DopplerSpectrum',{doppler('Gaussian',0.6),doppler('Flat')}, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',22, ...
    'PathGainsOutputPort',true);
```

Filter the modulated data by using the multipath Rayleigh fading channel System object.

```
[chanOut1,pathGains1] = rayleighchan(channelInput);
```

Set the System object to use the global stream for random number generation.

```
release(rayleighchan);
rayleighchan.RandomStream = 'Global stream';
```

Set the global stream to have the same seed that was specified when creating the multipath Rayleigh fading channel System object.

```
rng(22)
```

Filter the modulated data by using the multipath Rayleigh fading channel System object again.

```
[chanOut2,pathGains2] = rayleighchan(channelInput);
```

Verify that the channel and path gain outputs are the same for each of the two methods.

```
isequal(chanOut1,chanOut2)
```

```
ans = logical
   1
```

```
isequal(pathGains1,pathGains2)
```

```
ans = logical
   1
```

Display Impulse and Frequency Responses of Multipath Rayleigh Fading Channel

This example shows how to create a frequency-selective multipath Rayleigh fading channel System object and display its impulse and frequency responses.

Set the sample rate to 3.84 MHz. Specify path delays and gains by using the ITU pedestrian B channel configuration. Set the maximum Doppler shift to 50 Hz.

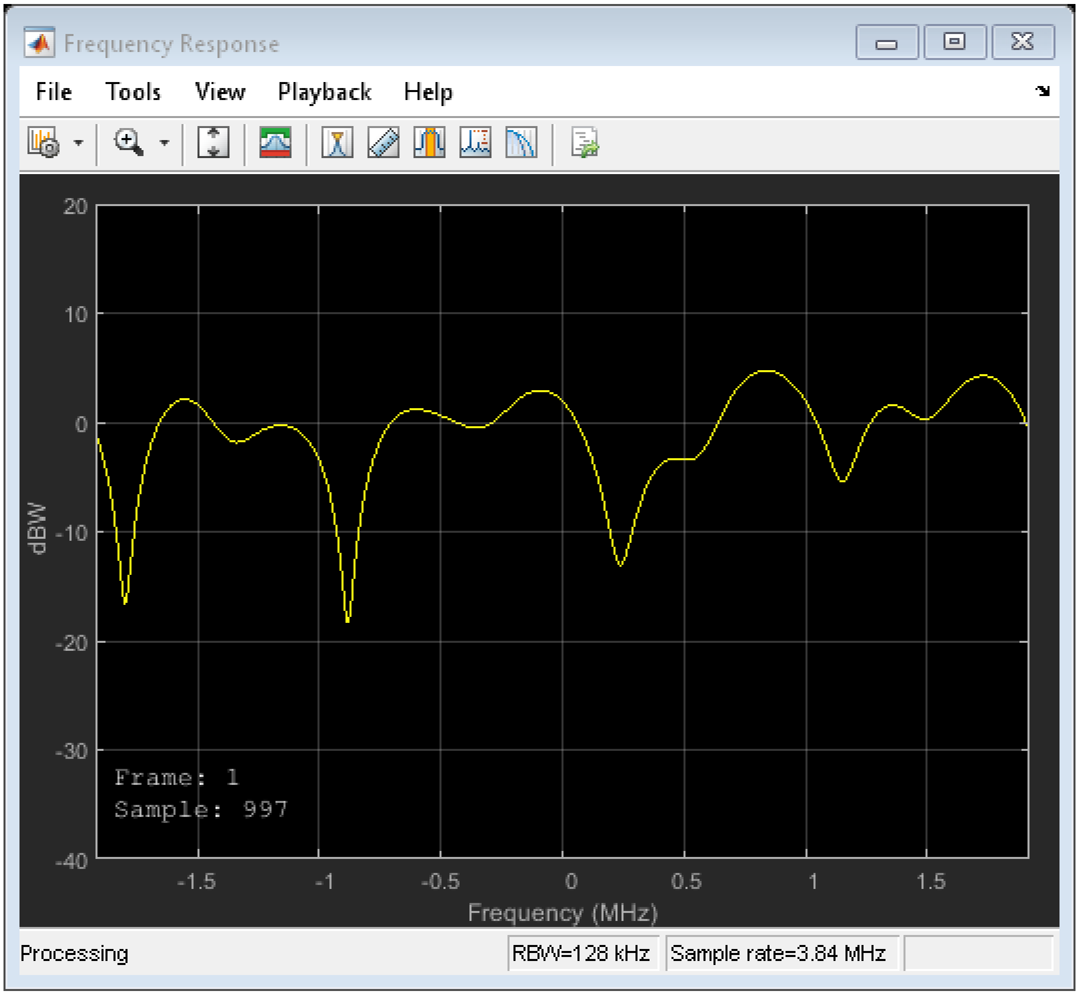
```
fs = 3.84e6; % Hz
pathDelays = [0 200 800 1200 2300 3700]*1e-9; % sec
avgPathGains = [0 -0.9 -4.9 -8 -7.8 -23.9]; % dB
fD = 50; % Hz
```

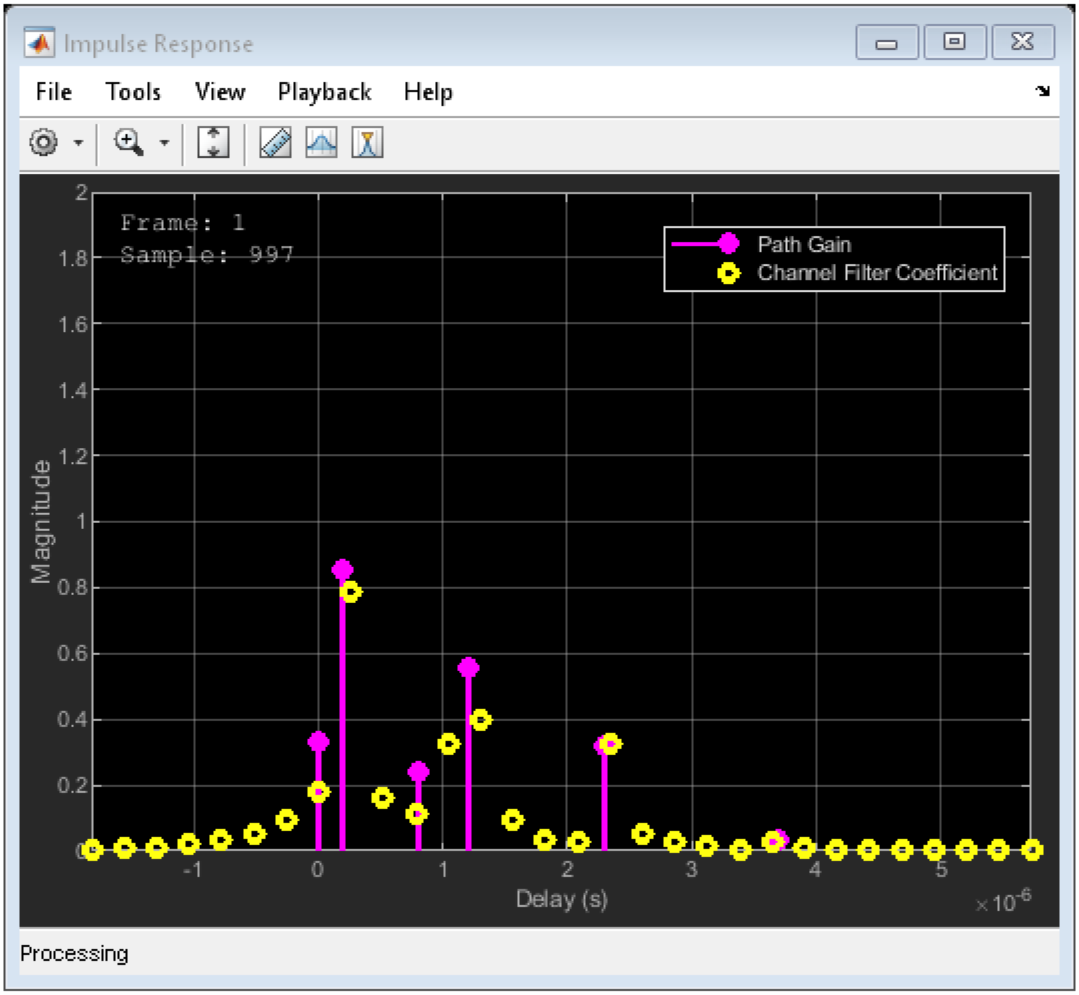
Create a multipath Rayleigh fading channel System object by using the previously defined properties and to visualize the impulse response and frequency response plots.

```
rayleighchan = comm.RayleighChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the multipath Rayleigh fading channel. The impulse response plot enables you to identify the individual paths and their corresponding filter coefficients. The frequency response plot shows the frequency-selective nature of the ITU pedestrian B channel.

```
x = randi([0 1],1000,1);
y = rayleighchan(x);
```





Model Multipath Rayleigh Fading Channel by Using Sum-of-Sinusoids Technique

The example shows how the channel state is maintained in cases in which data is discontinuously transmitted. Create a multipath Rayleigh fading channel System object and pass data through it using the sum-of-sinusoids technique.

Set the channel properties.

```
fs = 1000; % Sample rate (Hz)
pathDelays = [0 2.5e-3]; % Path delays (s)
pathPower = [0 -6]; % Path power (dB)
fD = 5; % Maximum Doppler shift (Hz)
ns = 1000; % Number of samples
nsdel = 100; % Number of samples for delayed paths
```

Define the overall simulation time and three time segments for which data is to be transmitted. In this case, the channel is simulated for 1s with a 1000 Hz sampling rate. One 1000-sample continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at times 0.1 s, 0.4 s, and 0.7 s, respectively.

```
to0 = 0.0;
to1 = 0.1;
to2 = 0.4;
to3 = 0.7;
t0 = (to0:ns-1)/fs; % Transmission 0
t1 = to1+(0:nsdel-1)/fs; % Transmission 1
t2 = to2+(0:nsdel-1)/fs; % Transmission 2
t3 = to3+(0:nsdel-1)/fs; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],ns,1);
d1 = randi([0 1],nsdel,1);
d2 = randi([0 1],nsdel,1);
d3 = randi([0 1],nsdel,1);
```

Create a frequency-flat multipath Rayleigh fading System object, specifying the sum-of-sinusoids fading technique. So that results can be repeated, specify a seed value. Use the default `InitialTime` property setting so that the fading channel will be simulated from time 0. Enable the path gains output.

```
rayleighchan1 = comm.RayleighChannel('SampleRate',fs, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the multipath Rayleigh fading channel System object. Set the source for initial time so that the fading channel offset time can be specified as an input argument when using the System object.

```
rayleighchan2 = clone(rayleighchan1);
rayleighchan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first multipath Rayleigh fading channel System object, `rayleighchan1`. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = rayleighchan1(d0);
```

Pass random data through the second multipath Rayleigh fading channel System object, `rayleighchan2`, where the initial time offsets are provided as input arguments.

```
[~,pg1] = rayleighchan2(d1,to1);
[~,pg2] = rayleighchan2(d2,to2);
[~,pg3] = rayleighchan2(d3,to3);
```

Compare the number of samples processed by the two channels by using the `info` object function. The `rayleighchan1` object processed 1000 samples, while the `rayleighchan2` object only processed 300 samples.

```
G = info(rayleighchan1);
H = info(rayleighchan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

```
ans = 1×2
```

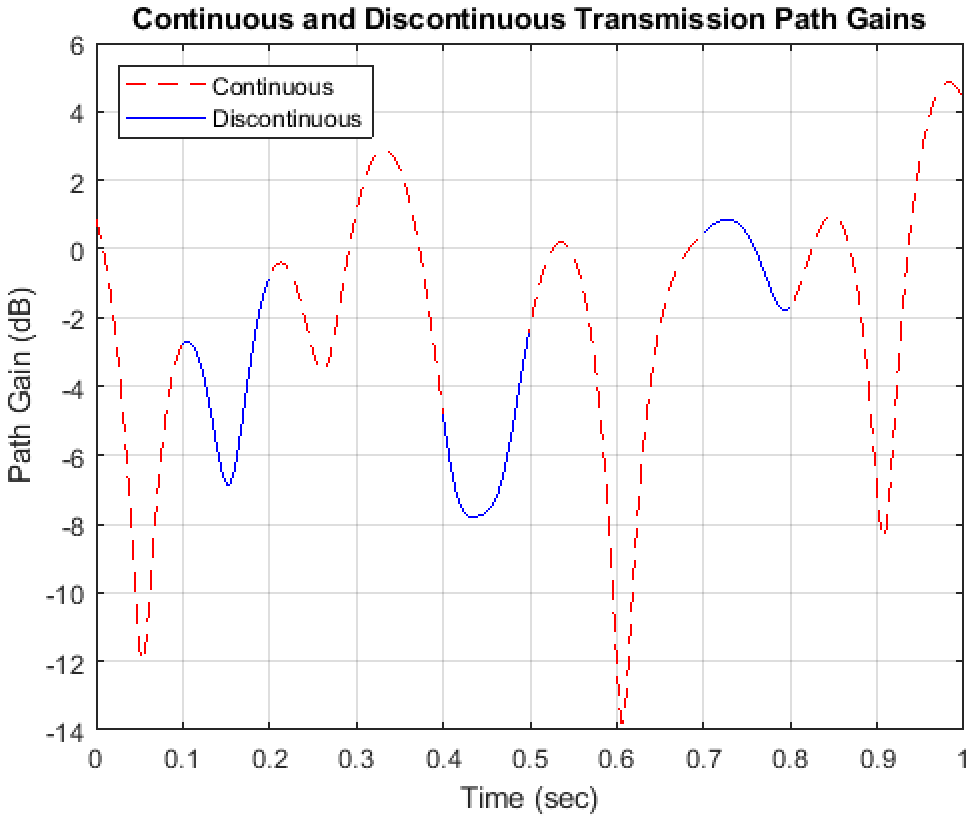
```
        1000         300
```

Convert the path gains into decibels.

```
pathGain0 = 20*log10(abs(pg0));
pathGain1 = 20*log10(abs(pg1));
pathGain2 = 20*log10(abs(pg2));
pathGain3 = 20*log10(abs(pg3));
```

Plot the path gains for the continuous and discontinuous cases. The gains for the three segments match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is suited to the simulation of packetized data, as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
title('Continuous and Discontinuous Transmission Path Gains')
```



Use Info Method to Reproduce Multipath Rayleigh Fading Channel Response

The example show how to use the `ChannelFilterCoefficients` property returned by the `info` object function of the `comm.RayleighChannel` System object to reproduce the multipath Rayleigh fading channel output.

Create a multipath Rayleigh fading channel System object, defining two paths and specifying data to pass through the channel.

```
rayleighchan = comm.RayleighChannel('SampleRate',1000,'PathDelays',[0 1.5e-3], ...
    'AveragePathGains',[0 -3],'PathGainsOutputPort',true)
```

```
rayleighchan = 
  comm.RayleighChannel with properties:

             SampleRate: 1000
             PathDelays: [0 0.0015]
       AveragePathGains: [0 -3]
     NormalizePathGains: true
    MaximumDopplerShift: 1.0000e-03
        DopplerSpectrum: [1x1 struct]

  Show all properties
```

```
data = randi([0 1],600,1);
```

Pass data through the channel. Assign the `ChannelFilterCoefficients` property value to the variable `coeff`.

```
[chanout1,pg] = rayleighchan(data);
chaninfo = info(rayleighchan)
```

```
chaninfo = struct with fields:
           ChannelFilterDelay: 6
    ChannelFilterCoefficients: [2x16 double]
          NumSamplesProcessed: 600
```

```
coeff = chaninfo.ChannelFilterCoefficients;
```

Calculate the fractional delayed input signal at the path delay locations stored in `coeff`.

```
Np = length(rayleighchan.PathDelays);
fracdelaydata = zeros(size(data,1),Np);
for ii = 1:Np
    fracdelaydata(:,ii) = filter(coeff(ii,:),1,data);
end
```

Apply the path gains and sum the results for all paths.

```
chanout2 = sum(pg .* fracdelaydata,2);
```

Compare the output of the multipath Rayleigh fading channel System object to the output reproduced using the path gains and the `ChannelFilterCoefficients` property of the multipath Rayleigh fading channel System object.

```
isequal(chanout1,chanout2)
```

```
ans = logical
   1
```

Verify Autocorrelation of Rayleigh Channel Path Gains

Verify that the autocorrelation of the path gain output from the Rayleigh channel System object is a Bessel function. Based on the results in [1] and Appendix A of [2], we know that when the autocorrelation of the path gain outputs is a Bessel function, and the Doppler spectrum is Jakes-shaped.

Initialize simulation parameters.

```
Rsym = 9600; % Input signal symbol rate (symbols/sec)
sps = 10; % Number of samples per input symbol
Fs = sps*Rsym; % Input signal sampling frequency (samples/sec)
Ts = 1/Fs; % Input signal sampling period (sec)
numsym = 1e6; % Total number of input symbols to simulate
numsamp = numsym*sps; % Total number of channel samples to simulate
fd = 100; % Maximum Doppler frequency shift (Hz)
num_acsamp = 5000; % Number of samples of autocovariance of complex fading process calculated
numtx = 1; % Number of transmit antennas
numrx = 1; % Number of receive antennas
numsin = 48; % Number of sinusoids
frameLen = 10000;
numFrames = numsamp/frameLen;
```

Configure a Rayleigh channel System object.

```
chan = comm.RayleighChannel('FadingTechnique','Sum of sinusoids', ...
    'NumSinusoids',numsin,'RandomStream','mt19937ar with seed', ...
    'PathDelays',0,'AveragePathGains',0,'SampleRate',Fs, ...
    'MaximumDopplerShift',fd,'PathGainsOutputPort',true);
```

Apply DPSK modulation to a random bit stream.

```
tx = randi([0 1],numsamp,numtx); % Random bit stream
dpskSig = dpskmod(tx,2); % DPSK signal
```

Pass modulated signal through the channel.

```
outsig = zeros(numsamp,numrx);
pg_rx = zeros(numsamp,numrx,numtx);
for frmNum = 1:numFrames
    [outsig((1:frameLen)+(frmNum-1)*frameLen,:),pathGains] = chan(dpskSig((1:frameLen)+(frmNum-1
    for i = 1:numrx
        pg_rx((1:frameLen)+(frmNum-1)*frameLen,i,:) = pathGains(:,:,:,i);
    end
end
```

Using the channel path gains received per antenna, compute the autocovariance of the fading process for each transmit-receive path.

```
autocov = zeros(frameLen+1,numrx,numtx);
autocov_normalized_real = zeros(num_acsamp+1,numrx,numtx);
```

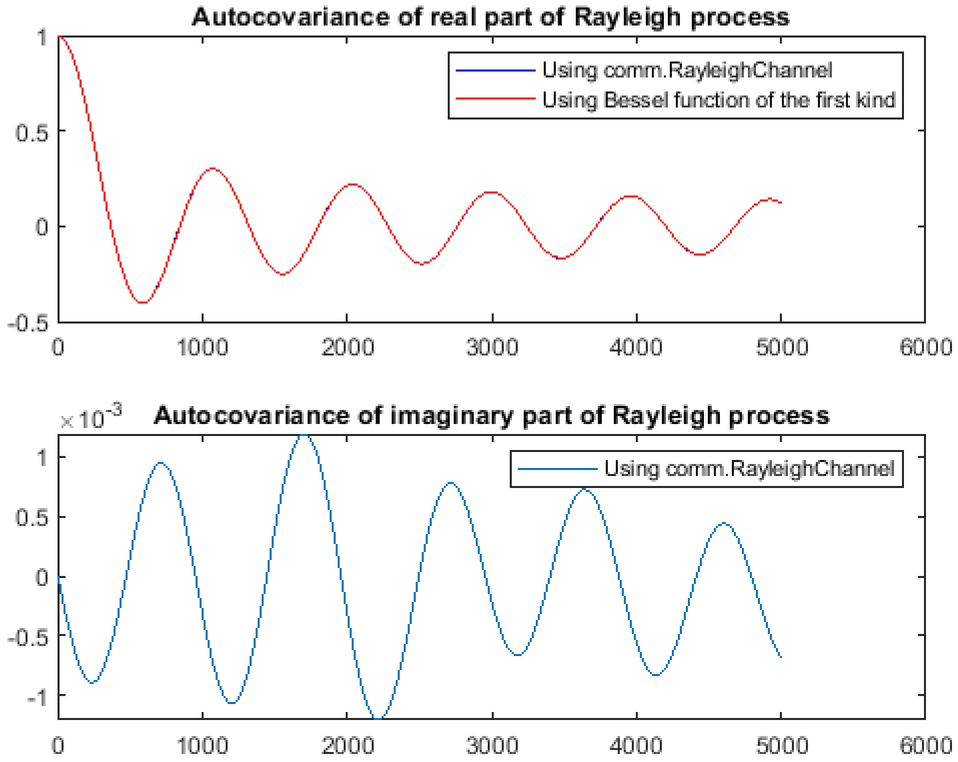
```
autocov_normalized_imag = zeros(num_acsamp+1,numrx,numtx);
for i = 1:numrx
    % Compute autocovariance of simulated complex fading process
    for j = 1:numtx
        autocov(:,i,j) = xcov(pg_rx(:,i,j),num_acsamp);
        % Real part of normalized autocovariance
        autocov_normalized_real(:,i,j) = real(autocov(num_acsamp+1:end,i,j)/autocov(num_acsamp+1
        % Imaginary part of normalized autocovariance
        autocov_normalized_imag(:,i,j) = imag(autocov(num_acsamp+1:end,i,j)/autocov(num_acsamp+1
    end
end
```

Using the `besselj` function compute the theoretical autocovariance of complex fading process.

```
Rrr = zeros(1,num_acsamp+1);
for n = 1:1:num_acsamp+1
    Rrr(n) = besselj(0,2*pi*fd*(n-1)*Ts);
end
Rrr_normalized = Rrr/Rrr(1);
```

Display the autocovariance comparing the results from the Rayleigh channel System object and the besselj function.

```
subplot(2,1,1)
plot(autocov_normalized_real,'b-')
hold on
plot(Rrr_normalized,'r-')
hold off
legend('Using comm.RayleighChannel','Using Bessel function of the first kind')
title('Autocovariance of real part of Rayleigh process')
subplot(2,1,2)
plot(autocov_normalized_imag)
legend('Using comm.RayleighChannel')
title('Autocovariance of imaginary part of Rayleigh process')
```



Compute mean square error.

```
act_mse_real = sum((autocov_normalized_real-repmat(Rrr_normalized.',1,numrx,numtx)).^2,1)/size(au
```

```
act_mse_real = 7.0043e-08
```

```
act_mse_imag = sum((autocov_normalized_imag-0).^2,1)/size(autocov_normalized_imag,1)
```

```
act_mse_imag = 4.1064e-07
```

References

1. Dent, P., G.E. Bottomley, and T. Croft. “Jakes Fading Model Revisited.” *Electronics Letters* 29, no. 13 (1993): 1162. https://doi.org/10.1049/el:19930777.

2. Pätzold, Matthias. *Mobile Fading Channels*. Chichester, UK: John Wiley & Sons, Ltd, 2002. https://doi.org/10.1002/0470847808.

More About

Cutoff Frequency Factor

The cutoff frequency factor, f_c, is determined for different Doppler spectrum types.

- For any Doppler spectrum type other than Gaussian and biGaussian, f_c equals 1.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation` × $\sqrt{2\log 2}$.

- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1) ×` $\sqrt{2\log 2}$.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2) ×` $\sqrt{2\log 2}$.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1) ×` $\sqrt{2\log 2}$.
  - In all other cases, f_c equals 1.

Channel Visualization

The `comm.RayleighChannel` System object enables visualization of the channel impulse response, frequency response, and Doppler spectrum.

Note

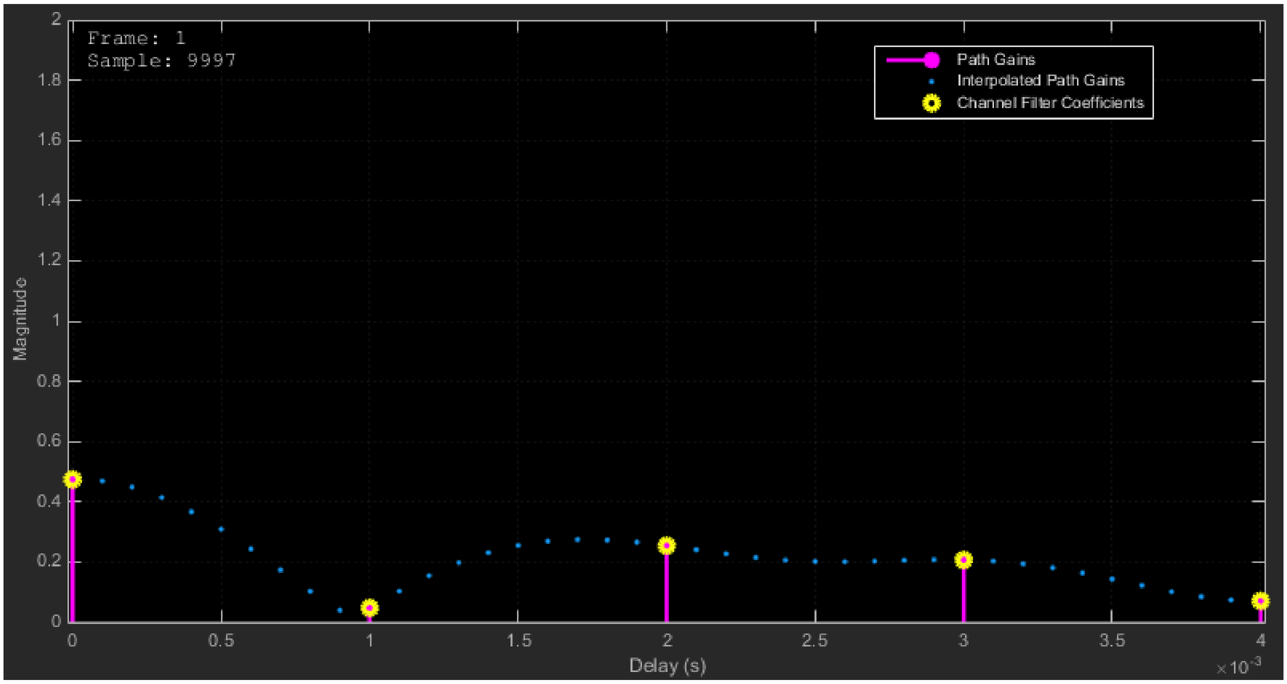
- The displayed and specified path gain locations can differ by as much as 5% of the input sample time.
- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Playback > Reduce Updates to Improve Performance** scope menu item. Reducing the percentage of samples to display and enabling reduced updates can speed up the rendering of the visualization scope.
- After the visualization scopes are manually closed, calls to the System object are executed at its normal speed.
- Code generation is available only when the `Visualization` property is set to `'Off'`.

Impulse Response

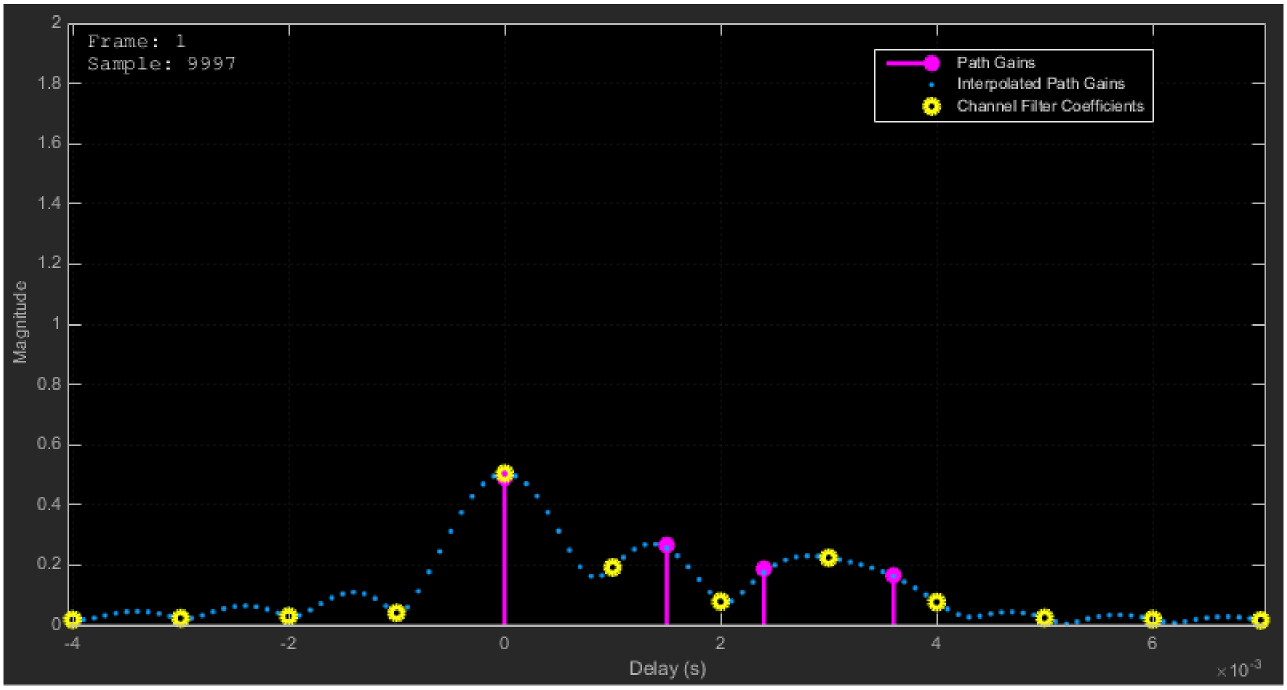
The impulse response scope displays the path gains, channel filter coefficients, and interpolated path gains. The path gains occur at time instances which correspond to the specified `PathDelays` property and might not be aligned with the input sampling time. The channel filter coefficients are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. In cases in which the path gains are aligned with the sampling time, the path gains overlap the filter coefficients. Sinc interpolation is used to connect the channel filter coefficients and is shown as the interpolated path gains. These points are used solely for display purposes and not used in subsequent channel filtering. For a flat fading channel (one path), the sinc interpolation curve is not displayed. For all impulse response plots, the frame and sample numbers are shown in the upper-left corner of the scope.

The impulse response plot shares the same toolbar and menus as the `dsp.ArrayPlot` System object.

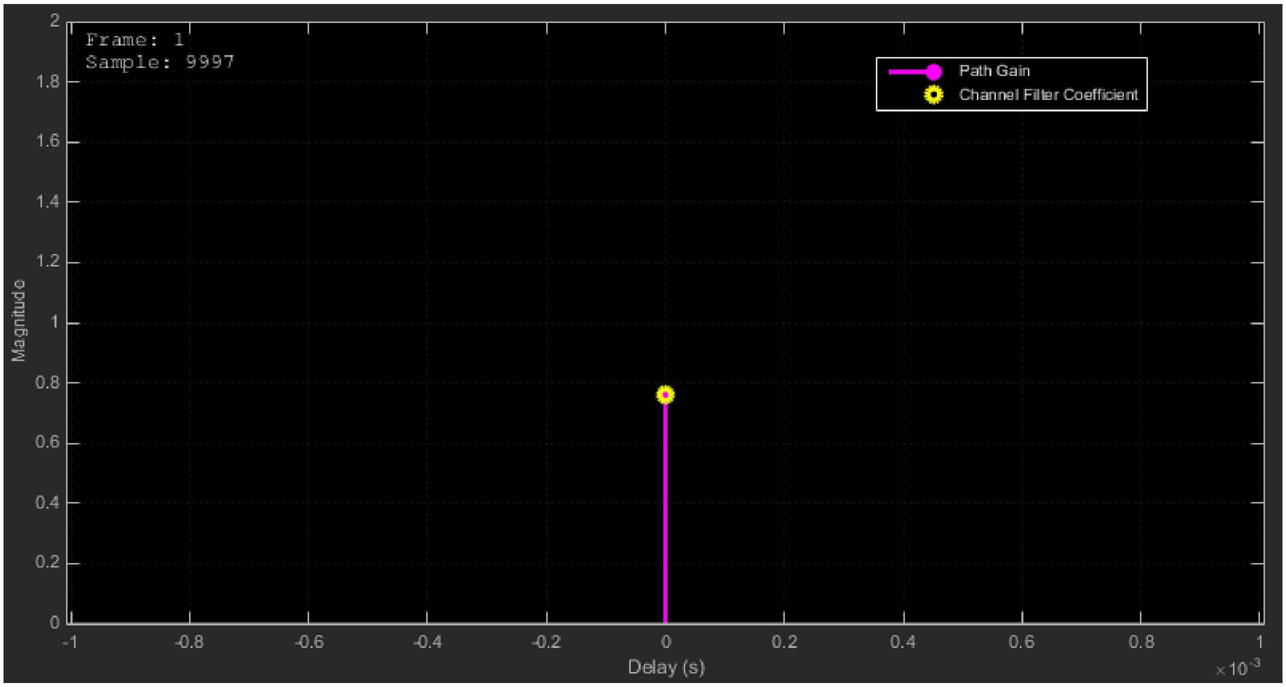
When the specified path gains align with the sample rate, the path gains and the channel filter coefficients overlap. In this figure the impulse response shows the path gains overlap the filter coefficients.



If the specified path gains are not aligned with the sample rate, the path gains and the channel filter coefficients do not overlap. In this figure, the filter coefficients are equally distributed. The path gains do not overlap with the channel filter coefficients because the path gains are not aligned with the sample rate.



This figure shows the impulse response for a frequency-flat channel. In this case, the interpolated path gains are not displayed.



Frequency response

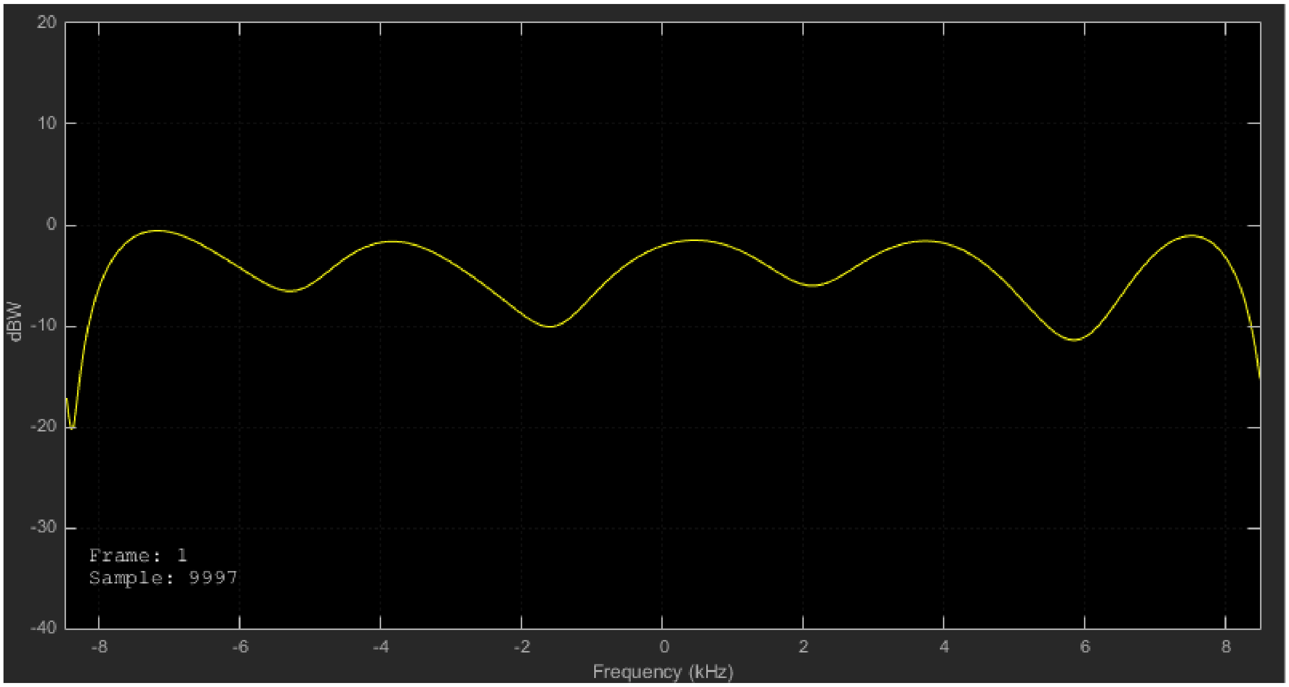
The frequency response scope displays the multipath Rayleigh fading channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. The frequency response plot shares the same toolbar and menus as the `dsp.SpectrumAnalyzer` System object.

The y-axis limits of the plot are computed based on the `NormalizePathGains` and `AveragePathGains` properties of the `comm.RayleighChannel` System object.

This table shows other selected default spectrum settings. These settings can be changed from their default values by using the **View > Spectrum Settings** menu.

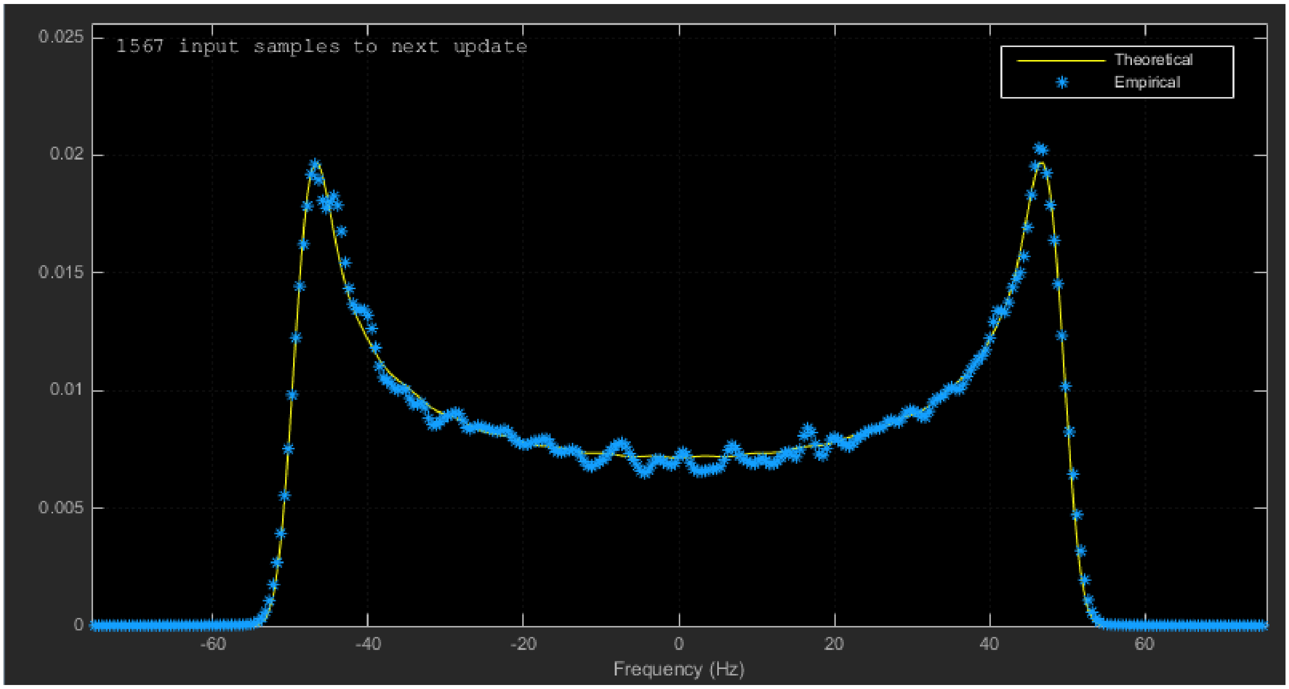
Spectrum Settings	Value
Main options>Type	`Power`
Main options>Window length	Channel filter length
Main options>NFFT	`512`
Window options>Window	`Rectangular`
Trace options>Units	`dBW`

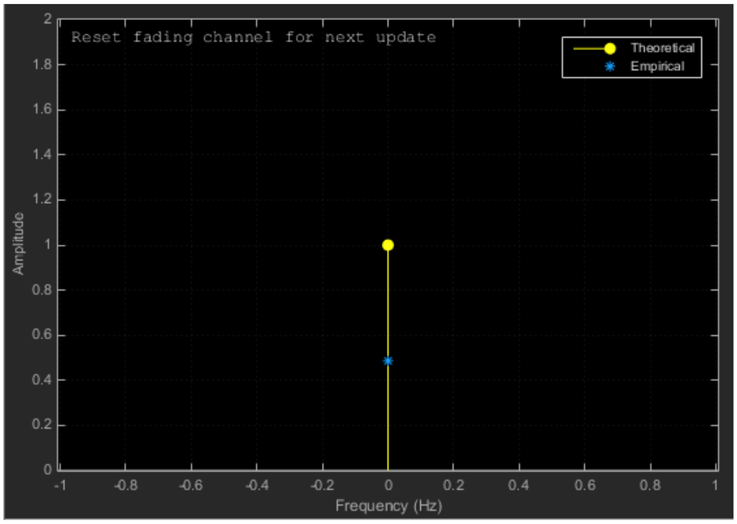
This figure shows the frequency response plot for a frequency selective channel.



Doppler Spectrum

The Doppler spectrum plot displays the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a line for the case of nonstatic channels and as a point for static channels. The empirical data is shown as * symbols. Before the empirical plot is updated, the internal buffer must be completely filled with filtered Gaussian samples. The empirical plot is the running mean of the spectrum calculated from each full buffer. For nonstatic channels, the number of input samples needed before the next update is displayed in the upper-left corner of the plot. The number of samples needed is a function of the sample rate and the maximum Doppler shift. For static channels, the text "`Reset fading channel for next update`" is displayed.





References

[1] Oestges, Claude, and Bruno Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*. 1st ed. Boston, MA: Elsevier, 2007.

[2] Correia, Luis M., and European Cooperation in the Field of Scientific and Technical Research (Organization), eds. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*. 1st ed. Amsterdam ; Boston: Elsevier/Academic Press, 2006.

[3] Kermoal, J.P., L. Schumacher, K.I. Pedersen, P.E. Mogensen, and F. Frederiksen. “A Stochastic MIMO Radio Channel Model with Experimental Validation.” *IEEE Journal on Selected Areas in Communications* 20, no. 6 (August 2002): 1211–26. https://doi.org/10.1109/JSAC.2002.801223.

[4] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan. *Simulation of Communication Systems*. Second edition. Boston, MA: Springer US, 2000.

[5] Patzold, M., Cheng-Xiang Wang, and B. Hogstad. “Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms.” *IEEE Transactions on Wireless Communications* 8, no. 6 (June 2009): 3122–31. https://doi.org/10.1109/TWC.2009.080769.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- To generate C code, set the `DopplerSpectrum` property to a single Doppler spectrum structure.
- Code generation is available only when the `Visualization` property is `'Off'`.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects
`comm.AWGNChannel` | `comm.ChannelFilter` | `comm.MIMOChannel` | `comm.RicianChannel` | `comm.WINNER2Channel`

Functions
`doppler`

Blocks
MIMO Fading Channel | SISO Fading Channel

Introduced in R2013b

comm.RayTracingChannel

Package: comm

Filter signal through multipath fading channel defined by propagation rays

Description

The comm.RayTracingChannel System object filters a signal through a multipath fading channel that is defined by propagation rays.

To filter a signal through a fading channel defined by propagation rays:

1. Create the comm.RayTracingChannel object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
rtchan = comm.RayTracingChannel
rtchan = comm.RayTracingChannel(Name,Value)
rtchan = comm.RayTracingChannel(rays,tx,rx)
```

Description

rtchan = comm.RayTracingChannel creates a ray-tracing fading channel System object, which defines the multipath environment using a set of propagation rays.

rtchan = comm.RayTracingChannel(Name,Value) sets properties using one or more name-value pairs. Enclose each property name in quotes. For example, 'SampleRate',1e6 sets the sample rate to 1 MHz.

rtchan = comm.RayTracingChannel(rays,tx,rx) creates a ray-tracing fading channel System object given inputs rays, tx, and rx.

- rays, specified as a set of comm.Ray objects, is used to set the PropagationRays property.
- tx, specified as a txsite object, is used to set the TransmitArray and TransmitArrayOrientationAxes properties.
- rx, specified as an rxsite, is used to set the ReceiveArray and ReceiveArrayOrientationAxes properties.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the release function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

SampleRate — Sample rate
10000000 (default) | positive scalar

Sample rate of the input signal in Hz, specified as a positive scalar.

Data Types: double

PropagationRays — Propagation rays
comm.Ray object (default) | row vector of comm.Ray objects | row cell array of comm.Ray objects

Propagation rays, specified as a comm.Ray object, a row vector of comm.Ray objects, or a row cell array of comm.Ray objects. This property specifies the propagation rays between the transmit and receive antenna arrays. All of the specified comm.Ray objects must have the same Frequency property setting. Any of the specified comm.Ray objects that have their PathSpecification property set to 'Locations', must have the same CoordinateSystem, TransmitterLocation, and ReceiverLocation property settings.

For code generation, the PropagationRays property must be a cell array of comm.Ray objects.

Data Types: comm.Ray object | cell

TransmitArray — Transmit antenna array
arrayConfig object (default) | phased.IsotropicAntennaElement | ...

Transmit antenna array, specified as one of these options.

- An arrayConfig object — You can adjust the Size property of the arrayConfig object to have it represent a uniform rectangular array (URA), uniform linear array (ULA), or single phased.IsotropicAntennaElement. The default configuration for an arrayConfig object is a 2-by-2 URA with an element spacing of 0.5 m.
- A phased array antenna System object — If you have the “Phased Array System Toolbox” software, you can specify phased array antenna System object configurations. For a list of these additional supported values, see “Phased Array Antenna Options” on page 3-1232.

TransmitArrayOrientationAxes — Orientation axes of transmit antenna array
eye(3) (default) | 3-by-3 unitary matrix

Orientation axes of the transmit antenna array, specified as a 3-by-3 unitary matrix indicating the rotation from the local coordinate system (LCS) to the global coordinate system (GCS). If the comm.Ray objects defined in the PropagationRays property set the CoordinateSystem property to 'Geographic', the GCS is the East-North-Up (ENU) coordinate system at the transmitter.

Data Types: double

ReceiveArray — Receive antenna array
arrayConfig object | phased.IsotropicAntennaElement | ...

Receive antenna array, specified as one of these options.

- An arrayConfig object — You can adjust the Size property of the arrayConfig object to have it represent a uniform rectangular array (URA), uniform linear array (ULA), or single phased.IsotropicAntennaElement. The default configuration for an arrayConfig object is a 2-by-2 URA with an element spacing of 0.5 m.

- A phased array antenna System object — If you have the “Phased Array System Toolbox” software, you can specify phased array antenna System object configurations. For a list of these additional supported values, see “Phased Array Antenna Options” on page 3-1232.

ReceiveArrayOrientationAxes — Orientation axes of receive antenna array

eye(3) (default) | 3-by-3 unitary matrix

Orientation axes of the receive antenna array, specified as a 3-by-3 unitary matrix indicating the rotation from the LCS to the GCS. If the comm.Ray objects defined in the PropagationRays property set the CoordinateSystem to 'Geographic', the GCS is the East-North-Up (ENU) coordinate system at the receiver.

Data Types: double

ReceiverVirtualVelocity — Receive antenna array instantaneous velocity

[1; 1; 0] (default) | three element column vector

Receive antenna array instantaneous velocity in the GCS in m/s, specified as a three element column vector of the form [*x*; *y*; *z*]. The elements of this vector specify *x*-, *y*-, and *z*-velocity, respectively. If the comm.Ray objects defined in the PropagationRays property set the CoordinateSystem to 'Geographic', the GCS is the East-North-Up (ENU) coordinate system at the receiver.

Data Types: double

NormalizeImpulseResponses — Option to normalize channel impulse responses

1 or true (default) | 0 or false

Option to normalize channel impulse responses (CIRs), specified as a logical 1 (true) or 0 (false). Set this property to 1 (true) to normalize the gains of CIRs to 0 dB from each transmit array element to each receive array element.

Data Types: logical

NormalizeChannelOutputs — Option to normalize channel outputs by number of receive elements

1 or true (default) | 0 or false

Option to normalize channel outputs by the number of receive elements, specified as a logical 1 (true) or 0 (false). Set this property to 1 (true) to normalize the channel output by the number of receive array elements.

Data Types: logical

Usage

Syntax

```
y = rtchan(x)
```

Description

`y = rtchan(x)` filters the input signal through a multipath fading channel defined by a set of propagation rays. When the channel filters the input signal, time zero is assigned to the ray with the minimum propagation delay. Arrival of other rays are delayed relative to time zero.

Input Arguments

x — Input signal
N_S-by-N_T matrix

Input signal, specified as an N_S-by-N_T matrix.

- N_S is the number of samples.
- N_T is the number of transmit array elements.

Data Types: `single` | `double`
Complex Number Support: Yes

Output Arguments

y — Output signal
N_S-by-N_R matrix

Output signal, returned as an N_S-by-N_R matrix.

- N_S is the number of samples.
- N_R is the number of receive array elements.

`y` is the same data type as input `x`.

`cir` — Channel impulse response
N_S-by-N_P-by-N_T-by-N_R array

Channel impulse response (CIR), returned as an N_S-by-N_P-by-N_T-by-N_R array.

- N_S is the number of samples.
- N_P is the number of paths (specifically, the number of rays as indicated by the length of the `PropagationRays` property).
- N_T is the number of transmit array elements.
- N_R is the number of receive array elements.

`cir` is the same data type as input `x`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.RayTracingChannel`

info	Characteristic information about ray-tracing channel
showProfile	Plot temporal and spatial profiles of ray-tracing channel
clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

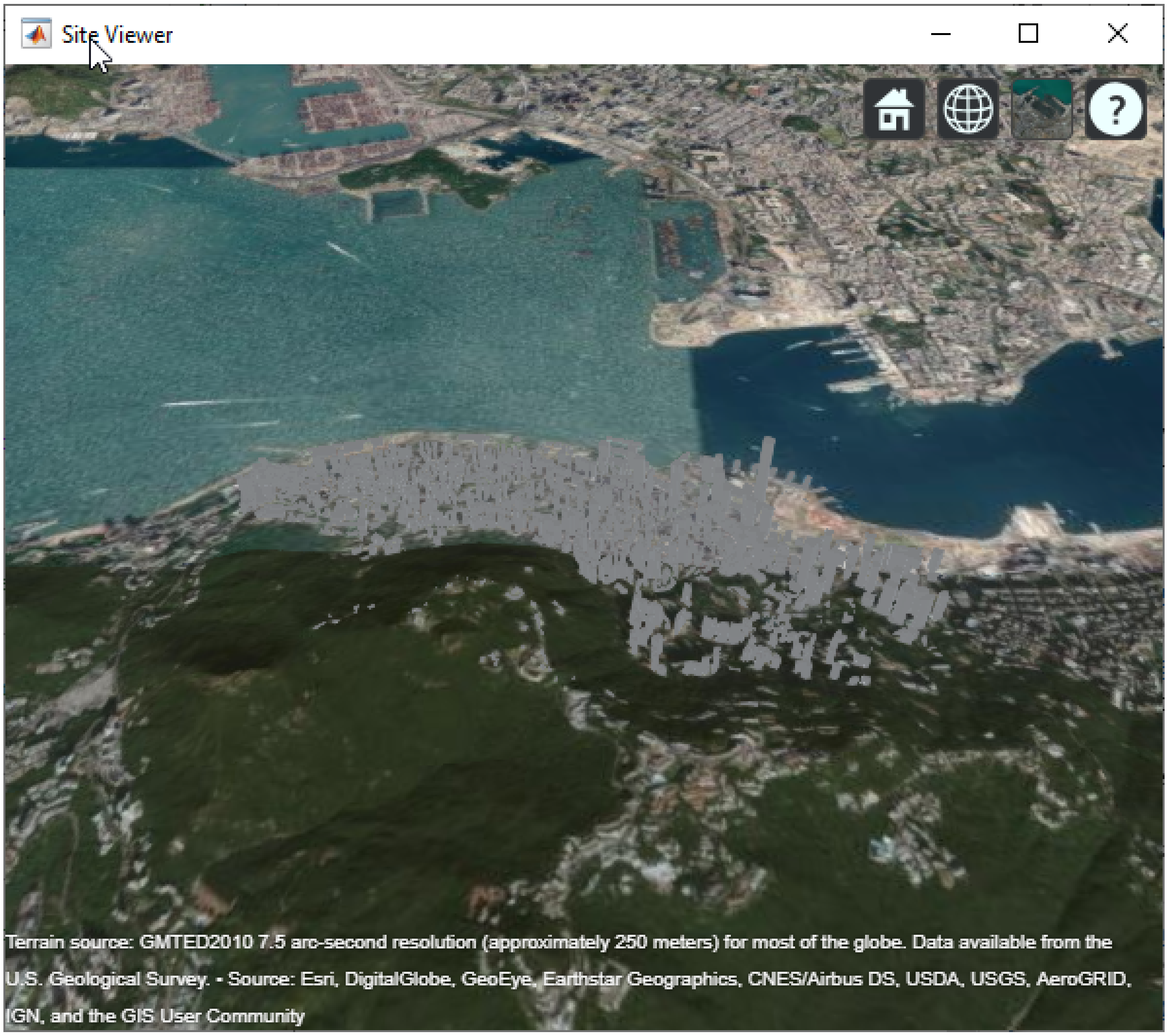
Examples

Apply Ray-Tracing Channel Between Two Sites in Hong Kong

Perform ray tracing between two sites in Hong Kong, China, build a multipath channel model using the ray-tracing result, and filter the signals through the channel.

Create a Site Viewer map display of buildings in Hong Kong. For more information about the osm file, see [1] on page 3-0   . Create transmitter and receiver sites.

```
sv = siteviewer("Buildings","hongkong.osm");
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

```
tx = txsite("Latitude",22.2789,"Longitude",114.1625, ...
    "AntennaAngle",30,"AntennaHeight",10,"TransmitterFrequency",28e9);
rx = rxsite("Latitude",22.2799,"Longitude",114.1617, ...
    "AntennaAngle",120,"AntennaHeight",1);
```

Perform ray tracing to find rays with up to 2 reflections.

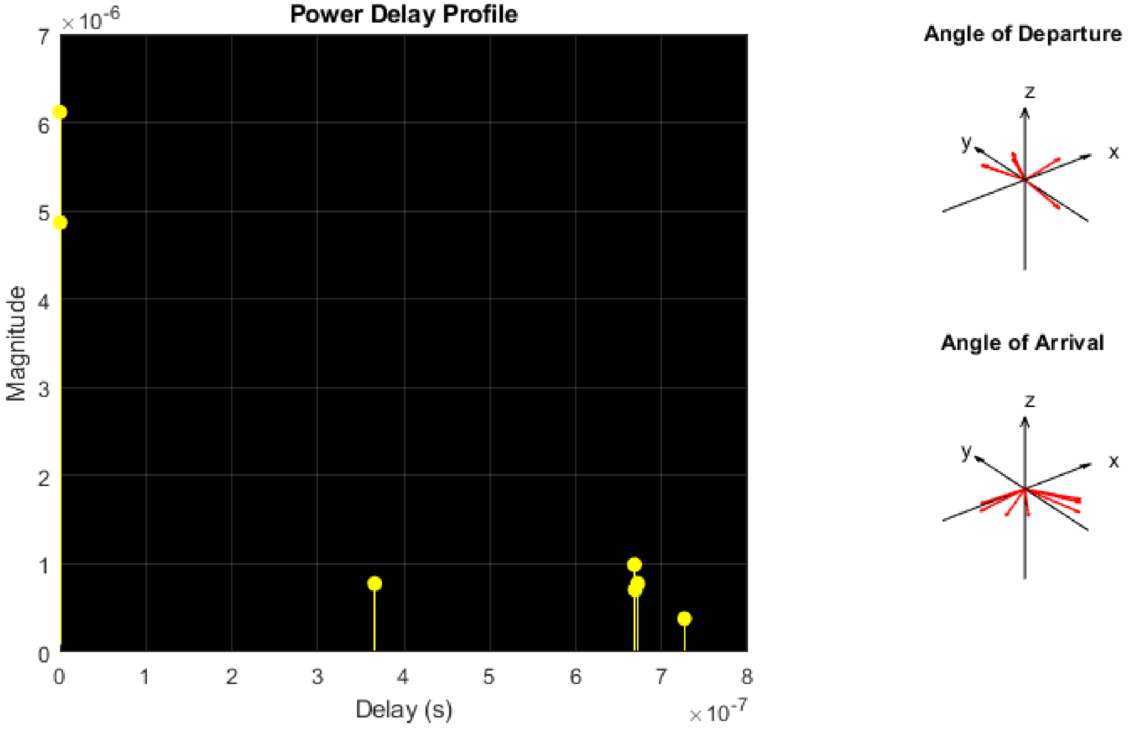
```
rays = raytrace(tx,rx,"NumReflections",[0 1 2]);
```

Create a channel model by using the transmitter site, receiver site, and calculated rays between the sites.

```
rtchan = comm.RayTracingChannel(rays{1},tx,rx);
```

Show temporal and spatial profiles of the ray-tracing channel.

```
showProfile(rtchan);
```

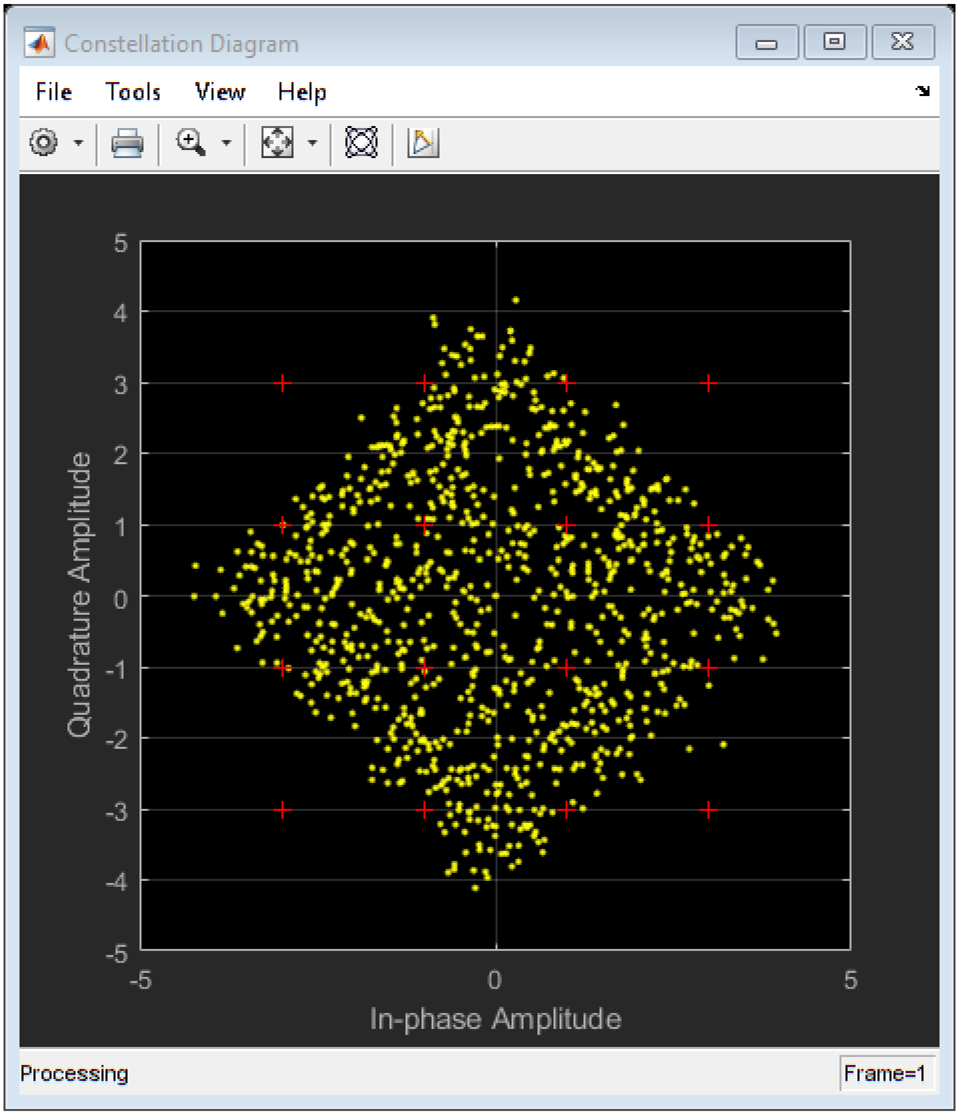


Filter a randomly generated 16-QAM signal through the channel.

```
modOrd = 16; % Modulation order
frmLen = 1e3; % Frame length
numTx = rtchan.info.NumTransmitElements;
x = qammod(randi([0,modOrd-1],frmLen,numTx),modOrd);
y = rtchan(x);
```

Show the filtered signal in a constellation diagram.

```
constdiag = comm.ConstellationDiagram("XLimits",[-5 5], ...
    "YLimits",[-5 5],"ReferenceConstellation", ...
    qammod(0:modOrd-1,modOrd));
constdiag(y);
```



Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

More About

Phased Array Antenna Options

If you have “Phased Array System Toolbox”, you can specify any of these phased antenna System object configurations for the `TransmitArray` and `ReceiveArray` properties.

- `phased.IsotropicAntennaElement`
- `phased.CustomAntennaElement`

- phased.URA with the Element property set to a phased.IsotropicAntennaElement or phased.CustomAntennaElement System object
- phased.ULA with the Element property set to a phased.IsotropicAntennaElement or phased.CustomAntennaElement System object
- phased.ConformalArray with the Element property set to a phased.IsotropicAntennaElement or phased.CustomAntennaElement System object

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

- For code generation, the PropagationRays property must be a cell array of comm.Ray objects.

See Also

Objects

arrayConfig | comm.ChannelFilter | comm.Ray | phased.ConformalArray | phased.CustomAntennaElement | phased.IsotropicAntennaElement | phased.ULA | phased.URA | rxsite | siteviewer | txsite

Functions

raytrace

Introduced in R2020b

comm.RBDSWaveformGenerator

Package: comm

Generate RDS/RBDS waveform

Description

The comm.RBDSWaveformGenerator System object generates configurable standard-compliant baseband RDS/RBDS waveforms in MATLAB. RDS/RBDS waveforms supplement FM radio stations with additional textual information, such as song title, artist name, and station description. The RDS/RBDS signal lies in the 57-kHz band of the baseband FM radio signal.

Use this object to generate a waveform containing RadioText Plus (RT+) information and register a custom encoding implementation for an Open Data Application (ODA). You can also specify the time, data, and the program type. The object supports short, scrolling 8-character text, and longer 32-character or 64-character text.

To generate baseband RDS/RBDS waveforms:

1. Create a comm.RBDSWaveformGenerator object and set the properties of the object.
2. Call step to generate the waveform.

Note Alternatively, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

rbdsgen = comm.RBDSWaveformGenerator creates an RDS/RBDS waveform generator object, rbdsgen, using the default properties.

rbdsgen = comm.RBDSWaveformGenerator(Name,Value) specifies additional properties using Name,Value pairs. Unspecified properties have default values.

Example:

```
rbdsgen = comm.RBDSWaveformGenerator( ...
     'GroupsPerFrame',20,'SamplesPerSymbol',10, ...
     'SendRadioTextPlus',true);
```

Properties

If a property is listed as tunable, then you can change its value even when the object is locked.

SamplesPerSymbol — Number of samples per symbol

10 (default) | positive even integer

Number of samples per symbol (bit), specified as a positive even integer. Half of the samples represent one amplitude level of Manchester coding. The other half of the samples represent the opposite level.

GroupsPerFrame — Number of groups per output frame
10 (default) | scalar integer

Number of groups per output frame, specified as a scalar integer. Each group is 104 symbols (bits) long.

RadioText — Long text conveyed with type 2A groups
'Long text' (default) | character vector

Radio text conveyed with type 2A groups, specified as a character vector that is up to 64 characters long. The object transmits the specified text four characters at a time, using type 2A groups.

Tunable: Yes

ProgramServiceName — Label of the program service
'ShortTxt' (default) | character vector

Label of the program service, specified as a character vector that is up to eight characters long. This information is conveyed as a short text with type 0A groups, two characters at a time.

Tunable: Yes

ProgramIdentificationCode — Program identification code
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0] (default) | 16-bit row vector

Program identification (PI) code, specified as a 16-bit row vector. In North America, the PI code conveys the call letters of the station. Example call letters include 'WABC' and 'KXYZ'.

To generate North American PI codes for a station's call letters, use the callLettersToPICode method.

ProgramType — Program type
'No program type or undefined' (default) | character vector

Program type, specified as a character vector containing one of the 31 values allowed by the RDS/RBDS standard. For a list of program types that the RDS/RBDS standard allows in North America, see [1].

Tunable: Yes

ProgramTypeName — Program type name
'' (default) | character vector

Program type name, specified as a character vector that is up to eight characters long. This text further characterizes the program type, such as 'Football' for the program type 'Sports'. The object conveys the program type name using type 10A groups. If this property is empty, then no 10A groups are generated.

Tunable: Yes

SendDateTime — Option to advertise date and time
false (default) | true

Option to advertise the date and time, specified as either `false` or `true`. When you set this property to `true`, one 4A group is periodically generated every 685 groups (once a minute).

`AlternativeFrequencies` — Alternative frequencies
`[]` (default) | row vector

Alternative frequencies, specified as a row vector in MHz. This information is conveyed with type 0A groups. It indicates other transmitters broadcasting the same program in the same or adjacent reception areas. With this information, receivers can switch to a different frequency with better reception.

`SendRadioTextPlus` — Option to send RT+ information
`false` (default) | `true`

Option to transmit RadioText Plus (RT+) information, specified as a scalar logical. When you set this property to `true`, the RT+ ODA information is advertised with type 3A groups. In addition, the RT+ content types, specified in `RadioTextType1`, `RadioTextType2`, and the two RT+ substrings indexed by `RadioTextIndices` are conveyed with the open-format type 11A group.

`RadioTextType1` — Content type of first RT+ substring
`'Item.Artist'` (default) | character vector

Content type of the first RT+ substring, specified as a character vector. Allowed values are the class names specified in the RT+ standard. For more details, see [2].

Tunable: Yes

`RadioTextType2` — Content type of second RT+ substring
`'Item.Title'` (default) | character vector

Content type of the second RT+ substring, specified as a character vector. Allowed values are the class names specified in the RT+ standard. For more details, see [2].

Tunable: Yes

`RadioTextIndices` — Start and end indices of RT+ substrings
`[1 2; 3 4]` (default) | 2-by-2 matrix of positive integers

Start and end indices of RT+ substrings, specified as a 2-by-2 matrix of positive integers. The first column indexes the beginning of each RT+ substring. The second column indexes the end of each substring.

Tunable: Yes

Methods

callLettersToPICode	Convert North-American call letters to binary PI code
registerODA	Register a custom encoding implementation for an ODA
reset	Reset states of RBDS waveform generator object
step	Generate RDS/RBDS waveform

Examples

Generate a Basic RBDS Waveform

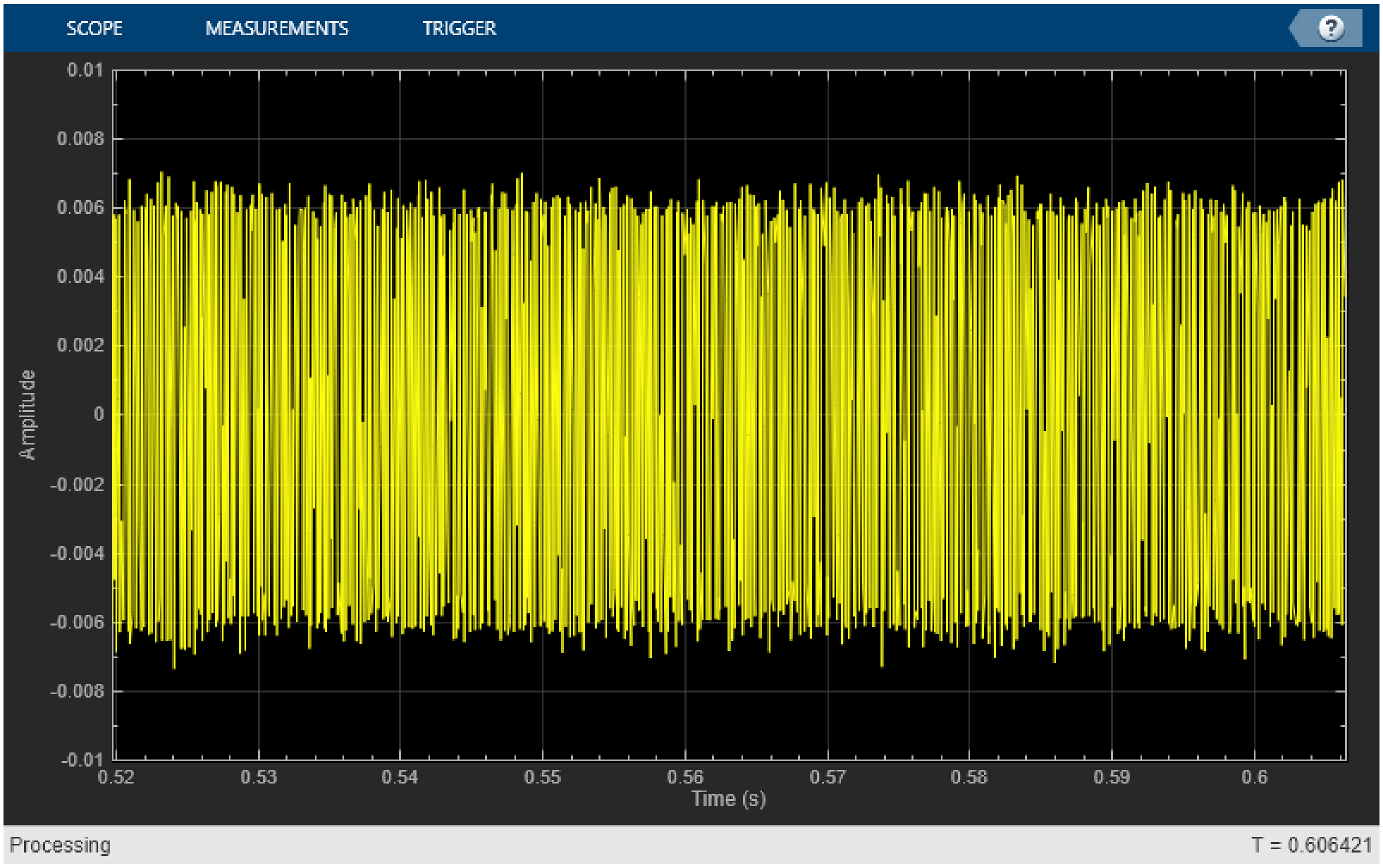
Generate a basic RBDS waveform, FM modulate the waveform with an audio signal, and then demodulate the waveform.

Each frame of the RBDS waveform contains 19 groups, with a group length of 104 bits (symbols) each. Set the number of samples per RBDS symbol to 10. Therefore, the number of samples in each frame of RBDS waveform is 104 x 10 x 19 = 19,760. According to the RBDS standard, the bit rate is 1187.5 Hz. So, the RBDS sample rate = 1187.5 x samples per RBDS symbol. Set the audio frame rate to 40 x 1187.5 = 47,500.

```
groupLen = 104;
sps = 10;
groupsPerFrame = 19;
rbdsFrameLen = groupLen*sps*groupsPerFrame;
afrRate = 40*1187.5;
rbdsRate = 1187.5*sps;
outRate = 4*57000;

afr = dsp.AudioFileReader('rbds_capture_47500.wav','SamplesPerFrame',rbdsFrameLen*afrRate/rbdsRat
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',groupsPerFrame,'SamplesPerSymbol',sps);

fmMod = comm.FMBroadcastModulator('AudioSampleRate',afr.SampleRate,'SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'RBDSSamplesPerSymbol',sps);
fmDemod = comm.FMBroadcastDemodulator('SampleRate',outRate,...
    'Stereo',true,'RBDS',true,'PlaySound',true);
scope = timescope('SampleRate',outRate,'YLimits',10^-2*[-1 1]);
```

Get the audio input and generate the RBDS waveform. FM modulate the stereo audio with the RBDS waveform, add noise, and FM demodulate the audio and RBDS waveforms. View the demodulated RBDS waveform in the time scope.

```
for idx = 1:7
    input = afr();                        % get current audio input
    rbdsWave = rbds();                    % generate RBDS info at the same configured rate
    yFM = fmMod([input input], rbdsWave); % FM modulate stereo audio with RBDS info
    rcv = awgn(yFM, 40);                  % add noise
    [audioRcv, rbdsRcv] = fmDemod(rcv);   % FM demodulate the audio and RBDS waveforms
    scope(rbdsRcv);
end
```



Generate RBDS Waveform with RadioText Plus Information

Create a `comm.RBDSWaveformGenerator` System object™ with 20 groups per frame and 10 samples per symbol. Add the Radio Text plus (RT+) information, such as artist name and song, title, to the waveform. Indicate the start and end of the RT+ substrings by using the `RadioTextIndices` property.

```
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',20,'SamplesPerSymbol',10,...
    'SendRadioTextPlus', true);
rbds.RadioText = 'MyArtist - MySongTitle';
rbds.RadioTextType1 = 'Item.Artist';
rbds.RadioTextType2 = 'Item.Title';
rbds.RadioTextIndices = [1 8; 12 22];
for idx = 1:10
    rbds.step();
end
```

Register a Custom Encoding Implementation

Register a custom encoding implementation for an Open Data Application (ODA) by using the `registerODA` method of the `comm.RBDSWaveformGenerator` System object™. Set the ODA ID to `'CD46'`, which is the ID for the traffic message channel. The allocated group type is 8A.

```
rbds = comm.RBDSWaveformGenerator();
odaID = 'CD46';
allocatedGroupType = '8A';
```

This example uses the following templates as a starting point for custom encoding implementation.

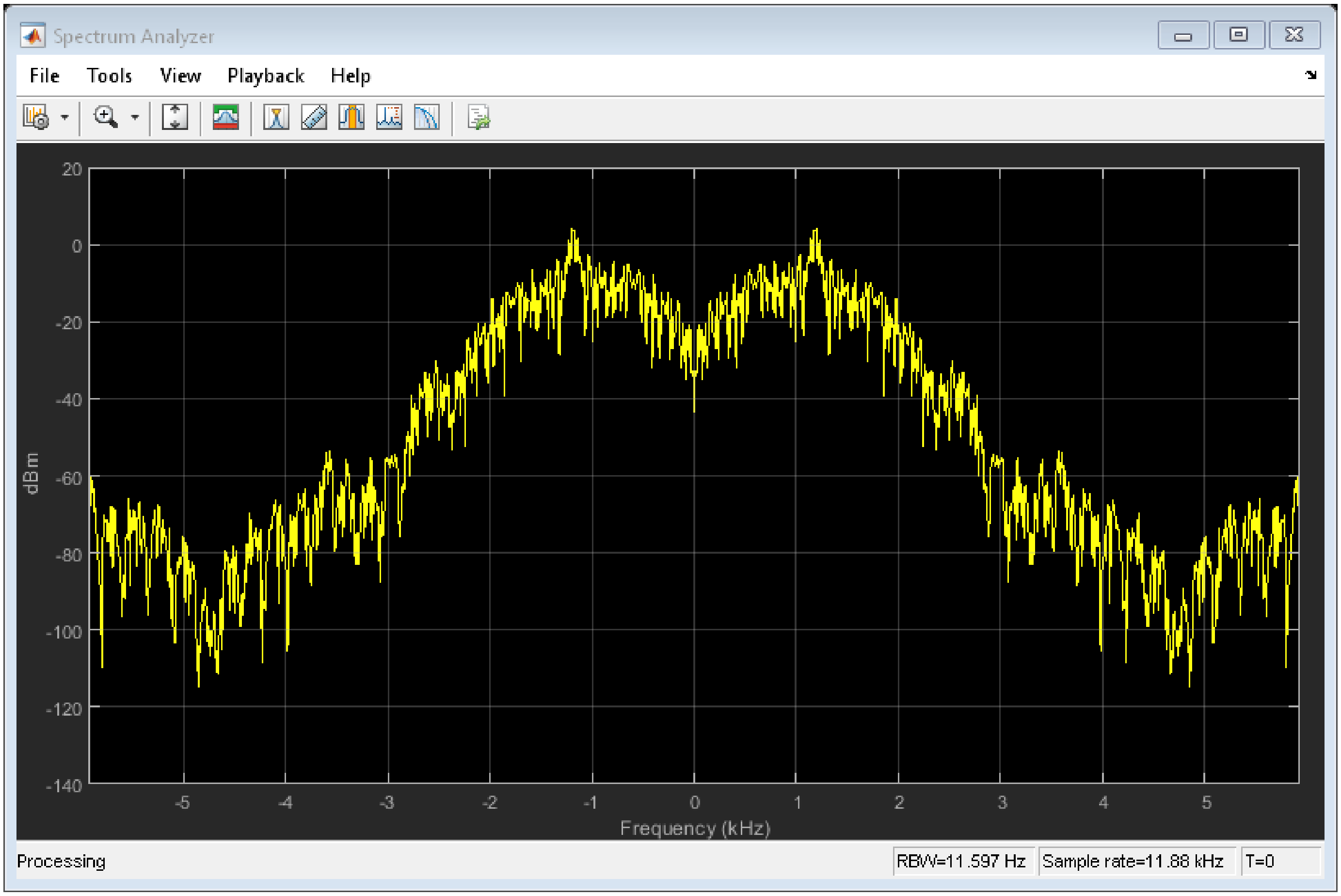
```
mainProcessingFcn = @CustomODAEncodingMain;
fcn3A             = @CustomODAEncoding3A;
registerODA(rbds,odaID,allocatedGroupType,mainProcessingFcn,fcn3A);
s = info(rbds);
s.ODAMap
```

```
ans=2×1 struct array with fields:
    ID
    GroupType
    FunctionMain
    Function3A
```

Configure RBDS Waveforms with Date and Time Information

Generate RBDS waveform with date and time information, the program type, and alternative frequencies. The `comm.RBDSWaveformGenerator` object uses type 4A groups for date and time information, type 10A groups for the program type information, and type 0A groups for alternative frequencies. View the waveform in a spectrum analyzer.

```
rbds = comm.RBDSWaveformGenerator('GroupsPerFrame',1000);
scope = dsp.SpectrumAnalyzer('SampleRate',1187.5*rbds.SamplesPerSymbol,'YLimits',[-140 20]);
rbds.SendDateTime = true;          % send type 4A groups
rbds.ProgramType = 'Sports';
rbds.ProgramTypeName = 'Football'; % send type 10A groups
rbds.AlternativeFrequencies = [99.1 102.5]; % info sent in type 0A groups
wave = rbds.step();
scope(wave)
```



Algorithms

`comm.RBDSWaveformGenerator` generates waveforms according to the RDS/RBDS standard [1]. The RDS/RBDS standard consists of three layers: physical layer, data-link layer, and session and application layer.

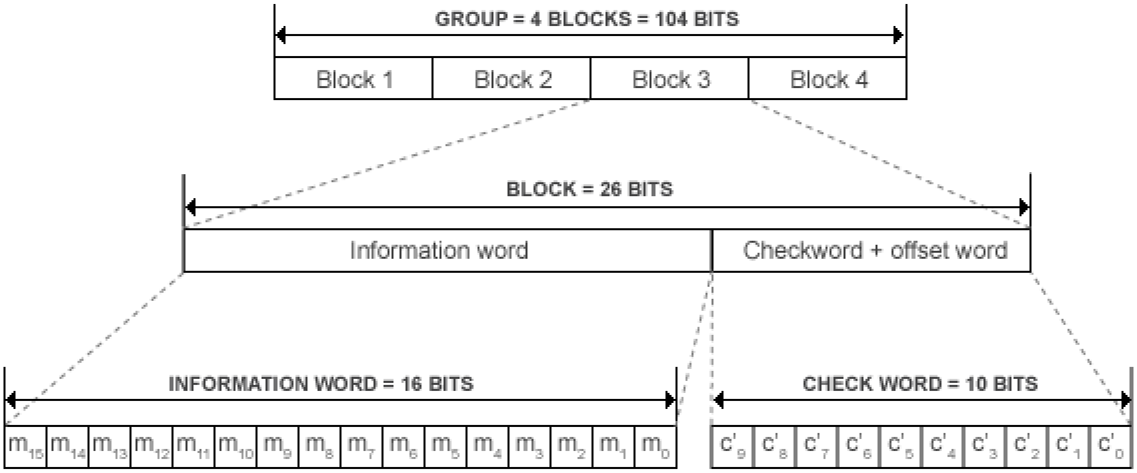
Physical Layer

The physical layer (first layer) converts the data-link bits to an analog waveform by conducting differential encoding and biphase symbol encoding (Manchester encoding) and pulse-shaping filtering.

Data-Link Layer

The data-link layer (second layer) performs (26,16) cyclic encoding shortened from (341,331) encoding [1]. The second layer is responsible for error detection, error correction, and the establishment of group-level synchronization. Each group of RDS/RBDS frames contains four blocks of 26 bits (that is 104 bits) each. Each block contains an information word and a check word. Each information word contains 16 bits, and each check word contains 10 bits.

Here is the baseband coding structure for the RDS/RBDS waveform. For more details, see [1].


GROUP = 4 BLOCKS = 104 BITS
Block 1
Block 2
Block 3
Block 4
BLOCK = 26 BITS
Information word
Checkword + offset word
INFORMATION WORD = 16 BITS
m15 m14 m13 m12 m11 m10 m9 m8 m7 m6 m5 m4 m3 m2 m1 m0
CHECK WORD = 10 BITS
c'9 c'8 c'7 c'6 c'5 c'4 c'3 c'2 c'1 c'0

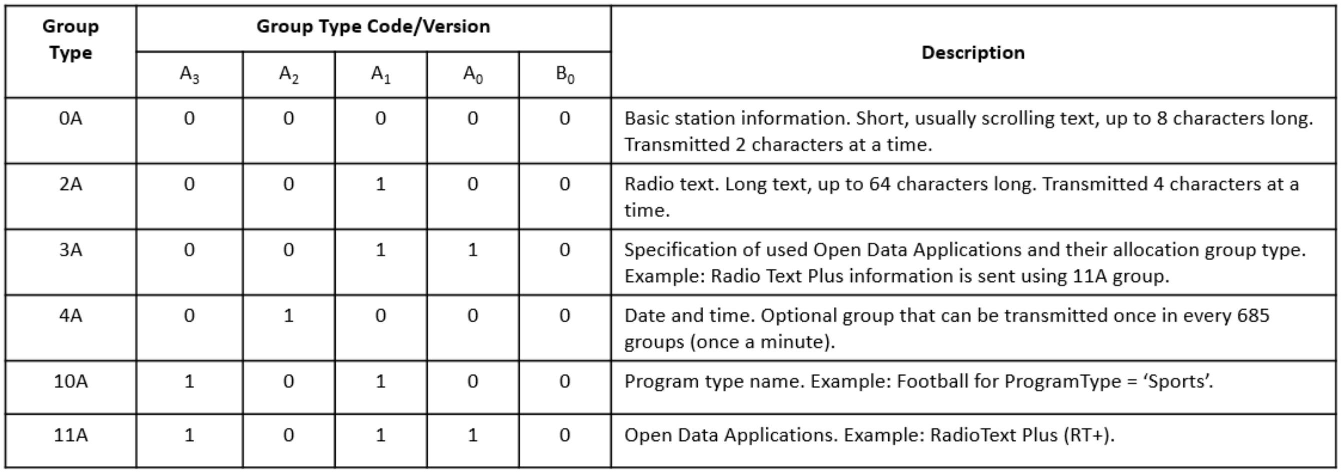
For each block, a unique offset word is modulo-2 added to the checkword bits. The added offset word provides a group and block synchronization system in the receiver (decoder). Because the addition of the offset is reversible in the decoder, the normal additive error-correcting and detecting properties of the basic code are unaffected.

Session and Application Layer

The first block in every group contains a program identification (PI) code. The first four bits of the second block of every group are allocated to a four-bit code. This code specifies the application of the group. Groups are referred to as types 0–15 according to the binary weighting A_3 = 8, A_2 = 4, A_1 = 2, A_0 = 1. The fifth bit of the second block, B_0, defines the version of the application. If B_0 = 0, the version of the group is A. The PI code in this version is inserted into block 1 only. Example group types include 0A, 1A, 2A, 3A, and 4A.

The Program Type code and Traffic Program Identification (PI) occupy fixed locations in block 2 of every group.

Group Types



Group Type	Group Type Code/Version					Description
	A_3	A_2	A_1	A_0	B_0	
0A	0	0	0	0	0	Basic station information. Short, usually scrolling text, up to 8 characters long. Transmitted 2 characters at a time.
2A	0	0	1	0	0	Radio text. Long text, up to 64 characters long. Transmitted 4 characters at a time.
3A	0	0	1	1	0	Specification of used Open Data Applications and their allocation group type. Example: Radio Text Plus information is sent using 11A group.
4A	0	1	0	0	0	Date and time. Optional group that can be transmitted once in every 685 groups (once a minute).
10A	1	0	1	0	0	Program type name. Example: Football for ProgramType = 'Sports'.
11A	1	0	1	1	0	Open Data Applications. Example: RadioText Plus (RT+).

References

[1] National Radio Systems Committee. *United States RBDS Standard: Specification of the radio broadcast data system (RBDS)*. Electronic Industries Association and National Association of Broadcasters. April 9, 1998.

[2] Westdeutscher Rundfunk WDR, Nokia, and Institut für Rundfunktechnik IRT. *RadioText Plus (RT+) Specification, Version 2.1.* 2006.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See "System Objects in MATLAB Code Generation" (MATLAB Coder).

In addition, the following limitations apply when you generate code that contains this System object or when you use this object in a MATLAB function block.

- The group type 4A cannot be transmitted in the generated code.
- The `registerODA` method is not supported.
- The `ProgramType` property is not tunable.

See Also

Objects

`comm.FMBroadcastDemodulator` | `comm.FMBroadcastModulator`

Introduced in R2017a

callLettersToPICode

System object: `comm.RBDSWaveformGenerator`
Package: `comm`

Convert North-American call letters to binary PI code

Syntax

```
picode = callLettersToPICode(rbdsgen, callLetters)
```

Description

`picode = callLettersToPICode(rbdsgen, callLetters)` returns the 16-bit program identification (PI) code that corresponds to `callLetters`. Acceptable call letter formats are 3-character or 4-character vectors beginning with `'K'` or `'W'`.

Introduced in R2017a

registerODA

System object: `comm.RBDSWaveformGenerator`
Package: `comm`

Register a custom encoding implementation for an ODA

Syntax

```
registerODA(rbdsgen,odaID,group,handleMain,handle3A)
```

Description

`registerODA(rbdsgen,odaID,group,handleMain,handle3A)` associates the Open Data Application (ODA) specified by the hexadecimal ID `odaID`, with the type `group` groups generated by `rbdsgen`. The four 16-bit information words of these groups are generated by the function handle `handleMain`. The third information word of type 3A groups, which is ODA-specific, is generated by the function handle `handle3A`.

Introduced in R2017a

reset

System object: `comm.RBDSWaveformGenerator`
Package: `comm`

Reset states of RBDS waveform generator object

Syntax

```
reset(rbdsgen)
```

Description

`reset(rbdsgen)` resets the states of the `RBDSWaveformGenerator` object, `rbdsgen`.

Introduced in R2017a

step

System object: `comm.RBDSWaveformGenerator`
Package: `comm`

Generate RDS/RBDS waveform

Syntax

```
y = step(rbdsgen)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(rbdsgen)` outputs a frame of the baseband RDS/RBDS waveform in column vector `y`. The waveform contains the number of 104-bit groups, specified in the `GroupsPerFrame` property of the object. Each symbol is oversampled according to the `SamplesPerSymbol` property. Thus, the output length is `SamplesPerSymbol` × 104 × `GroupsPerFrame` samples. The object uses an internal scheduler to determine the order and frequency of the transmitted group types.

Note `rbdsgen` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2017a

comm.RectangularQAMDemodulator

Package: `comm`

(To be removed) Demodulate using rectangular QAM signal constellation

Note `comm.RectangularQAMDemodulator` will be removed in a future release. Use `qamdemod` instead. For more information, see “Compatibility Considerations”.

Description

The `RectangularQAMDemodulator` object demodulates a signal that was modulated using quadrature amplitude modulation with a constellation on a rectangular lattice.

To demodulate a signal that was modulated using quadrature amplitude modulation:

1. Define and set up your rectangular QAM demodulator object. See “Construction” on page 3-1247.
2. Call `step` to demodulate the signal according to the properties of `comm.RectangularQAMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RectangularQAMDemodulator` creates a demodulator System object, `H`. This object demodulates the input signal using the rectangular quadrature amplitude modulation (QAM) method.

`H = comm.RectangularQAMDemodulator(Name,Value)` creates a rectangular QAM demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.RectangularQAMDemodulator(M,Name,Value)` creates a rectangular QAM demodulator object, `H`. This object has the `ModulationOrder` property set to `M`, and the other specified properties set to the specified values.

Properties

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation as scalar value with a positive, integer power of two. The default is `16`.

`PhaseOffset`

Phase offset of constellation

Specify the phase offset of the signal constellation, in radians, as a real scalar value. The default is `0`.

`BitOutput`

Output data as bits

Specify whether the output consists of groups of bits or integer symbol values. When you set this property to true the step method outputs a column vector of bit values whose length equals **log2**(`ModulationOrder` on page 3-0) times the number of demodulated symbols. When you set this property to `false`, the `step` method outputs a column vector with a length equal to the input data vector. This vector contains integer symbol values between `0` and `ModulationOrder-1`. The default is `false`.

`SymbolMapping`

Constellation encoding

Specify how the object maps an integer or group of **log2**(`ModulationOrder` on page 3-0) bits to the corresponding symbol as one of `Binary` | `Gray` | `Custom`. The default is `Gray`. When you set this property to `Gray`, the object uses a Gray-coded signal constellation. When you set this property to `Binary`, the object uses a natural binary-coded constellation. When you set this property to `Custom`, the object uses the signal constellation defined in the `CustomSymbolMapping` on page 3-0 property.

`CustomSymbolMapping`

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is `0:15`. This property is a row or column vector with a size of `ModulationOrder` on page 3-0 and with unique integer values in the range [`0`, `ModulationOrder-1`]. The values must be of data type `double`. The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point. This property applies when you set the `SymbolMapping` on page 3-0 property to `Custom`.

`NormalizationMethod`

Constellation normalization method

Specify the method used to normalize the signal constellation as `Minimum distance between symbols` | `Average power` | `Peak power`. The default is `Minimum distance between symbols`.

`MinimumDistance`

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is `2`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Minimum distance between symbols`.

AveragePower

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is 1. This property applies when you set the NormalizationMethod on page 3-0 property to Average power.

PeakPower

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive, real, numeric scalar value. The default is 1. This property applies when you set the NormalizationMethod on page 3-0 property to Peak power.

DecisionMethod

Demodulation decision method

Specify the decision method the object uses as Hard decision | Log-likelihood ratio | Approximate log-likelihood ratio. The default is Hard decision. When you set the BitOutput on page 3-0 property to false the object always performs hard-decision demodulation. This property applies when you set the BitOutput property to true.

VarianceSource

Source of noise variance

Specify the source of the noise variance as Property | Input port. The default is Property. This property applies when you set the BitOutput on page 3-0 property to true and the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio.

Variance

Noise variance

Specify the variance of the noise as a positive, real scalar value. The default is 1. If this value is very small (i.e., SNR is very high), log-likelihood ratio (LLR) computations may yield Inf or -Inf. This result occurs because the LLR algorithm computes the exponential of very large or very small numbers using finite-precision arithmetic. In such cases, using approximate LLR is recommended because its algorithm does not compute exponentials. This property applies when you set the BitOutput on page 3-0 property to true, the DecisionMethod on page 3-0 property to Log-likelihood ratio or Approximate log-likelihood ratio, and the VarianceSource on page 3-0 property to Property. This property is tunable.

OutputDataType

Data type of output

Specify the output data type as Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32. The default is Full precision.

This property applies only when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. In this case, when the OutputDataType on page 3-0 property is set to Full precision, and the input data type is single- or double-precision, the output data has the same data type as the input.

When the input data is of a fixed-point type, the output data type behaves as if you had set the OutputDataType property to Smallest unsigned integer.

When you set the BitOutput property to true and the DecisionMethod property to Hard Decision, then logical data type becomes a valid option.

When you set the BitOutput property to true and the DecisionMethod property to Log-likelihood ratio or Approximate log-likelihood ratio, the output data type is the same as that of the input. In this case, that data type can only be single- or double-precision.

Fixed-Point Properties

FullPrecisionOverride

Full precision override for fixed-point arithmetic

Specify whether to use full precision rules. If you set FullPrecisionOverride to true, which is the default, the object computes all internal arithmetic and output data types using full precision rules. These rules provide the most accurate fixed-point numerics. It also turns off the display of other fixed-point properties because they do not apply individually. These rules guarantee that no quantization occurs within the object. Bits are added, as needed, to ensure that no roundoff or overflow occurs. If you set FullPrecisionOverride to false, fixed-point data types are controlled through individual fixed-point property settings. For more information, see “Full Precision for Fixed-Point System Objects” on page 3-1252.

DerotateFactorDataType

Data type of derotate factor

Specify the derotate factor data type as Same word length as input | Custom. The default is Same word length as input. This property applies when you set the BitOutput on page 3-0 property to false, or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision. The object uses the derotate factor in the computations only when the step method input is of a fixed-point type and the PhaseOffset on page 3-0 property has a value that is not a multiple of $\pi/2$.

CustomDerotateFactorDataType

Fixed-point data type of derotate factor

Specify the derotate factor fixed-point type as an unscaled numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the DerotateFactorDataType on page 3-0 property to Custom.

DenormalizationFactorDataType

Data type of denormalization factor

Specify the denormalization factor data type as Same word length as input | Custom. The default is Same word length as input. This property applies when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision.

CustomDenormalizationFactorDataType

Fixed-point data type of denormalization factor

Specify the denormalization factor fixed-point type as an unscaled numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the DenormalizationFactorDataType on page 3-0 property to Custom.

ProductDataType

Data type of product

Specify the product data type as Full precision | Custom. The default is Full precision. This property applies when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision.

CustomProductDataType

Fixed-point data type of product

Specify the product fixed-point type as an unscaled numerictype object with a signedness of Auto. The default is numerictype([],32). This property applies when you set the ProductDataType on page 3-0 property to Custom.

ProductRoundingMethod

Rounding of fixed-point numeric value of product

Specify the product rounding method as Ceiling | Convergent | Floor | Nearest | Round | Simplest | Zero. The default is Floor. This property applies when the object is not in a full precision configuration, when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision.

ProductOverflowAction

Action when fixed-point numeric value of product overflows

Specify the product overflow action as Wrap | Saturate. The default is Wrap. This property applies when the object is not in a full precision configuration, when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision.

SumDataType

Data type of sum

Specify the sum data type as Full precision | Same as product | Custom. The default is Full precision. This property applies when you set the FullPrecisionOverride on page 3-0 property to false, when you set the BitOutput on page 3-0 property to false or when you set the BitOutput property to true and the DecisionMethod on page 3-0 property to Hard decision.

CustomSumDataType

Fixed-point data type of sum

Specify the sum fixed-point type as an unscaled `numerictype` object with a signedness of Auto. The default is `numerictype([],32)`. This property applies when you set the `FullPrecisionOverride` on page 3-0 property to false or when you set the `SumDataType` on page 3-0 property `Custom`.

Methods

constellation	(To be removed) Calculate or plot ideal signal constellation
step	(To be removed) Demodulate using rectangular QAM method

Common to All System Objects	
`release`	Allow System object property value changes

More About

Full Precision for Fixed-Point System Objects

`FullPrecisionOverride` is a convenience property that, when you set to `true`, automatically sets the appropriate properties for an object to use full-precision to process fixed-point input.

For System objects, full precision, fixed-point operation refers to growing just enough additional bits to compute the ideal full precision result. This operation has no minimum or maximum range overflow nor any precision loss due to rounding or underflow. It is also independent of any hardware-specific settings. The data types chosen are based only on known data type ranges and not on actual numeric values. Full precision for System objects does not optimize coefficient values. When you set the `FullPrecisionOverride` property to `true`, the other fixed-point properties it controls no longer apply and any of their non-default values are ignored. These properties are also hidden. To specify individual fixed-point properties, first set `FullPrecisionOverride` to `false`.

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM Demodulator Baseband block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.RectangularQAMDemodulator` will be removed in a future release. Use qamdemod instead.

Not recommended starting in R2018b

Constellation normalization by `PeakPower` and `AveragePower` (other than unit average power) as supported by `comm.RectangularQAMModulator` and `comm.RectangularQAMDemodulator` is not inherently provided by functions. This code shows you how to perform peak power and average power normalization using `qammod` and `qamdemod` functions.

Average Power Normalization for Hard Decision

Output shown here compares computation of average power normalization for hard decision using alternative approach. Functions used follow the output.

```
>> averagePowerReplacement(64)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1

>> averagePowerReplacement(32)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1
```

Functions shown here compares computation of average power normalization for hard decision using alternative approach.

```
function averagePowerReplacement(M)
% QAM alternative for "Average power" normalization method when using functions

    avgPow = 100;
    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);
    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj([0:M-1]');
    constellationFcn = qammod([0:M-1]', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);
    z1 = demodObj(y1);
    objModDemodOutputIsEqual = isequal(x, z1)

    % 2) qamdemod() demodulates modulator System object output
    y1ScaledForFcn = (2/minD) .* y1;
    z2 = qamdemod(y1ScaledForFcn, M);
    objModFcnDemodIsEqual = isequal(x, z2)

    % 3) Demodulator System object demodulates qammod() output
    y2 = qammod(x, M);
    y2ScaledForSO = (minD/2) .* y2;
    z3 = demodObj(y2ScaledForSO);
    fcnModObjDemodIsEqual = isequal(x, z3)

end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);

end
```

Peak Power Normalization for Hard Decision

Output shown here compares computation of peak power normalization for hard decision using alternative approach. Functions used follow the output.

```
>> peakPowerReplacement(16)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1

>> peakPowerReplacement(128)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1
```

Functions shown here compares computation of peak power normalization for hard decision using alternative approach.

```
function peakPowerReplacement(M)
% QAM alternative for "Peak power" normalization method when using functions

    pkPow = 5;
    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);
    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj([0:M-1]');
    constellationFcn = qammod([0:M-1]', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);
    z1 = demodObj(y1);
    objModDemodOutputIsEqual = isequal(x, z1)

    % 2) qamdemod() demodulates modulator System object output
    y1ScaledForFcn = (2/minD) .* y1;
    z2 = qamdemod(y1ScaledForFcn, M);
    objModFcnDemodIsEqual = isequal(x, z2)

    % 3) Demodulator System object demodulates qammod() output
    y2 = qammod(x, M);
    y2ScaledForSO = (minD/2) .* y2;
    z3 = demodObj(y2ScaledForSO);
    fcnModObjDemodIsEqual = isequal(x, z3)

end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);

end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);

end
```

Average Power Normalization for Approximate LLR

Output shown here compares computation of average power normalization for approximate LLR using alternative approach. Functions used follow the output.

```
>> averagePowerReplacementApproxLLR(16, 10, 1) % M=16, avgPow=10, snrdB=1
maxConstellationErr =
     0
maxOutputErr =
     0
>> averagePowerReplacementApproxLLR(32, 7, 20) % M=32, avgPow=7, snrdB=20
maxConstellationErr =
     0
maxOutputErr =
   8.5265e-14
>> averagePowerReplacementApproxLLR(64, 111, -2) % M=64, avgPow=111, snrdB=-2
maxConstellationErr =
     0
maxOutputErr =
   2.2204e-15
>> averagePowerReplacementApproxLLR(1024, 87, 33) % M=1024, avgPow=87, snrdB=33
maxConstellationErr =
     0
maxOutputErr =
   1.3642e-12
```

Functions shown here compares computation of average power normalization for approximate LLR using alternative approach.

```
function averagePowerReplacementApproxLLR(M, avgPow, snrdB)
% QAM alternative for "Average power" normalization method for
% Approximate LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Approximate log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Functions' output is same as System objects' output, with right
    % scaling
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Scale the received signal for the constellation used by function
    y2RecScaled = (2/minD) .* y2Rec;
    % Scale the noise variance appropriately
    z2 = qamdemod(y2RecScaled, M, 'OutputType', 'approxllr', ...
        'NoiseVariance', nv1 * (2/minD)^2);

    maxOutputErr = max(abs(z1-z2))
end
```

```
function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);
end
```

Peak Power Normalization for Approximate LLR

Output shown here compares computation of peak power normalization for approximate LLR using alternative approach. Functions used follow the output.

```
>> peakPowerReplacementApproxLLR(4, 2.5, 7) % M=4, pkPow=2.5, snrdB=7
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> peakPowerReplacementApproxLLR(16, 19, 0) % M=16, pkPow=19, snrdB=0
maxConstellationErr =
     0
maxOutputErr =
   4.4409e-15
>> peakPowerReplacementApproxLLR(128, 12, 4.4) % M=128, pkPow=12, snrdB=4.4
maxConstellationErr =
     0
maxOutputErr =
   2.6645e-15
>> peakPowerReplacementApproxLLR(256, 221, 16) % M=256, pkPow=221, snrdB=16
maxConstellationErr =
     0
maxOutputErr =
   2.8422e-14
```

Functions shown here compares computation of peak power normalization for approximate LLR using alternative approach.

```
function peakPowerReplacementApproxLLR(M, pkPow, snrdB)
% QAM alternative for "Peak power" normalization method for
% Approximate LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);
    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Approximate log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Functions' output is same as System objects' output, with right
    % scaling
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Scale the received signal for the constellation used by function
    y2RecScaled = (2/minD) .* y2Rec;
    % Scale the noise variance appropriately
    z2 = qamdemod(y2RecScaled, M, 'OutputType', 'approxllr', ...
        'NoiseVariance', nv1 * (2/minD)^2);

    maxOutputErr = max(abs(z1-z2))

end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);

end
```

Average Power Normalization for LLR

Output shown here compares computation of average power normalization for LLR using alternative approach. Functions used follow the output.

```
>> averagePowerReplacementLLR(16, 20, 1) % M=16, avgPow=20, snrdB=1
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
>> averagePowerReplacementLLR(32, 5.5, 15) % M=32, avgPow=5.5, snrdB=15
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> averagePowerReplacementLLR(64, 100, -2) % M=64, avgPow=100, snrdB=-2
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
```

```
>> averagePowerReplacementLLR(256, 117, 8) % M=256, avgPow=117, snrdB=8
maxConstellationErr =
     0
maxOutputErr =
   3.5527e-15
```

Functions shown here compares computation of average power normalization for LLR using alternative approach.

```
function averagePowerReplacementLLR(M, avgPow, snrdB)
% QAM alternative for "Average power" normalization method for
% LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Get the same output as System object using functions
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Create inputs required by utility function to compute LLR
    nBits = log2(M);
    bitwiseMapping = de2bi((0:M-1)', nBits, 'left-msb');
    % c0 contains indices of mapping which has 0 at various bit positions
    [c0, ~] = find(bitwiseMapping==0);
    c0 = reshape(int32(c0), M/2, nBits);
    % c1 contains indices of mapping which has 1 at various bit positions
    [c1, ~] = find(bitwiseMapping==1);
    c1 = reshape(int32(c1), M/2, nBits);

    z2 = comm.internal.utilities.computeLLRsim(y2Rec, M, nBits, ...
        scaledConstellationFcn, c0, c1, nv1);

    maxOutputErr = max(abs(z1-z2))
```

```
end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);
end
```

Peak Power Normalization for LLR

Output shown here compares computation of peak power normalization for LLR using alternative approach. Functions used follow the output.

```
>> peakPowerReplacementLLR(8, 21, 0) % M=8, pkPow=21, snrdB=0
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
>> peakPowerReplacementLLR(64, 7, 22) % M=64, pkPow=7, snrdB=22
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> peakPowerReplacementLLR(512, 1000, -5) % M=512, pkPow=1000, snrdB=-5
maxConstellationErr =
     0
maxOutputErr =
   2.6645e-15
>> peakPowerReplacementLLR(1024, 1, 6) % M=1024, pkPow=1, snrdB=6
maxConstellationErr =
     0
maxOutputErr =
   3.5527e-15
```

Functions shown here compares computation of peak power normalization for LLR using alternative approach.

```
function peakPowerReplacementLLR(M, pkPow, snrdB)
% QAM alternative for "Peak power" normalization method for
% LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))
    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Get the same output as System object using functions
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Create inputs required by utility function to compute LLR
    nBits = log2(M);
    bitwiseMapping = de2bi((0:M-1)', nBits, 'left-msb');
    % c0 contains indices of mapping which has 0 at various bit positions
    [c0, ~] = find(bitwiseMapping==0);
    c0 = reshape(int32(c0), M/2, nBits);
    % c1 contains indices of mapping which has 1 at various bit positions
    [c1, ~] = find(bitwiseMapping==1);
    c1 = reshape(int32(c1), M/2, nBits);

    z2 = comm.internal.utilities.computeLLRsim(y2Rec, M, nBits, ...
        scaledConstellationFcn, c0, c1, nv1);

    maxOutputErr = max(abs(z1-z2))
end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
`genqamdemod` | `qamdemod`

Objects
`comm.GeneralQAMDemodulator`

Introduced in R2012a

constellation

System object: comm.RectangularQAMDemodulator
Package: comm

(To be removed) Calculate or plot ideal signal constellation

Note comm.RectangularQAMDemodulator will be removed in a future release. Use qamdemod instead.

Syntax

```
y = constellation(h)
constellation(h)
```

Description

y = constellation(h) returns the numerical values of the constellation.

constellation(h) generates a constellation plot for the object.

Examples

Plot QAM Reference Constellations

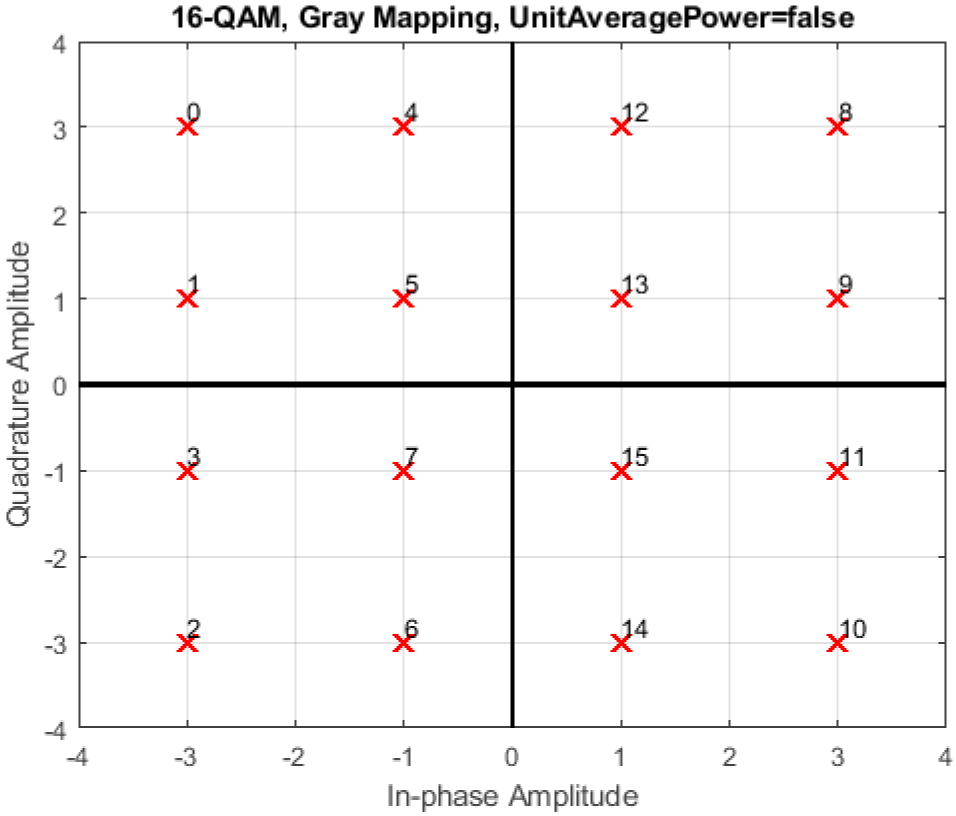
Plot QAM reference constellation using the qammod and qamdemod functions. Show that the 'PlotConstellation,true' Name,Value pair property works for both qammod and qamdemod functions. Also show the symbol ordering for Gray and binary code ordering by representing the data in binary format.

Create symbols for a 16-QAM modulator.

```
M = 16; % For 16-QAM
refSym = (0:M-1)';
```

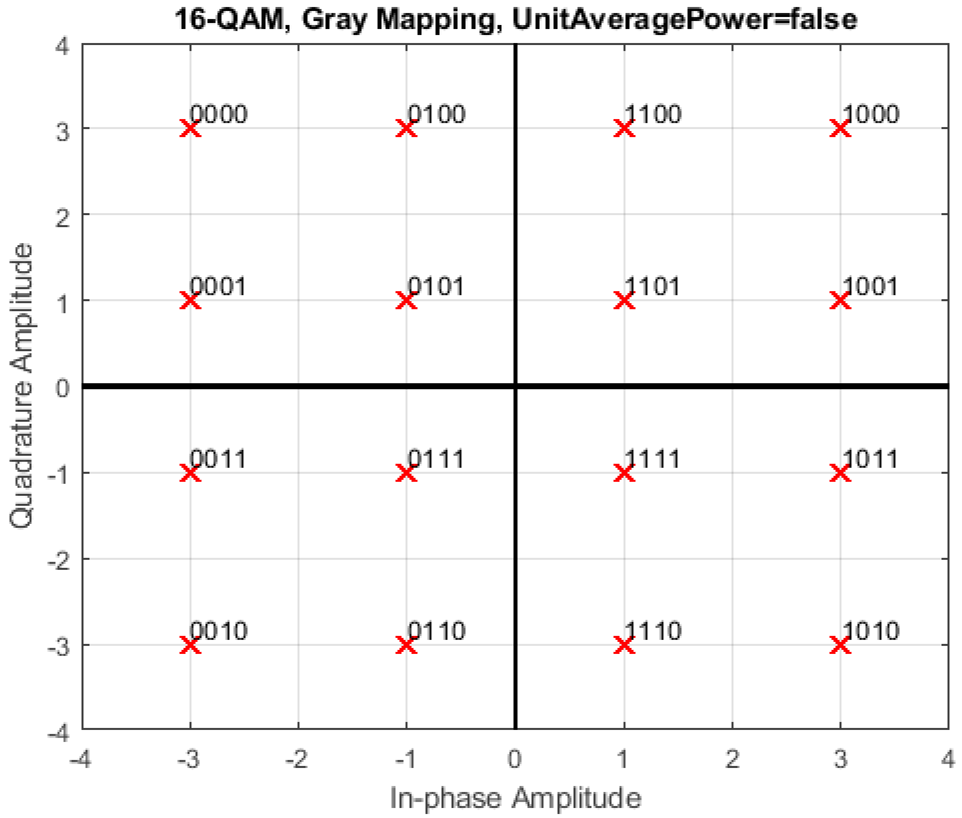
Plot the reference constellation using the qammod function.

```
qammod(refSym,M,'PlotConstellation',true);
```



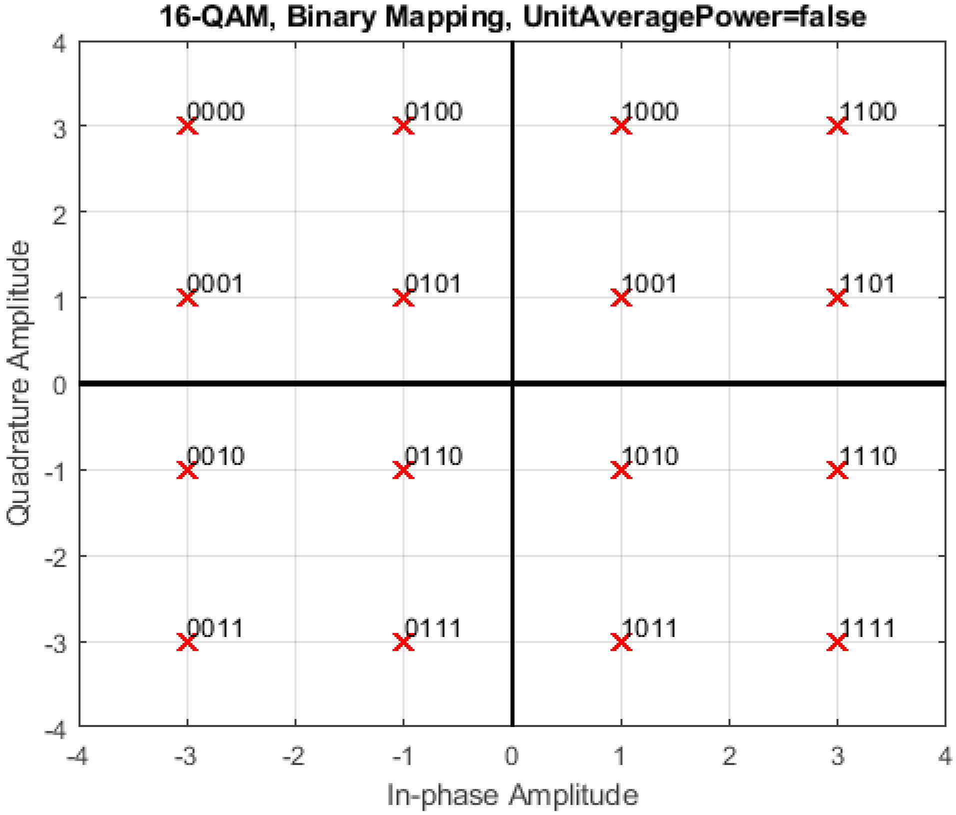
The default symbol order is Gray code ordering. To highlight the Gray symbol mapping, replot the reference constellation using binary input type. When you specify `'InputType','bit'`, the input signal must contain binary values, and the number of rows must be an integer multiple of `log2(M)`. Tranpose the input vector so that the input symbols map to the column vectors.

```
biRefSym = de2bi(refSym);
qammod(biRefSym',M,'PlotConstellation',true,'InputType','bit');
```



Replot the reference constellation using binary-coded symbol ordering.

```
biRefSym = de2bi(refSym);
qammod(biRefSym',M,'bin','PlotConstellation',true,'InputType','bit');
```

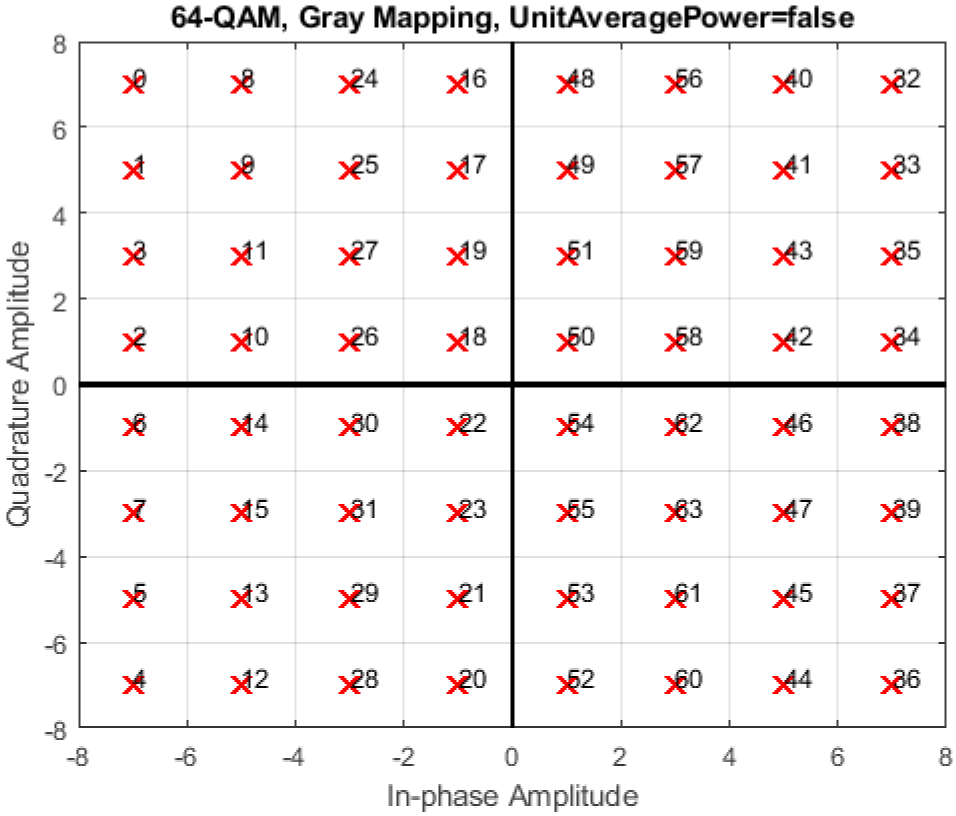


Create symbols for a 64-QAM modulator.

```
M = 64; % For 64-QAM
refSym = (0:M-1);
```

Plot the reference constellation using the `qamdemod` function.

```
qamdemod(refSym,M,'PlotConstellation',true);
```


64-QAM, Gray Mapping, UnitAveragePower=false
Quadrature Amplitude
In-phase Amplitude
8
6
4
2
0
-2
-4
-6
-8
-8
-6
-4
-2
0
2
4
6
8
0 8 24 16 48 56 40 32
1 9 25 17 49 57 41 33
3 11 27 19 51 59 43 35
2 10 26 18 50 58 42 34
6 14 30 22 54 62 46 38
7 15 31 23 55 63 47 39
5 13 29 21 53 61 45 37
4 12 28 20 52 60 44 36

step

System object: comm.RectangularQAMDemodulator
Package: comm

(To be removed) Demodulate using rectangular QAM method

Note comm.RectangularQAMDemodulator will be removed in a future release. Use qamdemod instead.

Syntax

```
Y = step(H,X)
Y = step(H,X,VAR)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) demodulates the input data, X, with the rectangular QAM demodulator System object, H, and returns, Y. Input X must be a scalar or a column vector with double or single precision data type. When ModulationOrder is an even power of two and you set the BitOutput property to false or, when you set the DecisionMethod to Hard decision and the BitOutput property to true, the data type of the input can also be signed integer, or signed fixed point (fi objects). Depending on the BitOutput property value, output Y can be integer or bit valued.

Y = step(H,X,VAR) uses soft decision demodulation and noise variance VAR. This syntax applies when you set the BitOutput property to true, the DecisionMethod property to Approximate log-likelihood ratioor Log-likelihood ratio, and the VarianceSource property to Input port. The data type of input VAR must be double or single precision.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.RectangularQAMModulator

Package: comm

(To be removed) Modulate using rectangular QAM signal constellation

Note comm.RectangularQAMModulator will be removed in a future release. Use qammod instead. For more information, see “Compatibility Considerations”.

Description

The RectangularQAMModulator object modulates using M-ary quadrature amplitude modulation with a constellation on a rectangular lattice. The output is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal.

To modulate a signal using quadrature amplitude modulation:

1. Define and set up your rectangular QAM modulator object. See “Construction” on page 3-1269.
2. Call step to modulate the signal according to the properties of comm.RectangularQAMModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.RectangularQAMModulator creates a modulator object, H. This object modulates the input using the rectangular quadrature amplitude modulation (QAM) method.

H = comm.RectangularQAMModulator(Name,Value) creates a rectangular QAM modulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.RectangularQAMModulator(M,Name,Value) creates a rectangular QAM modulator object, H. This object has the ModulationOrder property set to M, and the other specified properties set to the specified values.

Properties

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation as scalar value that is a positive integer power of two. The default is 16.

PhaseOffset

Phase offset of constellation

Specify the phase offset of the signal constellation, in radians, as a real scalar value. The default is 0.

BitInput

Assume bit inputs

Specify whether the input is bits or integers. The default is false. When you set this property to true, the step method input requires a column vector of bit values. The length of this vector must be an integer multiple of **log2**(ModulationOrder on page 3-0). This vector contains bit representations of integers between 0 and ModulationOrder–1. When you set this property to false, the step method input must be a column vector of integer symbol values between 0 and ModulationOrder–1.

SymbolMapping

Constellation encoding

Specify how the object maps an integer or group of **log2**(ModulationOrder on page 3-0) input bits to the corresponding symbol as Binary | Gray | Custom. The default is Gray. When you set this property to Gray, the System object uses a Gray-coded signal constellation. When you set this property to Binary, the object uses a natural binary-coded constellation. When you set this property to Custom, the object uses the signal constellation defined in the CustomSymbolMapping on page 3-0 property.

CustomSymbolMapping

Custom constellation encoding

Specify a custom constellation symbol mapping vector. The default is 0:15. This property is a row or column vector with a size of ModulationOrder on page 3-0 . This vector has unique integer values in the range [0, ModulationOrder–1]. These values must be of data type double. The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point. This property applies when you set the SymbolMapping on page 3-0 property to Custom.

NormalizationMethod

Constellation normalization method

Specify the method used to normalize the signal constellation as Minimum distance between symbols | Average power | Peak power. The default is Minimum distance between symbols.

MinimumDistance

Minimum distance between symbols

Specify the distance between two nearest constellation points as a positive, real, numeric scalar value. The default is 2. This property applies when you set the NormalizationMethod on page 3-0 property to Minimum distance between symbols.

`AveragePower`

Average power of constellation

Specify the average power of the symbols in the constellation as a positive, real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Average power`.

`PeakPower`

Peak power of constellation

Specify the maximum power of the symbols in the constellation as a positive real, numeric scalar value. The default is `1`. This property applies when you set the `NormalizationMethod` on page 3-0 property to `Peak power`.

`OutputDataType`

Data type of output

Specify the output data type as `double` | `single` | `Custom`. The default is `double`.

Fixed-Point Properties

`CustomOutputDataType`

Fixed-point data type of output

Specify the output fixed-point type as a `numerictype` object with a signedness of Auto. The default is `numerictype([],16)`. This property applies when you set the `OutputDataType` on page 3-0 property to `Custom`.

Methods

constellation	(To be removed) Calculate or plot ideal signal constellation
step	(To be removed) Modulate using rectangular QAM method

Common to All System Objects	
`release`	Allow System object property value changes

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM Modulator Baseband block reference page. The object properties correspond to the block parameters.

Compatibility Considerations

`comm.RectangularQAMModulator` will be removed in a future release. Use qammod instead.
Not recommended starting in R2018b

Constellation normalization by `PeakPower` and `AveragePower` (other than unit average power) as supported by `comm.RectangularQAMModulator` and `comm.RectangularQAMDemodulator` is not inherently provided by functions. This code shows you how to perform peak power and average power normalization using `qammod` and `qamdemod` functions.

Average Power Normalization for Hard Decision

Output shown here compares computation of average power normalization for hard decision using alternative approach. Functions used follow the output.

```
>> averagePowerReplacement(64)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1

>> averagePowerReplacement(32)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1
```

Functions shown here compares computation of average power normalization for hard decision using alternative approach.

```
function averagePowerReplacement(M)
% QAM alternative for "Average power" normalization method when using functions

    avgPow = 100;
    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);
    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj([0:M-1]');
    constellationFcn = qammod([0:M-1]', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);
    z1 = demodObj(y1);
    objModDemodOutputIsEqual = isequal(x, z1)

    % 2) qamdemod() demodulates modulator System object output
    y1ScaledForFcn = (2/minD) .* y1;
    z2 = qamdemod(y1ScaledForFcn, M);
    objModFcnDemodIsEqual = isequal(x, z2)
    % 3) Demodulator System object demodulates qammod() output
    y2 = qammod(x, M);
    y2ScaledForSO = (minD/2) .* y2;
    z3 = demodObj(y2ScaledForSO);
    fcnModObjDemodIsEqual = isequal(x, z3)

end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);

end
```

Peak Power Normalization for Hard Decision

Output shown here compares computation of peak power normalization for hard decision using alternative approach. Functions used follow the output.

```
>> peakPowerReplacement(16)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1

>> peakPowerReplacement(128)
maxConstellationErr =
     0
objModDemodOutputIsEqual =
  logical
   1
objModFcnDemodIsEqual =
  logical
   1
fcnModObjDemodIsEqual =
  logical
   1
```

Functions shown here compares computation of peak power normalization for hard decision using alternative approach.

```
function peakPowerReplacement(M)
% QAM alternative for "Peak power" normalization method when using functions

    pkPow = 5;
    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);
    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj([0:M-1]');
    constellationFcn = qammod([0:M-1]', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);
    z1 = demodObj(y1);
    objModDemodOutputIsEqual = isequal(x, z1)

    % 2) qamdemod() demodulates modulator System object output
    y1ScaledForFcn = (2/minD) .* y1;
    z2 = qamdemod(y1ScaledForFcn, M);
    objModFcnDemodIsEqual = isequal(x, z2)

    % 3) Demodulator System object demodulates qammod() output
    y2 = qammod(x, M);
    y2ScaledForSO = (minD/2) .* y2;
    z3 = demodObj(y2ScaledForSO);
    fcnModObjDemodIsEqual = isequal(x, z3)

end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);

end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);

end
```

Average Power Normalization for Approximate LLR

Output shown here compares computation of average power normalization for approximate LLR using alternative approach. Functions used follow the output.

```
>> averagePowerReplacementApproxLLR(16, 10, 1) % M=16, avgPow=10, snrdB=1
maxConstellationErr =
     0
maxOutputErr =
     0
```

```
>> averagePowerReplacementApproxLLR(32, 7, 20) % M=32, avgPow=7, snrdB=20
maxConstellationErr =
     0
maxOutputErr =
   8.5265e-14
>> averagePowerReplacementApproxLLR(64, 111, -2) % M=64, avgPow=111, snrdB=-2
maxConstellationErr =
     0
maxOutputErr =
   2.2204e-15
>> averagePowerReplacementApproxLLR(1024, 87, 33) % M=1024, avgPow=87, snrdB=33
maxConstellationErr =
     0
maxOutputErr =
   1.3642e-12
```

Functions shown here compares computation of average power normalization for approximate LLR using alternative approach.

```
function averagePowerReplacementApproxLLR(M, avgPow, snrdB)
% QAM alternative for "Average power" normalization method for
% Approximate LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Approximate log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Functions' output is same as System objects' output, with right
    % scaling
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Scale the received signal for the constellation used by function
    y2RecScaled = (2/minD) .* y2Rec;
    % Scale the noise variance appropriately
    z2 = qamdemod(y2RecScaled, M, 'OutputType', 'approxllr', ...
        'NoiseVariance', nv1 * (2/minD)^2);
    maxOutputErr = max(abs(z1-z2))
end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);
end
```

Peak Power Normalization for Approximate LLR

Output shown here compares computation of peak power normalization for approximate LLR using alternative approach. Functions used follow the output.

```
>> peakPowerReplacementApproxLLR(4, 2.5, 7) % M=4, pkPow=2.5, snrdB=7
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> peakPowerReplacementApproxLLR(16, 19, 0) % M=16, pkPow=19, snrdB=0
maxConstellationErr =
     0
maxOutputErr =
   4.4409e-15
>> peakPowerReplacementApproxLLR(128, 12, 4.4) % M=128, pkPow=12, snrdB=4.4
maxConstellationErr =
     0
maxOutputErr =
   2.6645e-15
>> peakPowerReplacementApproxLLR(256, 221, 16) % M=256, pkPow=221, snrdB=16
maxConstellationErr =
     0
maxOutputErr =
   2.8422e-14
```

Functions shown here compares computation of peak power normalization for approximate LLR using alternative approach.

```
function peakPowerReplacementApproxLLR(M, pkPow, snrdB)
% QAM alternative for "Peak power" normalization method for
% Approximate LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Approximate log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Functions' output is same as System objects' output, with right
    % scaling
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Scale the received signal for the constellation used by function
    y2RecScaled = (2/minD) .* y2Rec;
    % Scale the noise variance appropriately
    z2 = qamdemod(y2RecScaled, M, 'OutputType', 'approxllr', ...
        'NoiseVariance', nv1 * (2/minD)^2);

    maxOutputErr = max(abs(z1-z2))

end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);

end
```

Average Power Normalization for LLR

Output shown here compares computation of average power normalization for LLR using alternative approach. Functions used follow the output.

```
>> averagePowerReplacementLLR(16, 20, 1) % M=16, avgPow=20, snrdB=1
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
>> averagePowerReplacementLLR(32, 5.5, 15) % M=32, avgPow=5.5, snrdB=15
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> averagePowerReplacementLLR(64, 100, -2) % M=64, avgPow=100, snrdB=-2
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
>> averagePowerReplacementLLR(256, 117, 8) % M=256, avgPow=117, snrdB=8
maxConstellationErr =
     0
maxOutputErr =
   3.5527e-15
```

Functions shown here compares computation of average power normalization for LLR using alternative approach.

```
function averagePowerReplacementLLR(M, avgPow, snrdB)
% QAM alternative for "Average power" normalization method for
% LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = avgPow2MinD(avgPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Average power', ...
        'AveragePower', avgPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Get the same output as System object using functions
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Create inputs required by utility function to compute LLR
    nBits = log2(M);
    bitwiseMapping = de2bi((0:M-1)', nBits, 'left-msb');
    % c0 contains indices of mapping which has 0 at various bit positions
    [c0, ~] = find(bitwiseMapping==0);
    c0 = reshape(int32(c0), M/2, nBits);
    % c1 contains indices of mapping which has 1 at various bit positions
    [c1, ~] = find(bitwiseMapping==1);
    c1 = reshape(int32(c1), M/2, nBits);
    z2 = comm.internal.utilities.computeLLRsim(y2Rec, M, nBits, ...
        scaledConstellationFcn, c0, c1, nv1);

    maxOutputErr = max(abs(z1-z2))

end

function minD = avgPow2MinD(avgPow, M)
% Average power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = (M - 1)/6;
    else
        % Cross QAM
        if (nBits > 4)
            sf = ((31 * M / 32) - 1) / 6;
        else
            sf = ((5 * M / 4) - 1) / 6;
        end
    end
    minD = sqrt(avgPow/sf);
end
```

Peak Power Normalization for LLR

Output shown here compares computation of peak power normalization for LLR using alternative approach. Functions used follow the output.

```
>> peakPowerReplacementLLR(8, 21, 0) % M=8, pkPow=21, snrdB=0
maxConstellationErr =
     0
maxOutputErr =
   8.8818e-16
>> peakPowerReplacementLLR(64, 7, 22) % M=64, pkPow=7, snrdB=22
maxConstellationErr =
     0
maxOutputErr =
   7.1054e-15
>> peakPowerReplacementLLR(512, 1000, -5) % M=512, pkPow=1000, snrdB=-5
maxConstellationErr =
     0
maxOutputErr =
   2.6645e-15
>> peakPowerReplacementLLR(1024, 1, 6) % M=1024, pkPow=1, snrdB=6
maxConstellationErr =
     0
maxOutputErr =
   3.5527e-15
```

Functions shown here compares computation of peak power normalization for LLR using alternative approach.

```
function peakPowerReplacementLLR(M, pkPow, snrdB)
% QAM alternative for "Peak power" normalization method for
% LLR output when using functions
%
% M - Modulation order. Must be supported by QAM modulator-demodulator
% avgPow - Average constellation power
% snrdB - SNR, in dB. AWGN channel adds noise to provide this SNR.

    minD = pkPow2MinD(pkPow, M);

    modObj = comm.RectangularQAMModulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow);

    % 1) The two constellations are same
    constellationSO = modObj((0:M-1)');
    constellationFcn = qammod((0:M-1)', M);
    scaledConstellationFcn = (minD/2) .* constellationFcn;
    err = constellationSO - scaledConstellationFcn;
    maxConstellationErr = max(abs(err))

    x = randi([0, M-1], 100, 1);

    y1 = modObj(x);

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y1Rec = awgn(y1, snrdB, 'measured', 'db');

    % noise variance
    nv = mean(abs(y1).^2) / 10^(snrdB/10);

    demodObj = comm.RectangularQAMDemodulator('ModulationOrder', M, ...
        'NormalizationMethod', 'Peak power', ...
        'PeakPower', pkPow, ...
        'BitOutput', true, ...
        'DecisionMethod', 'Log-likelihood ratio', ...
        'Variance', nv);

    z1 = demodObj(y1Rec);

    % 2) Get the same output as System object using functions
    y2 = qammod(x, M);
    % Scale function's output so that it matches System object output.
    y2Scaled = (minD/2) .* y2;

    % Add noise
    % Reset global rng stream for repeatable noise samples
    reset(RandStream.getGlobalStream);
    y2Rec = awgn(y2Scaled, snrdB, 'measured', 'db');

    % noise variance
    nv1 = mean(abs(y2Scaled).^2) / 10^(snrdB/10);

    % Create inputs required by utility function to compute LLR
    nBits = log2(M);
    bitwiseMapping = de2bi((0:M-1)', nBits, 'left-msb');
    % c0 contains indices of mapping which has 0 at various bit positions
    [c0, ~] = find(bitwiseMapping==0);
    c0 = reshape(int32(c0), M/2, nBits);
    % c1 contains indices of mapping which has 1 at various bit positions
    [c1, ~] = find(bitwiseMapping==1);
    c1 = reshape(int32(c1), M/2, nBits);

    z2 = comm.internal.utilities.computeLLRsim(y2Rec, M, nBits, ...
        scaledConstellationFcn, c0, c1, nv1);

    maxOutputErr = max(abs(z1-z2))
end

function minD = pkPow2MinD(pkPow, M)
% Peak power to minimum distance
    nBits = log2(M);
    if (mod(nBits,2)==0)
        % Square QAM
        sf = 0.5*M - sqrt(M) + 0.5;
    else
        % Cross QAM
        mBy32 = M/32;
        if (nBits > 4)
            sf = (13 * mBy32) - (5 * sqrt(mBy32)) + 0.5;
        else
            sf = (20 * mBy32) - (6 * sqrt(mBy32)) + 0.5;
        end
    end
    minD = sqrt(pkPow/sf);
end
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Functions
`genqammod` | `qammod`

Objects
`comm.GeneralQAMModulator`

Introduced in R2012a

constellation

System object: `comm.RectangularQAMModulator`
Package: `comm`

(To be removed) Calculate or plot ideal signal constellation

Note `comm.RectangularQAMModulator` will be removed in a future release. Use `qammod` instead.

Syntax

```
y = constellation(h)
constellation(h)
```

Description

`y = constellation(h)` returns the numerical values of the constellation.

`constellation(h)` generates a constellation plot for the object.

Examples

Plot QAM Reference Constellations

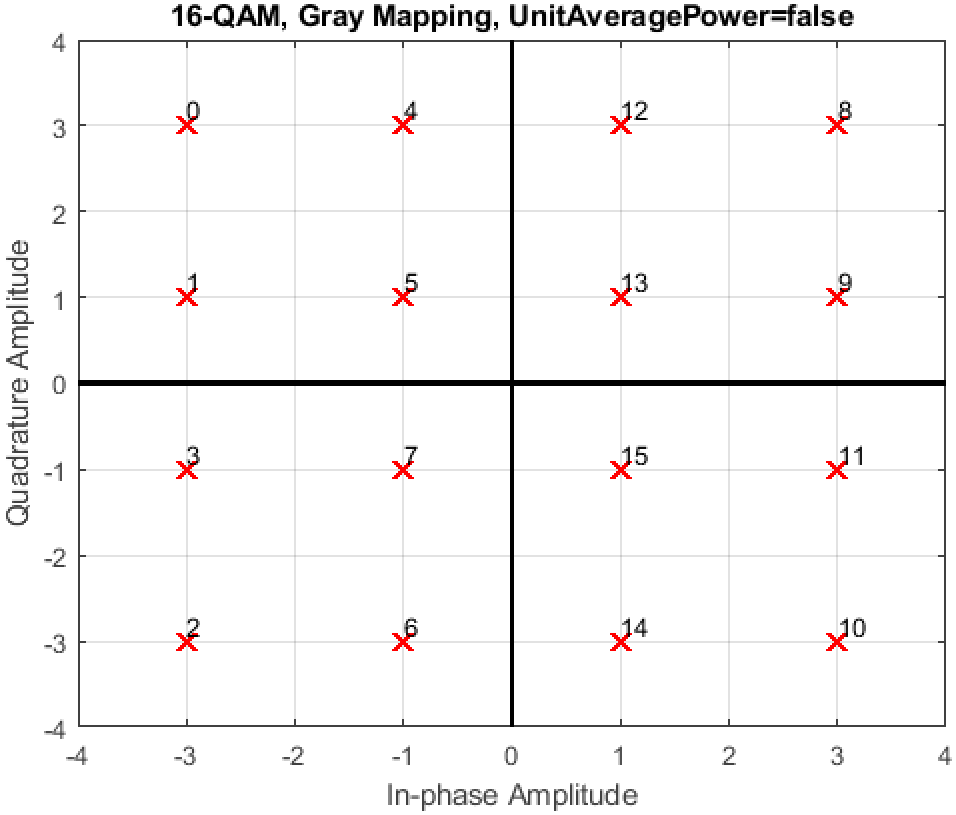
Plot QAM reference constellation using the `qammod` and `qamdemod` functions. Show that the `'PlotConstellation,true'` Name,Value pair property works for both `qammod` and `qamdemod` functions. Also show the symbol ordering for Gray and binary code ordering by representing the data in binary format.

Create symbols for a 16-QAM modulator.

```
M = 16; % For 16-QAM
refSym = (0:M-1)';
```

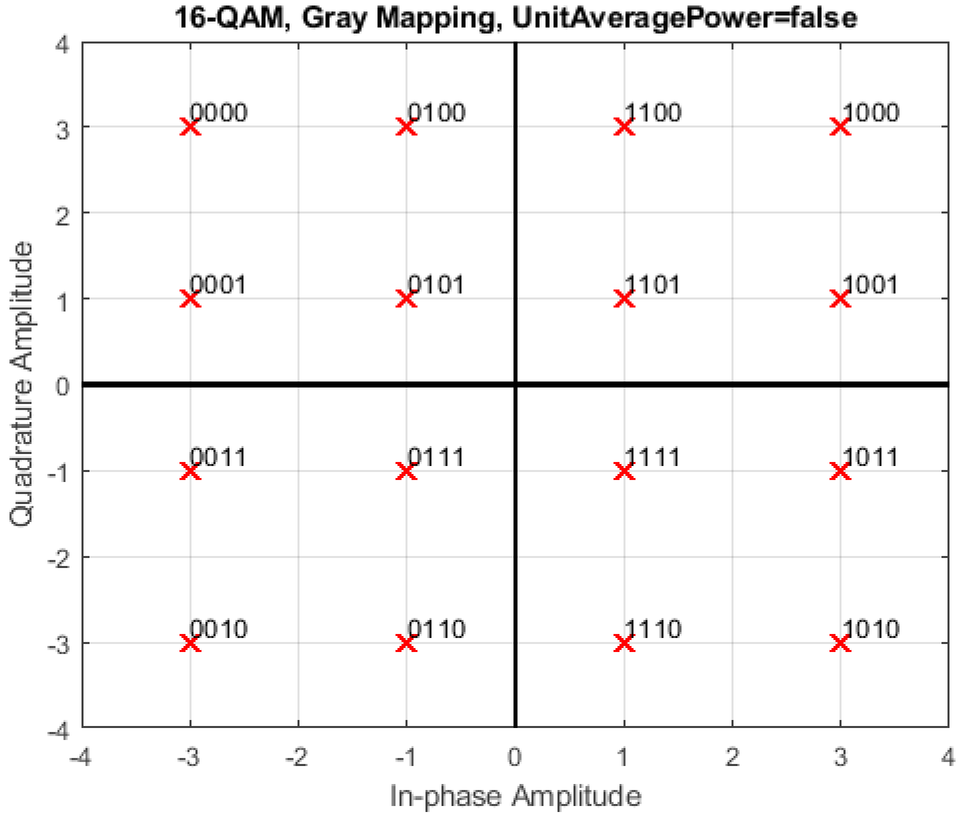
Plot the reference constellation using the `qammod` function.

```
qammod(refSym,M,'PlotConstellation',true);
```



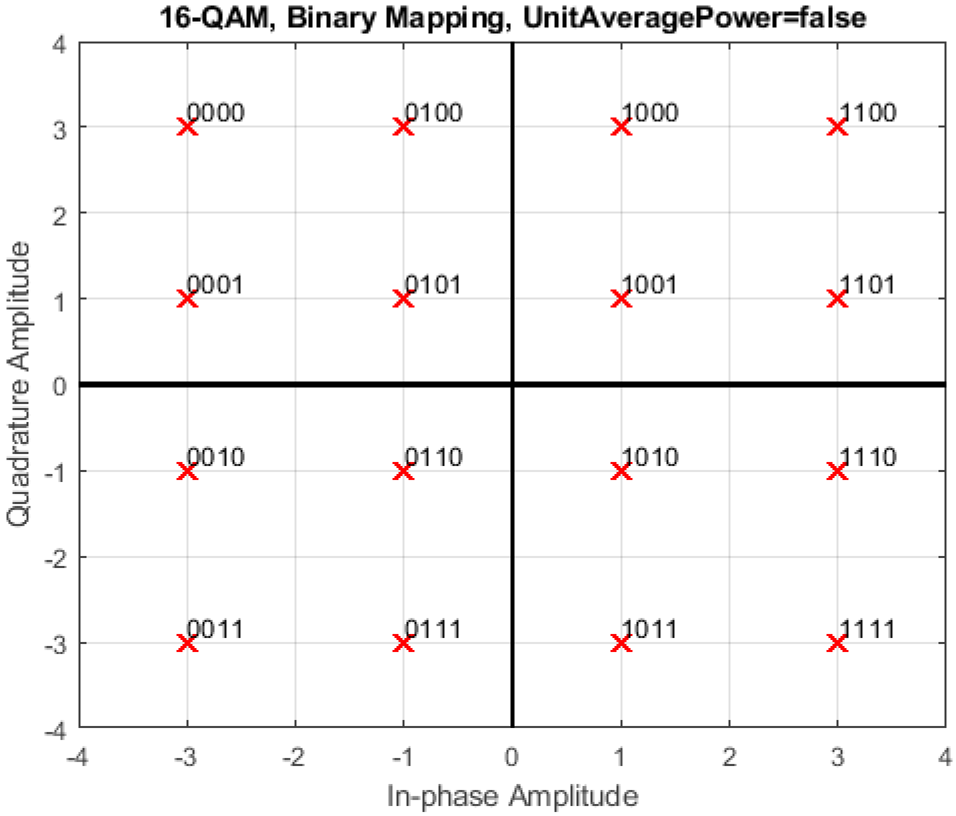
The default symbol order is Gray code ordering. To highlight the Gray symbol mapping, replot the reference constellation using binary input type. When you specify `'InputType','bit'`, the input signal must contain binary values, and the number of rows must be an integer multiple of `log2(M)`. Tranpose the input vector so that the input symbols map to the column vectors.

```
biRefSym = de2bi(refSym);
qammod(biRefSym',M,'PlotConstellation',true,'InputType','bit');
```



Replot the reference constellation using binary-coded symbol ordering.

```
biRefSym = de2bi(refSym);
qammod(biRefSym',M,'bin','PlotConstellation',true,'InputType','bit');
```

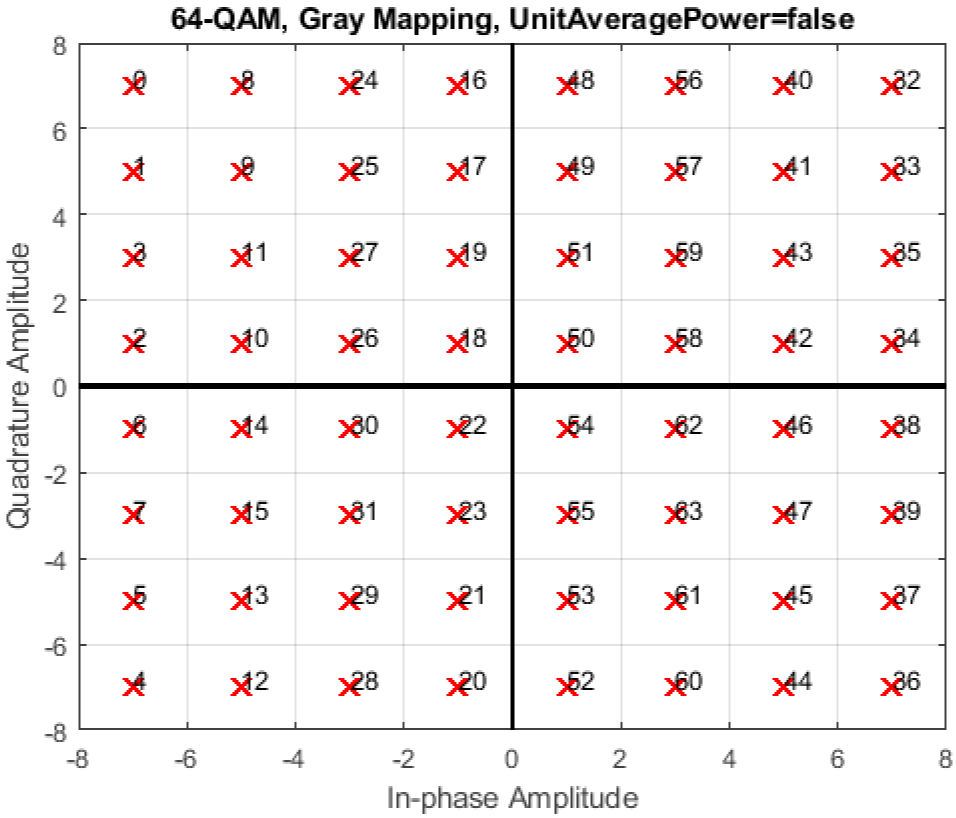


Create symbols for a 64-QAM modulator.

```
M = 64; % For 64-QAM
refSym = (0:M-1);
```

Plot the reference constellation using the `qamdemod` function.

```
qamdemod(refSym,M,'PlotConstellation',true);
```


64-QAM, Gray Mapping, UnitAveragePower=false
Quadrature Amplitude
In-phase Amplitude
8
6
4
2
0
-2
-4
-6
-8
-8
-6
-4
-2
0
2
4
6
8
0
8
24
16
48
56
40
32
1
9
25
17
49
57
41
33
3
11
27
19
51
59
43
35
2
10
26
18
50
58
42
34
6
14
30
22
54
62
46
38
7
15
31
23
55
63
47
39
5
13
29
21
53
61
45
37
4
12
28
20
52
60
44
36

step

System object: `comm.RectangularQAMModulator`
Package: `comm`

(To be removed) Modulate using rectangular QAM method

Note `comm.RectangularQAMModulator` will be removed in a future release. Use `qammod` instead.

Syntax

`Y = step(H,X)`

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` modulates input data, `X`, with the rectangular QAM modulator object, `H`. It returns the baseband modulated output, `Y`. Depending on the value of the `BitInput` property, input `X` can be an integer or bit valued column vector with numeric, logical, or fixed-point data types.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.RectangularQAMTCMDemodulator

Package: `comm`

Demodulate convolutionally encoded data mapped to rectangular QAM signal constellation

Description

The `RectangularQAMTCMDemodulator` object uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a rectangular QAM signal constellation.

To demodulate convolutionally encoded data mapped to a rectangular QAM signal constellation:

1. Define and set up your rectangular QAM TCM demodulator object. See "Construction" on page 3-1288.
2. Call `step` to demodulate the signal according to the properties of `comm.RectangularQAMTCMDemodulator`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.RectangularQAMTCMDemodulator` creates a trellis-coded, rectangular, quadrature amplitude (QAM TCM) demodulator System object, `H`. This object demodulates convolutionally encoded data that has been mapped to a rectangular QAM constellation.

`H = comm.RectangularQAMTCMDemodulator(Name,Value)` creates a rectangular, QAM TCM, demodulator object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.RectangularQAMTCMDemodulator(TRELLIS,Name,Value)` creates a rectangular QAM TCM demodulator object, `H`. This object has the TrellisStructure property set to `TRELLIS`, and the other specified properties set to the specified values.

Properties

`TrellisStructure`

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the `istrellis` function to check whether a structure is a valid trellis. The default is the result of `poly2trellis([3 1 1], [ 5 2 0 0; 0 0 1 0; 0 0 0 1])`.

TerminationMethod

Termination method of encoded frame

Specify the termination method as `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object saves the internal state metric at the end of each frame. The next frame uses the same state metric. The object treats each traceback path independently. If the input signal contains only one symbol, you should use `Continuous` mode.

When you set this property to `Truncated`, the object treats each input vector independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state.

When you set this property to `Terminated`, the object treats each input vector independently, and the traceback path always starts and ends in the all-zeros state.

TracebackDepth

Traceback depth for Viterbi decoder

Specify the scalar, integer number of trellis branches to construct each traceback path. The default is `21`. The Traceback depth parameter influences the decoding accuracy and delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

When you set the TerminationMethod property to `Continuous`, the decoding delay consists of TracebackDepth zero symbols or TracebackDepth×K zero bits for a rate K/N convolutional code.

When you set the TerminationMethod property to `Truncated` or `Terminated`, no output delay occurs and the traceback depth must be less than or equal to the number of symbols in each input vector.

ResetInputPort

Enable demodulator reset input

Set this property to `true` to enable an additional input to the `step` method. The default is `false`. When this additional reset input is a nonzero value, the internal states of the encoder reset to initial conditions. This property applies when you set the TerminationMethod property to `Continuous`.

ModulationOrder

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive, integer scalar value. The number of points must be `4`, `8`, `16`, `32`, or `64`. The default is `16`. The ModulationOrder property value must equal the number of possible input symbols to the convolutional decoder of the rectangular QAM TCM demodulator object. The `ModulationOrder` must equal 2^N for a rate K/N convolutional code.

OutputDataType

Data type of output

Specify output data type as `logical` | `double`. The default is `double`.

Methods

reset	Reset states of the rectangular QAM TCM demodulator object
step	Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate and Demodulate Using Rectangular 16-QAM TCM

Modulate and demodulate data using 16-QAM TCM in an AWGN channel. Estimate the BER.

Create QAM TCM modulator and demodulator System objects™.

```
hMod = comm.RectangularQAMTCMModulator;
hDemod = comm.RectangularQAMTCMDemodulator('TracebackDepth',16);
```

Create an AWGN channel object.

```
hAWGN = comm.AWGNChannel('EbNo',5);
```

Determine the delay through the QAM TCM demodulator. The demodulator uses the Viterbi algorithm to decode the TCM signal that was modulated using rectangular QAM. To accurately calculate the bit error rate, the delay through the decoder must be known.

```
bitsPerSymbol = log2(hDemod.TrellisStructure.numInputSymbols);
delay = hDemod.TracebackDepth*bitsPerSymbol;
```

Create an error rate calculator object with the `ReceiveDelay` property set to `delay`.

```
hErrorCalc = comm.ErrorRate('ReceiveDelay',delay);
```

Generate binary data and modulate with 16-QAM TCM. Pass the signal through an AWGN channel and demodulate. Calculate the error statistics. The loop runs until either 100 bit errors are encountered or 1e7 total bits are transmitted.

```
% Initialize the error results vector.
errStats = [0 0 0];

while errStats(2) < 100 && errStats(3) < 1e7
    % Transmit frames of 200 3-bit symbols
    txData = randi([0 1],600,1);
    % Modulate
    txSig = step(hMod,txData);
    % Pass through AWGN channel
    rxSig = step(hAWGN,txSig);
    % Demodulate
    rxData = step(hDemod,rxSig);
    % Collect error statistics
    errStats = step(hErrorCalc,txData,rxData);
end
```

Display the error data.

```
fprintf('Error rate = %4.2e\nNumber of errors = %d\n', ...
    errStats(1),errStats(2))
```

```
Error rate = 1.94e-03
Number of errors = 100
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM TCM Decoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.GeneralQAMTCMDemodulator` | `comm.RectangularQAMTCMModulator` | `comm.ViterbiDecoder`

Introduced in R2012a

reset

System object: `comm.RectangularQAMTCMDemodulator`
Package: `comm`

Reset states of the rectangular QAM TCM demodulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RectangularQAMTCMDemodulator` object, `H`.

step

System object: `comm.RectangularQAMTCMDemodulator`
Package: `comm`

Demodulate convolutionally encoded data mapped to rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` demodulates the rectangular QAM modulated input data, `X`, and uses the Viterbi algorithm to decode the resulting demodulated, convolutionally encoded bits. `X` must be a complex, double or single precision column vector. The step method outputs a demodulated, binary data column vector, `Y`. When the convolutional encoder represents a rate K/N code, the length of the output vector is K*L, where L is the length of the input vector, `X`.

`Y = step(H,X,R)` resets the decoder to the all-zeros state when you input a reset signal, `R` that is non-zero. `R` must be a double precision or logical, scalar integer. This syntax applies when you set the `ResetInputPort` property to true.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.RectangularQAMTCMModulator

Package: comm

Convolutionally encode binary data and map using rectangular QAM signal constellation

Description

The RectangularQAMTCMModulator object implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a rectangular QAM signal constellation.

To convolutionally encode binary data and map the result using a rectangular QAM constellation:

1. Define and set up your rectangular QAM TCM modulator object. See “Construction” on page 3-1294.
2. Call step to modulate the signal according to the properties of comm.RectangularQAMTCMModulator. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

H = comm.RectangularQAMTCMModulator creates a trellis-coded, rectangular, quadrature amplitude (QAM TCM) System object, H. This object convolutionally encodes a binary input signal and maps the result to a rectangular QAM constellation.

H = comm.RectangularQAMTCMModulator(Name,Value) creates a rectangular QAM TCM modulator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

H = comm.RectangularQAMTCMModulator(TRELLIS,Name,Value) creates a rectangular QAM TCM modulator object, H. This object has the TrellisStructure property set to TRELLIS and the other specified properties set to the specified values.

Properties

TrellisStructure

Trellis structure of convolutional code

Specify trellis as a MATLAB structure that contains the trellis description of the convolutional code. Use the istrellis function to check whether a structure is a valid trellis. The default is the result of poly2trellis([3 1 1], [5 2 0 0; 0 0 1 0; 0 0 0 1]).

`TerminationMethod`

Termination method of encoded frame

Specify the termination method as `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

When you set this property to `Continuous`, the object retains the encoder states at the end of each input vector for use with the next input vector.

When you set this property to `Truncated`, the object treats each input vector independently. The encoder is reset to the all-zeros state at the start of each input vector.

When you set this property to `Terminated`, the object treats each input vector independently. For each input vector, the object uses extra bits to set the encoder to the all-zeros state at the end of the vector. For a rate *K*/*N* code, the `step` method outputs the vector with a length given by $y = N \times {}^{(L+S)}\!/_{K}$, where *S* = *constraintLength*–1 (or, in the case of multiple constraint lengths, *S* = **sum**(*constraintLength*(*i*)–1)). *L* is the length of the input to the `step` method.

`ResetInputPort`

Enable modulator reset input

Set this property to true to enable an additional input to the step method. The default is `false`. When you set the reset input to the step method to a nonzero value, the object resets the encoder to the all-zeros state. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

`ModulationOrder`

Number of points in signal constellation

Specify the number of points in the signal constellation used to map the convolutionally encoded data as a positive integer scalar value equal to `4`, `8`, `16`, `32`, or `64`. The default is `16`. The value of the `ModulationOrder` on page 3-0 property must equal the number of possible output symbols from the convolutional encoder of the QAM TCM modulator. Thus, the value for the `ModulationOrder` property must equal 2^N for a rate *K*/*N* convolutional code.

`OutputDataType`

Data type of output

Specify the output data type as one of `double` | `single`. The default is `double`.

Methods

reset	Reset states of the rectangular QAM TCM modulator object
step	Convolutionally encode binary data and map using rectangular QAM constellation

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Modulate Data Using Rectangular QAM TCM

Modulate data using rectangular 16-QAM TCM modulation and display the scatter plot.

Generate random binary data. The length of the data vector must be an integer multiple of the number of input streams into the encoder, log2(8) = 3.

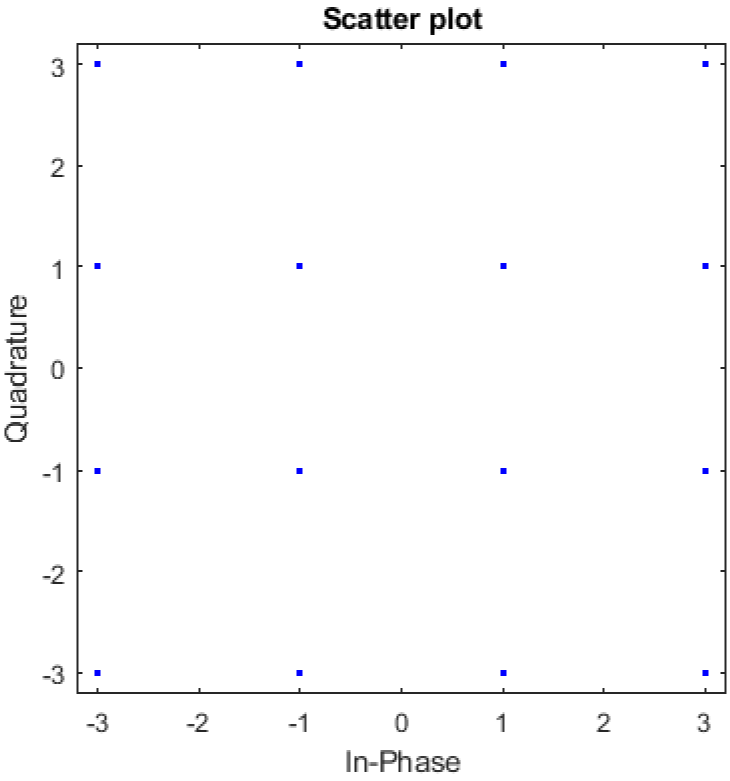
```
data = randi([0 1],3000,1);
```

Create a modulator System object™ and use its `step` function to modulate the data.

```
hMod = comm.RectangularQAMTCMModulator;
modData = step(hMod,data);
```

Plot the modulated data.

```
scatterplot(modData)
```



Algorithms

This object implements the algorithm, inputs, and outputs described on the Rectangular QAM TCM Encoder block reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.ConvolutionalEncoder` | `comm.GeneralQAMTCMModulator` | `comm.RectangularQAMTCMDemodulator`

Introduced in R2012a

reset

System object: `comm.RectangularQAMTCMModulator`
Package: `comm`

Reset states of the rectangular QAM TCM modulator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `RectangularQAMTCMModulator` object, `H`.

step

System object: comm.RectangularQAMTCMModulator
Package: comm

Convolutionally encode binary data and map using rectangular QAM constellation

Syntax

```
Y = step(H,X)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Y = step(H,X) convolutionally encodes and modulates the input data numeric or logical column vector X, and returns the encoded and modulated data, Y. X must be of data type numeric, logical, or unsigned fixed point of word length 1 (fi object). When the convolutional encoder represents a rate *K*/*N* code, the length of the input vector, X, must be *K*×*L*, for some positive integer *L*. The step method outputs a complex column vector, *Y*, of length *L*.

Y = step(H,X,R) resets the encoder of the rectangular QAM TCM modulator object to the all-zeros state when you input a non-zero reset signal, R. R must be a double precision or logical, scalar integer. This syntax applies when you set the ResetInputPort property to true.

Note obj specifies the System object on which to run this step method.

The object performs an initialization the first time the step method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the release method to unlock the object.

comm.RicianChannel

Package: `comm`

Filter input signal through multipath Rician fading channel

Description

The `comm.RicianChannel` System object filters an input signal through a multipath Rician fading channel. For more information on fading model processing, see Methodology for Simulating Multipath Fading Channels.

To filter an input signal using a multipath Rician fading channel:

1. Create the `comm.RicianChannel` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
ricianlchan = comm.RicianChannel
ricianlchan = comm.RicianChannel(Name,Value)
```

Description

`ricianlchan = comm.RicianChannel` creates a frequency-selective or frequency-flat multipath Rician fading channel System object. This object filters a real or complex input signal through the multipath channel to obtain the channel-impaired signal.

`ricianlchan = comm.RicianChannel(Name,Value)` sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, `comm.RicianChannel('SampleRate',2)` sets the input signal sample rate to 2.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`SampleRate` — Input signal sample rate
`1` (default) | positive scalar

Input signal sample rate in Hz, specified as a positive scalar.

Data Types: `double`

`PathDelays` — Discrete path delay

`0` (default) | scalar | row vector

Discrete path delay in seconds, specified as a scalar or row vector.

- When `PathDelays` is a scalar, the channel is frequency flat.
- When `PathDelays` is a vector, the channel is frequency selective.

Data Types: `double`

`AveragePathGains` — Average of the discrete paths

`0` (default) | scalar | row vector

Average gains of the discrete paths in decibels, specified as a scalar or row vector. `AveragePathGains` must be of the same size as the `PathDelays` property.

Data Types: `double`

`NormalizePathGains` — Normalize average path gains to 0 dB

`true` or `1` (default) | `false` or `0`

Normalize average path gains to 0 dB, specified as a logical `1` (`true`) or `0` (`false`).

- When `NormalizePathGains` is `true`, the fading processes are normalized so that the total power of the path gains, averaged over time, is 0 dB.
- When `NormalizePathGains` is `false`, the total power of the path gains is not normalized.

Data Types: `logical`

`KFactor` — Rician *K*-factor

`3` (default) | positive scalar | row vector of nonnegative values

Rician *K*-factor, specified as a positive scalar or row vector of nonnegative values. The vector must be the same length as the `PathDelays` property value.

- When `KFactor` is a scalar, the first discrete path is a Rician fading process with a Rician *K*-factor of `KFactor`. Any remaining discrete paths are independent Rayleigh fading processes.
- When `KFactor` is a row vector, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process with a Rician *K*-factor specified by that element. The discrete path corresponding to any zero-valued elements of the `KFactor` vector are Rayleigh fading processes. At least one element value must be nonzero.

Data Types: `double`

`DirectPathDopplerShift` — Doppler shift

`0` (default) | scalar | row vector

Doppler shift of the line-of-sight components of a multipath Rician fading channel, specified as a scalar or row vector. Units are in hertz. The `DirectPathDopplerShift` property must be of the same size as the `KFactor` property.

- When `DirectPathDopplerShift` is a scalar, the value represents the line-of-sight component Doppler shift of the first discrete path. This path exhibits a Rician fading process.

- When `DirectPathDopplerShift` is a row vector, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process. The corresponding element of `DirectPathDopplerShift` specifies the line-of-sight component Doppler shift of that discrete path.

Data Types: `double`

`DirectPathInitialPhase` — Initial phase

`0` (default) | scalar | row vector

Initial phase of the line-of-sight components of a multipath Rician fading channel, specified as a scalar or row vector. Units are in radians. The `DirectPathInitialPhase` property must be of the same size as the `KFactor` property.

- When `DirectPathInitialPhase` is a scalar, the value represents the line-of-sight component initial phase of the first discrete path. This path exhibits a Rician fading process.
- When `DirectPathInitialPhase` is a row vector, the discrete path corresponding to a positive element of the `KFactor` vector is a Rician fading process. The corresponding element of `DirectPathInitialPhase` specifies the line-of-sight component initial phase of that discrete path.

Data Types: `double`

`MaximumDopplerShift` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths, specified as a nonnegative scalar. Units are in hertz.

The maximum Doppler shift limit applies to each channel path. When you set this property to `0`, the channel remains static for the entire input. You can use the `reset` object function to generate a new channel realization. The `MaximumDopplerShift` property value must be smaller than `SampleRate`/10/f_c for each path. f_c represents the cutoff frequency factor of the path. For most Doppler spectrum types, the value of f_c is 1. For Gaussian and biGaussian Doppler spectrum types, f_c is dependent on the Doppler spectrum structure fields. For more details about how f_c is defined, see “Cutoff Frequency Factor” on page 3-1313.

Data Types: `double`

`DopplerSpectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | Doppler spectrum structure | 1-by-N_P cell array of Doppler spectrum structures

Doppler spectrum shape for all channel paths, specified as a Doppler spectrum structure or a 1-by-N_P cell array of Doppler spectrum structures. These Doppler spectrum structures must be outputs of the form returned from the `doppler` function. N_P is the number of discrete delay paths, that is, the length of the `PathDelays` property.

- When `DopplerSpectrum` is defined by a single Doppler spectrum structure, all paths have the same specified Doppler spectrum.
- When `DopplerSpectrum` is defined by a cell array of Doppler spectrum structures, each path has the Doppler spectrum specified by the corresponding structure in the cell array.

Options for the spectrum type are specified by the `specType` input to the `doppler` function. If you set the `FadingTechnique` property to `'Sum of sinusoids'`, you must set `DopplerSpectrum` to `doppler('Jakes')`.

Dependencies

To enable this property, set the `MaximumDopplerShift` property to a positive value. The `MaximumDopplerShift` property defines the maximum Doppler shift value permitted when specifying the Doppler spectrum.

Data Types: `struct` | `cell`

`FadingTechnique` — Channel model fading technique

`'Filtered Gaussian noise'` (default) | `'Sum of sinusoids'`

Channel model fading technique, specified as `'Filtered Gaussian noise'` or `'Sum of sinusoids'`.

Data Types: `char` | `string`

`NumSinusoids` — Number of sinusoids used

`48` (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

To enable this property, set the `FadingTechnique` property to `'Sum of sinusoids'`.

Data Types: `double`

`InitialTimeSource` — Source to control start time of fading process

`'Property'` (default) | `'Input port'`

Source to control the start time of the fading process, specified as `'Property'` or `'Input port'`.

- When `InitialTimeSource` is set to `'Property'`, use the `InitialTime` property to set the initial time offset.
- When `InitialTimeSource` is set to `'Input port'`, specify the start time of the fading process by using the `itime` input to the System object. The input value can change in consecutive calls to the System object.

Dependencies

This property applies when the `FadingTechnique` property to `'Sum of sinusoids'`.

Data Types: `string` | `char`

`InitialTime` — Initial time offset

`0` (default) | nonnegative scalar

Initial time offset, in seconds, for the fading model, specified as a nonnegative scalar. `InitialTime` must be greater than end time of the last frame. `InitialTime` is rounded up to the nearest sample position when it is not a multiple of 1/`SampleRate`.

Dependencies

To enable this property, set the `FadingTechnique` property to `'Sum of sinusoids'` and the `InitialTimeSource` property to `'Property'`.

Data Types: `double`

RandomStream — Source of random number stream

'Global stream' (default) | 'mt19937ar with seed'

Source of random number stream, specified as 'Global stream' or 'mt19937ar with seed'.

- 'Global stream' — The current global random number stream is used for normally distributed random number generation. In this case, the reset object function resets the channel filters only.
- 'mt19937ar with seed' — The mt19937ar algorithm is used for normally distributed random number generation. In this case, the reset object function resets the channel filters and reinitializes the random number stream to the value of the Seed property.

Data Types: string | char

Seed — Initial seed of mt19937ar random number stream

73 (default) | nonnegative integer

Initial seed of mt19937ar random number stream generator algorithm, specified as a nonnegative integer. When the reset object function is called, the mt19937ar random number stream is reinitialized to the Seedproperty value.

Dependencies

This property applies when you set the RandomStream property to 'mt19937ar with seed'.

Data Types: double

PathGainsOutputPort — Option to output path gains

false or 0 (default) | true or 1

Option to output path gains, specified as 0 (false) or 1 (true). Set this property to 1 (true) to output the channel path gains of the underlying fading process.

Data Types: logical

Visualization — Channel visualization

'Off' (default) | 'Impulse response' | 'Frequency response' | 'Impulse and frequency responses' | 'Doppler spectrum'

Channel visualization, specified as 'Off', 'Impulse response', 'Frequency response', 'Impulse and frequency responses', or 'Doppler spectrum'. For more information, see “Channel Visualization” on page 3-1313.

Data Types: string | char

SamplesToDisplay — Percentage of samples to display

'25%' (default) | '10%' | '50%' | '100%'

Percentage of samples to display, specified as '25%', '10%', '50%', or '100%'. Displaying fewer samples improves (decreases) the display update rate at the expense of decreasing the visualized precision.

Dependencies

To enable this property, set the Visualization property to 'Impulse response', 'Frequency response', or 'Impulse and frequency responses'.

Data Types: string | char

PathsForDopplerDisplay — Path for Doppler display

1 (default) | positive integer

Path for Doppler display, specified as a positive integer. Use this property to select the discrete path used in constructing a Doppler spectrum plot. The specified path must be an element of {1, 2, ..., N_P}. In this set, N_P is the number of discrete delay paths, that is, the length of the PathDelays property value.

Dependencies

To enable this property, set the Visualization property to 'Doppler spectrum'.

Data Types: double

Usage

Syntax

```
y = rayleighchan(x)
[y,pathgains] = rayleighchan(x)
___ = rayleighchan(x,itime)
```

Description

y = rayleighchan(x) filters input signal x through a multipath Rician fading channel and returns the output signal in y.

[y,pathgains] = rayleighchan(x) returns the channel path gains of the underlying multipath Rician fading process in pathgains. To enable this syntax set the PathGainsOutputPort property set to 1 (true).

___ = rayleighchan(x,itime) passes data through the multipath Rician fading channel beginning at the initial time specified by itime. To enable this syntax, set the FadingTechnique property to 'Sum of sinusoids' and the InitialTimeSource property to 'Input port'.

Input Arguments

x — Input signal

N_S-by-1 vector

Input signal, specified as an N_S-by-1 vector, where N_S is the number of samples.

Data Types: single | double
Complex Number Support: Yes

itime — Initial time

0 | nonnegative scalar

Initial time in seconds, specified as a nonnegative scalar. The itime input must be greater than the end time of the last frame. When itime is not a multiple of 1/SampleRate, it is rounded up to the nearest sample position.

Data Types: single | double

Output Arguments

y — Output signal
N_S-by-1 vector

Output signal, returned as an N_S-by-1 vector of complex values with the same data precision as the input signal `x`. N_S is the number of samples.

`pathgains` — Path gains
N_S-by-N_P array

Path gains, returned as an N_S-by-N_P array. N_S is the number of samples. N_P is the number of discrete delay paths, that is, the length of the `PathDelays` property value. `pathgains` contains complex values with the same precision as the input signal `x`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.RicianChannel`

info	Characteristic information about fading channel object

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Produce Same Rician Channel Outputs Using two Random Number Generation Methods

This example shows how to produce the same multipath Rician fading channel response by using two different methods for random number generation. The multipath Rician fading channel System object™ includes two methods for random number generation. You can use the current global stream or the mt19937ar algorithm with a specified seed. By interacting with the global stream, the System object can produce the same outputs from the two methods.

Create a PSK modulator System object to modulate randomly generated data.

```
pskModulator = comm.PSKModulator;
channelInput = pskModulator(randi([0 pskModulator.ModulationOrder-1],1024,1));
```

Create a multipath Rician fading channel System object, specifying the random number generation method as the my19937ar algorithm and the random number seed as 73.

```
ricianchan = comm.RicianChannel(...
    'SampleRate',1e6,...
    'PathDelays',[0.0 0.5 1.2]*1e-6,...
    'AveragePathGains',[0.1 0.5 0.2],...
    'KFactor',2.8,...
    'DirectPathDopplerShift',5.0,...
    'DirectPathInitialPhase',0.5,...
    'MaximumDopplerShift',50,...
    'DopplerSpectrum',doppler('Bell', 8),...
    'RandomStream','mt19937ar with seed', ...
    'Seed',73, ...
    'PathGainsOutputPort',true);
```

Filter the modulated data by using the multipath Rician fading channel System object.

```
[RicianChanOut1, RicianPathGains1] = ricianchan(channelInput);
```

Set the System object to use the global stream for random number generation.

```
release(ricianchan);
ricianchan.RandomStream = 'Global stream';
```

Set the global stream to have the same seed that was specified when creating the multipath Rician fading channel System object.

```
rng(73)
```

Filter the modulated data by using the multipath Rician fading channel System object again.

```
[RicianChanOut2,RicianPathGains2] = ricianchan(channelInput);
```

Verify that the channel and path gain outputs are the same for each of the two methods.

```
isequal(RicianChanOut1,RicianChanOut2)
```

```
ans = logical
   1
```

```
isequal(RicianPathGains1,RicianPathGains2)
```

```
ans = logical
   1
```

Display Impulse and Frequency Responses of Multipath Rician Fading Channel

This example shows how to create a frequency-selective multipath Rician fading channel and display its impulse and frequency responses.

Set the sample rate to 3.84 MHz. Specify path delays and gains by using the ITU pedestrian B channel configuration. Set the Rician K-factor to 10 and the maximum Doppler shift to 50 Hz.

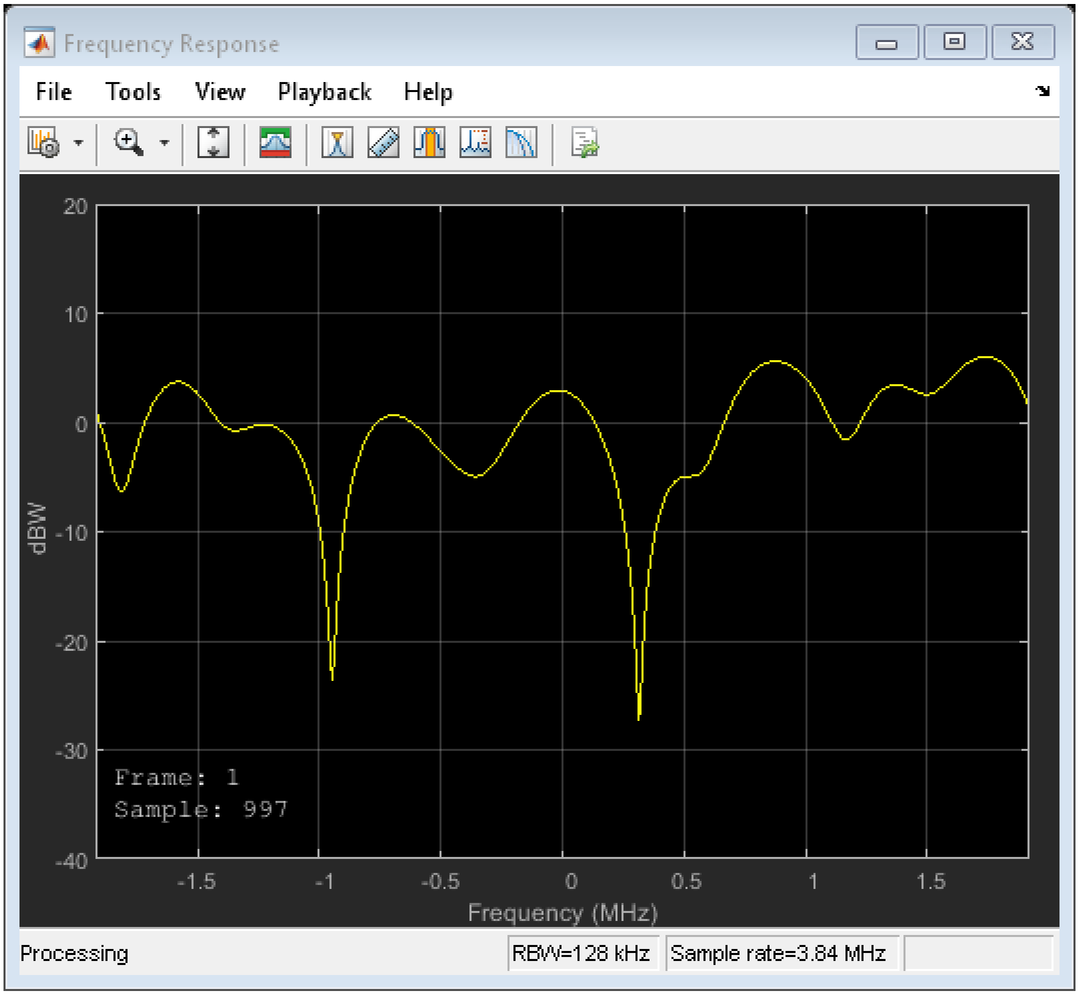
```
fs = 3.84e6; % Hz
pathDelays = [0 200 800 1200 2300 3700]*1e-9; % sec
avgPathGains = [0 -0.9 -4.9 -8 -7.8 -23.9]; % dB
fD = 50; % Hz
```

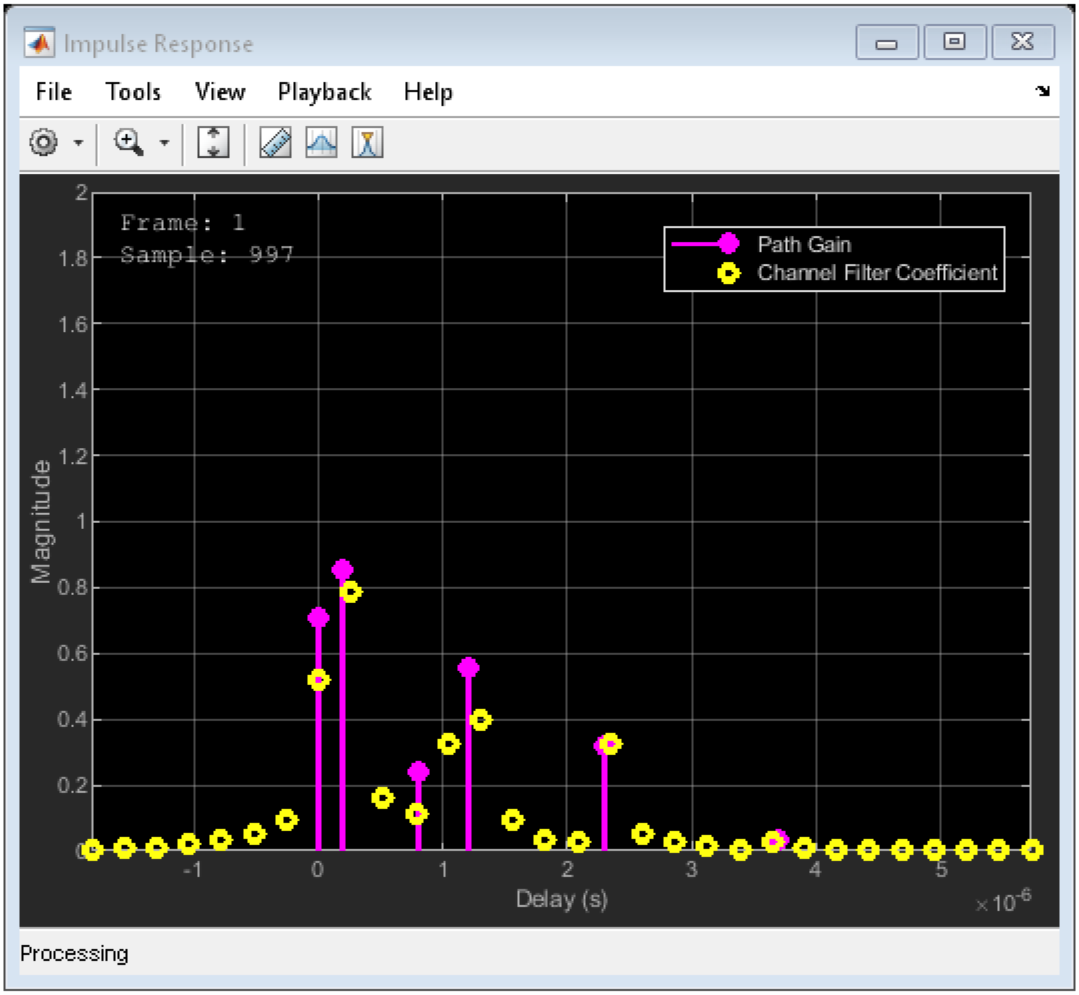
Create a multipath Rician fading channel System object by using the previously defined properties and to visualize the impulse response and frequency response plots.

```
ricianChan = comm.RicianChannel('SampleRate',fs, ...
    'PathDelays',pathDelays, ...
    'AveragePathGains',avgPathGains, ...
    'KFactor',10, ...
    'MaximumDopplerShift',fD, ...
    'Visualization','Impulse and frequency responses');
```

Generate random binary data and pass it through the multipath Rician fading channel. The impulse response plot enables you to identify the individual paths and their corresponding filter coefficients. The frequency response plot shows the frequency-selective nature of the ITU pedestrian B channel.

```
x = randi([0 1],1000,1);
y = ricianChan(x);
```





Model a Rician Channel Using Sum-of-Sinusoids Technique

The example shows how the channel state is maintained in cases in which data is discontinuously transmitted. Create a Rician channel object and pass data through it using the sum-of-sinusoids technique.

Set the channel properties.

```
fs = 1000; % Sample rate (Hz)
pathDelays = [0 2.5e-3]; % Path delays (s)
pathPower = [0 -6]; % Path power (dB)
fD = 5; % Maximum Doppler shift (Hz)
ns = 1000; % Number of samples
nsdel = 100; % Number of samples for delayed paths
```

Define the overall simulation time and three time segments for which data is to be transmitted. In this case, the channel is simulated for 1s with a 1000 Hz sampling rate. One 1000-sample continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at times 0.1 s, 0.5 s, and 0.8 s, respectively.

```
to0 = 0.0;
to1 = 0.1;
to2 = 0.5;
to3 = 0.8;
t0 = (to0:ns-1)/fs;      % Transmission 0
t1 = to1+(0:nsdel-1)/fs; % Transmission 1
t2 = to2+(0:nsdel-1)/fs; % Transmission 2
t3 = to3+(0:nsdel-1)/fs; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],ns,1);
d1 = randi([0 1],nsdel,1);
d2 = randi([0 1],nsdel,1);
d3 = randi([0 1],nsdel,1);
```

Create a frequency-flat multipath Rician fading System object, specifying the sum-of-sinusoids fading technique. So that results can be repeated, specify a seed value. Use the default `InitialTime` property setting so that the fading channel will be simulated from time 0. Enable the path gains output.

```
ricianchan1 = comm.RicianChannel('SampleRate',fs, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the multipath Rician fading channel System object. Set the source for initial time so that the fading channel offset time can be specified as an input argument when using the System object.

```
ricianchan2 = clone(ricianchan1);
ricianchan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first multipath Rician fading channel System object, `ricianchan1`. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = ricianchan1(d0);
```

Pass random data through the second multipath Rician fading channel System object, `ricianchan2`, where the initial time offsets are provided as input arguments.

```
[~,pg1] = ricianchan2(d1,to1);
[~,pg2] = ricianchan2(d2,to2);
[~,pg3] = ricianchan2(d3,to3);
```

Compare the number of samples processed by the two channels by using the `info` object function. The `ricianchan1` object processed 1000 samples, while the `ricianhan2` object only processed 300 samples.

```
G = info(ricianchan1);
H = info(ricianchan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

```
ans = 1×2
```

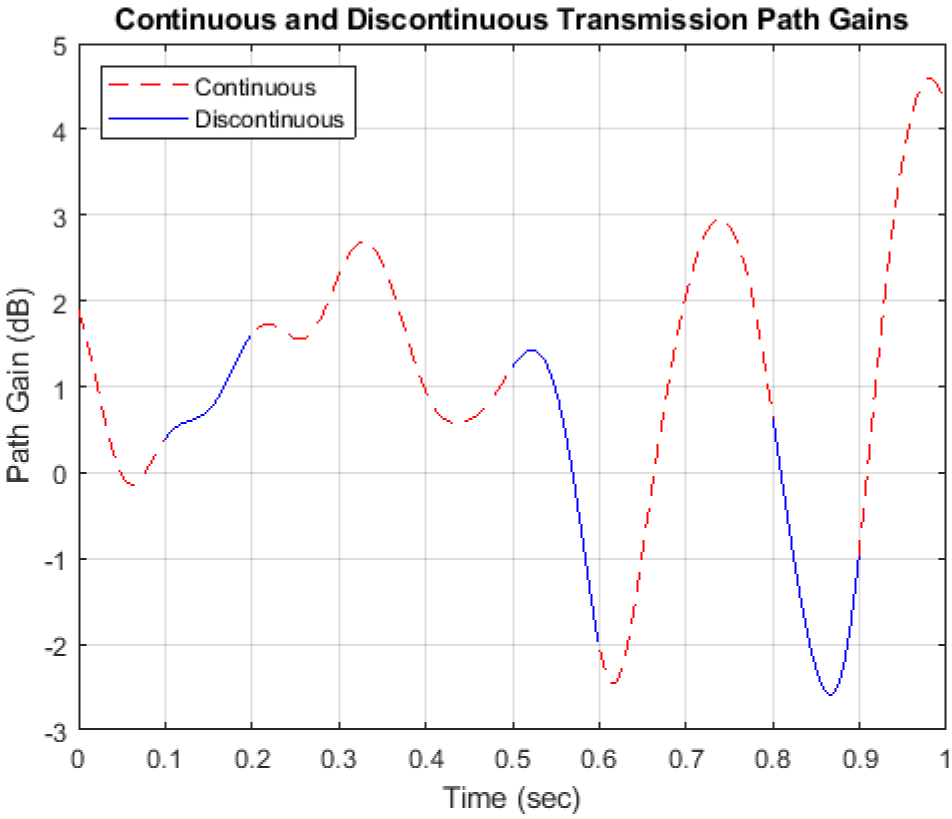
```
        1000         300
```

Convert the path gains into decibels.

```
pathGain0 = 20*log10(abs(pg0));
pathGain1 = 20*log10(abs(pg1));
pathGain2 = 20*log10(abs(pg2));
pathGain3 = 20*log10(abs(pg3));
```

Plot the path gains for the continuous and discontinuous cases. The gains for the three segments match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is suited to the simulation of packetized data, as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
title('Continuous and Discontinuous Transmission Path Gains')
```



Use Info Method to Reproduce Multipath Rican Fading Channel Response

The example show how to use the `ChannelFilterCoefficients` property returned by the `info` object function of the `comm.RicianChannel` System object to reproduce the multipath Rician fading channel output.

Create a multipath Rician fading channel System object, defining two paths and specifying data to pass through the channel.

```
ricianchan = comm.RicianChannel('SampleRate',1000,'PathDelays',[0 1e-3], ...
    'AveragePathGains',[0 -2],'PathGainsOutputPort',true)
```

```
ricianchan = 
  comm.RicianChannel with properties:

                SampleRate: 1000
                PathDelays: [0 1.0000e-03]
          AveragePathGains: [0 -2]
        NormalizePathGains: true
                   KFactor: 3
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 1.0000e-03
           DopplerSpectrum: [1x1 struct]

  Show all properties
```

```
data = randi([0 1],600,1);
```

Pass data through the channel. Assign the `ChannelFilterCoefficients` property value to the variable `coeff`.

```
[chanout1,pg] = ricianchan(data);
chaninfo = info(ricianchan)
```

```
chaninfo = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: [2x2 double]
          NumSamplesProcessed: 600
```

```
coeff = chaninfo.ChannelFilterCoefficients;
```

Calculate the fractional delayed input signal at the path delay locations stored in `coeff`.

```
Np = length(ricianchan.PathDelays);
fracdelaydata = zeros(size(data,1),Np);
for ii = 1:Np
    fracdelaydata(:,ii) = filter(coeff(ii,:),1,data);
end
```

Apply the path gains and sum the results for all paths.

```
chanout2 = sum(pg .* fracdelaydata,2);
```

Compare the output of the multipath Rician fading channel System object to the output reproduced using the path gains and the `ChannelFilterCoefficients` property of the multipath Rician fading channel System object.

```
isequal(chanout1,chanout2)

ans = logical
   1
```

More About

Cutoff Frequency Factor

The cutoff frequency factor, f_c, is determined for different Doppler spectrum types.

- For any Doppler spectrum type other than Gaussian and biGaussian, f_c equals 1.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation` × $\sqrt{2\log 2}$.
- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)` × $\sqrt{2\log 2}$.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - In all other cases, f_c equals 1.

Channel Visualization

The `comm.RicianChannel` System object enables visualization of the channel impulse response, frequency response, and Doppler spectrum.

Note

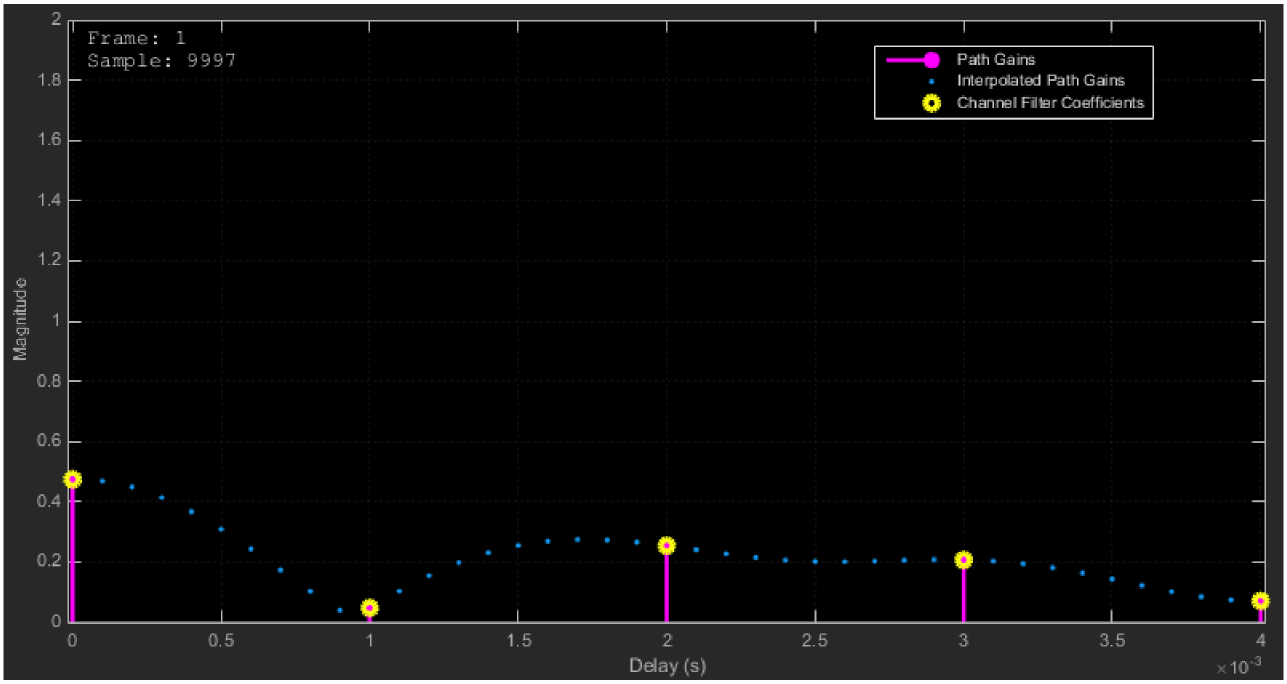
- The displayed and specified path gain locations can differ by as much as 5% of the input sample time.
- The visualization display speed is controlled by the combination of the `SamplesToDisplay` property and the **Playback > Reduce Updates to Improve Performance** scope menu item. Reducing the percentage of samples to display and enabling reduced updates can speed up the rendering of the visualization scope.
- After the visualization scopes are manually closed, calls to the System object are executed at its normal speed.
- Code generation is available only when the `Visualization` property is set to `'Off'`.

Impulse Response

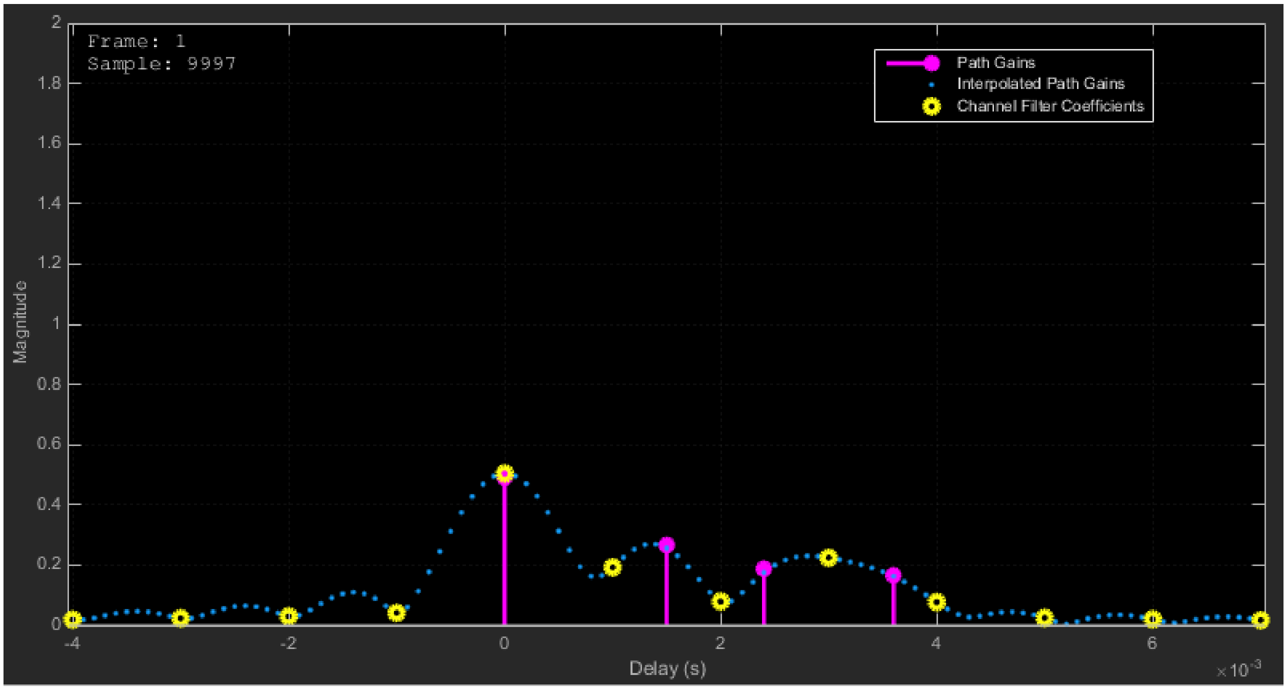
The impulse response scope displays the path gains, channel filter coefficients, and interpolated path gains. The path gains occur at time instances which correspond to the specified `PathDelays` property and might not be aligned with the input sampling time. The channel filter coefficients are used to model the channel. They are interpolated from the actual path gains and are aligned with the input sampling time. In cases in which the path gains are aligned with the sampling time, the path gains overlap the filter coefficients. Sinc interpolation is used to connect the channel filter coefficients and is shown as the interpolated path gains. These points are used solely for display purposes and not used in subsequent channel filtering. For a flat fading channel (one path), the sinc interpolation curve is not displayed. For all impulse response plots, the frame and sample numbers are shown in the upper-left corner of the scope.

The impulse response plot shares the same toolbar and menus as the `dsp.ArrayPlot` System object.

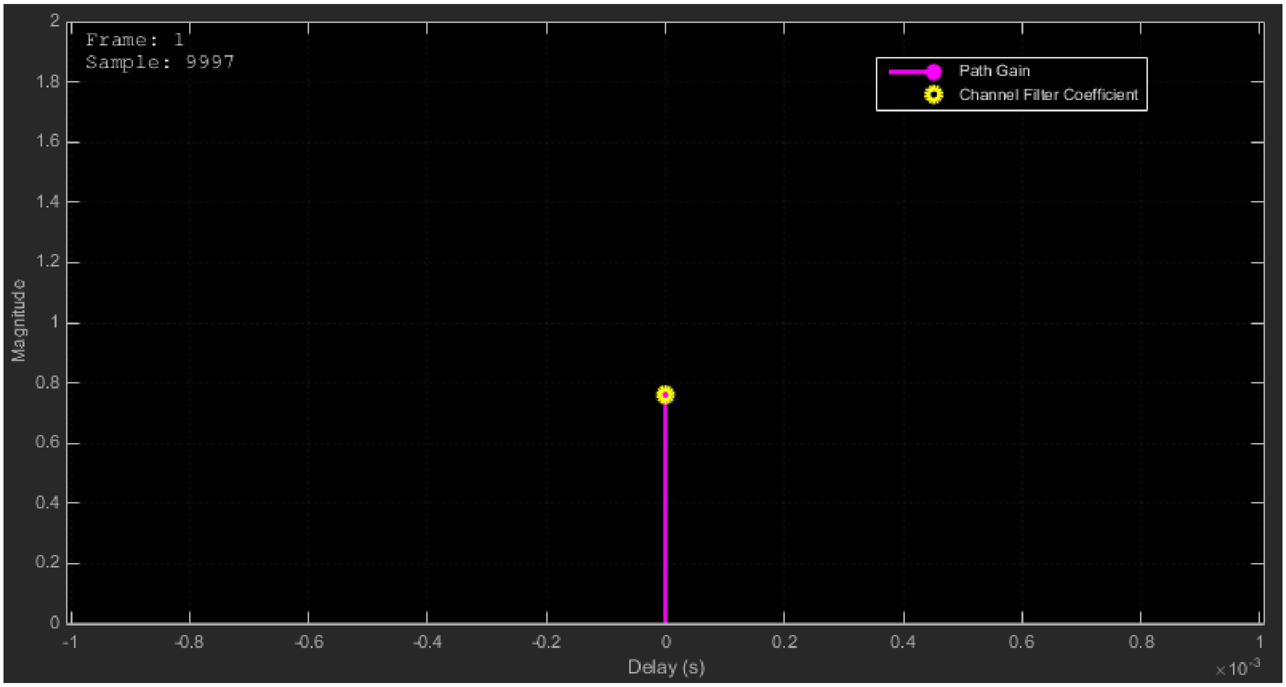
When the specified path gains align with the sample rate, the path gains and the channel filter coefficients overlap. In this figure the impulse response shows the path gains overlap the filter coefficients.



If the specified path gains are not aligned with the sample rate, the path gains and the channel filter coefficients do not overlap. In this figure, the filter coefficients are equally distributed. The path gains do not overlap with the channel filter coefficients because the path gains are not aligned with the sample rate.



This figure shows the impulse response for a frequency-flat channel. In this case, the interpolated path gains are not displayed.



Frequency response

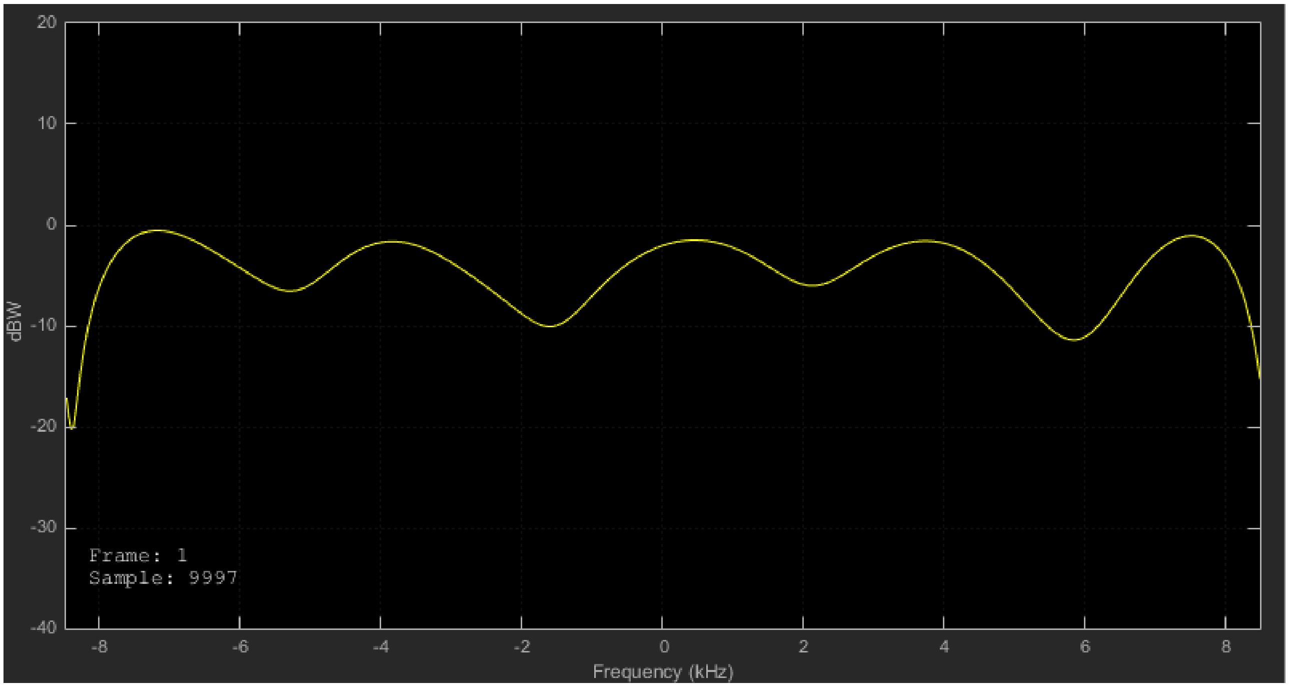
The frequency response scope displays the multipath Rayleigh fading channel spectrum by taking a discrete Fourier transform of the channel filter coefficients. The frequency response plot shares the same toolbar and menus as the `dsp.SpectrumAnalyzer` System object.

The y-axis limits of the plot are computed based on the `NormalizePathGains` and `AveragePathGains` properties of the `comm.RicianChannel` System object.

This table shows other selected default spectrum settings. These settings can be changed from their default values by using the **View > Spectrum Settings** menu.

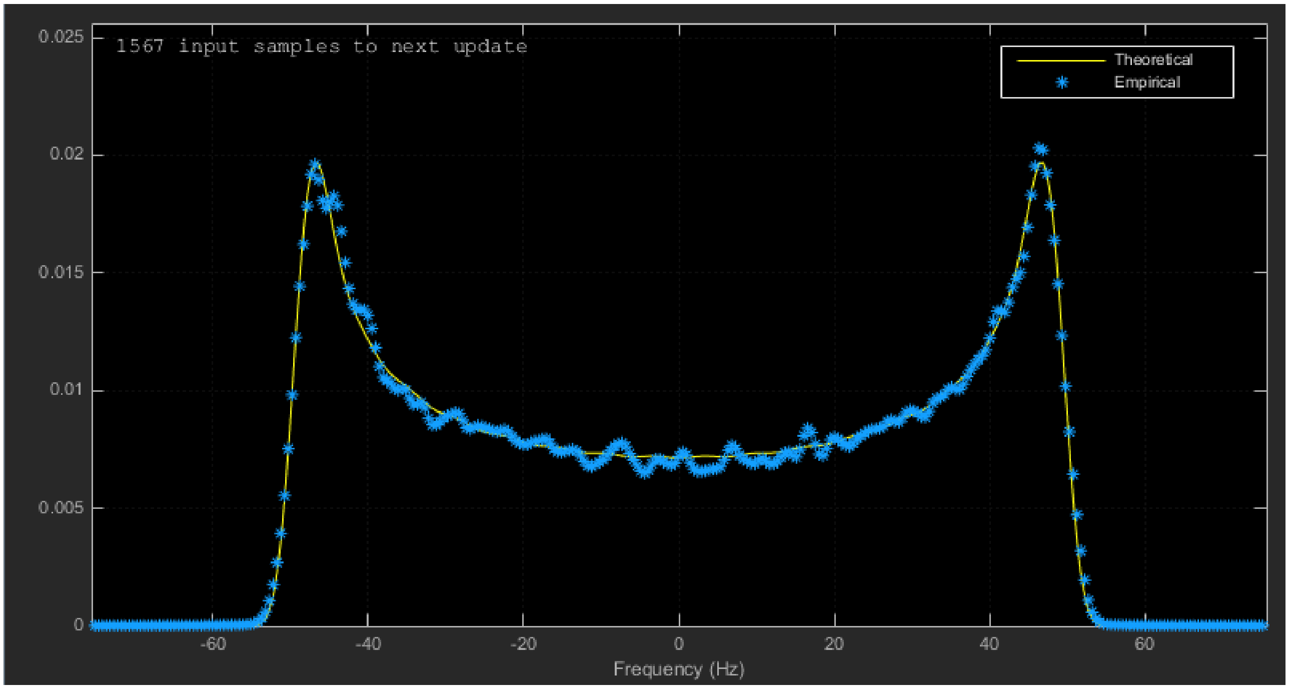
Spectrum Settings	Value
Main options>Type	`Power`
Main options>Window length	Channel filter length
Main options>NFFT	`512`
Window options>Window	`Rectangular`
Trace options>Units	`dBW`

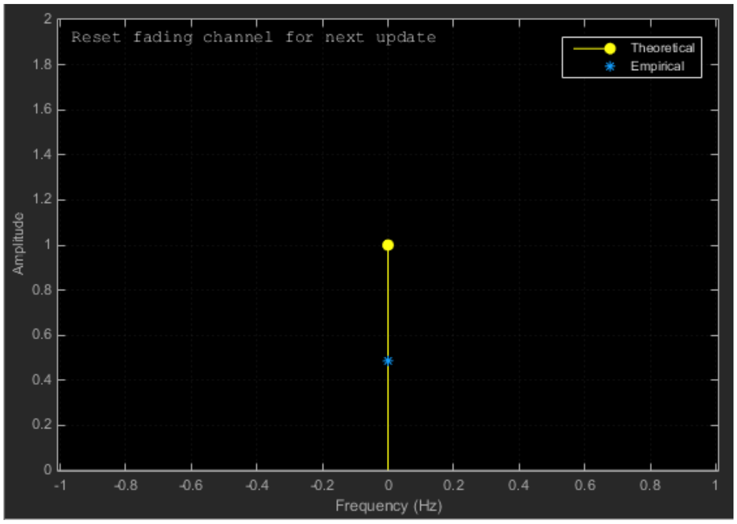
This figure shows the frequency response plot for a frequency selective channel.



Doppler Spectrum

The Doppler spectrum plot displays the theoretical Doppler spectrum and the empirically determined data points. The theoretical data is displayed as a line for the case of nonstatic channels and as a point for static channels. The empirical data is shown as * symbols. Before the empirical plot is updated, the internal buffer must be completely filled with filtered Gaussian samples. The empirical plot is the running mean of the spectrum calculated from each full buffer. For nonstatic channels, the number of input samples needed before the next update is displayed in the upper-left corner of the plot. The number of samples needed is a function of the sample rate and the maximum Doppler shift. For static channels, the text "`Reset fading channel for next update`" is displayed.





References

[1] Oestges, Claude, and Bruno Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*. 1st ed. Boston, MA: Elsevier, 2007.

[2] Correia, Luis M., and European Cooperation in the Field of Scientific and Technical Research (Organization), eds. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*. 1st ed. Amsterdam ; Boston: Elsevier/Academic Press, 2006.

[3] Kermoal, J.P., L. Schumacher, K.I. Pedersen, P.E. Mogensen, and F. Frederiksen. “A Stochastic MIMO Radio Channel Model with Experimental Validation.” *IEEE Journal on Selected Areas in Communications* 20, no. 6 (August 2002): 1211–26. https://doi.org/10.1109/JSAC.2002.801223.

[4] Jeruchim, Michel C., Philip Balaban, and K. Sam Shanmugan. *Simulation of Communication Systems*. Second edition. Boston, MA: Springer US, 2000.

[5] Patzold, M., Cheng-Xiang Wang, and B. Hogstad. “Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms.” *IEEE Transactions on Wireless Communications* 8, no. 6 (June 2009): 3122–31. https://doi.org/10.1109/TWC.2009.080769.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- To generate C code, set the `DopplerSpectrum` property to a single Doppler spectrum structure.
- Code generation is available only when the `Visualization` property is `'Off'`.
- See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.AWGNChannel` | `comm.ChannelFilter` | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.WINNER2Channel`

Functions

`doppler`

Blocks

MIMO Fading Channel | SISO Fading Channel

Introduced in R2013b

comm.RSDecoder

Package: `comm`

Decode data using Reed-Solomon decoder

Description

The `RSDecoder` object recovers a message vector from a Reed-Solomon codeword vector. For proper decoding, the property values for this object should match the property values in the corresponding RS Encoder object.

To decode data using a Reed-Solomon decoding scheme:

1. Define and set up your Reed-Solomon decoder object. See “Construction” on page 3-1319.
2. Call `step` to decode data according to the properties of `comm.RSDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`dec = comm.RSDecoder` creates a block decoder System object, `dec`. This object performs Reed-Solomon (RS) decoding.

`dec = comm.RSDecoder(N,K)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N` and the `MessageLength` property set to `K`.

`dec = comm.RSDecoder(N,K,GP)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, and the `GeneratorPolynomial` property set to `GP`.

`dec = comm.RSDecoder(N,K,GP,S)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, the `GeneratorPolynomial` property set to `GP`, and the `ShortMessageLength` property set to `S`.

`dec = comm.RSDecoder(N,K,GP,S,Name,Value)` creates an RS decoder object, `dec` with the `CodewordLength` property set to `N`, the `MessageLength` property set to `K`, the `GeneratorPolynomial` property set to `GP`, and each specified property `Name` set to the specified `Value`.

`dec = comm.RSDecoder(Name,Value)` creates an RS decoder object, `dec`, with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

Note The input and output signal lengths are listed in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91 on the `comm.BCHDecoder` reference page.

`BitInput`

Assume that input is bits

Specify whether the input comprises bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input data value must be a numeric, column vector of integers. The `step` method outputs an encoded data output vector. The output result is a column vector of integers. Each symbol that forms the input message and output codewords is an integer between 0 and $2^M - 1$. These integers correspond to an element of the finite Galois field `gf`(2^M). M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` on page 3-0 and `PrimitivePolynomial` on page 3-0 properties.

When you set this property to `true`, the input value must be a numeric, column vector of bits. The encoded data output result is a column vector of bits.

`CodewordLength`

Codeword length

Specify the codeword length of the RS code in symbols as a double-precision, positive, integer scalar value. The default is `7`.

For a full-length RS code, the value of this property must be $2^M–1$, where M is an integer such that $3 \leq M \leq 16$.

`MessageLength`

Message length

Specify the message length in symbols as a double-precision positive integer scalar value. The default is `3`.

`ShortMessageLengthSource`

Short message length source

Specify the source of the shortened message as `Auto` or `Property`. When this property is set to `Auto`, the RS code is defined by the `CodewordLength` on page 3-0 , `MessageLength` on page 3-0 , `GeneratorPolynomial` on page 3-0 , and `PrimitivePolynomial` on page 3-0 properties.

When `ShortMessageLengthSource` is set to `Property`, you must specify the `ShortMessageLength` on page 3-0 property, which is used with the other properties to define the RS code. The default is `Auto`.

ShortMessageLength

Shortened message length

Specify the length of the shortened message in symbols as a double-precision positive integer scalar whose value must be less than or equal to `MessageLength` on page 3-0 .

When `ShortMessageLength` < `MessageLength`, the RS code is shortened. The default is `3`.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as `Auto` or `Property`. The default is `Auto`.

When you set this property to `Auto`, the object chooses the generator polynomial automatically. The object calculates the generator polynomial based on the value of the `PrimitivePolynomialSource` on page 3-0 property.

When you set this property to `Auto`, the object automatically chooses the generator polynomial. The object calculates the generator polynomial based on the value of the `PrimitivePolynomial` on page 3-0 property.

When you set this property to Property, you must specify a generator polynomial using the `GeneratorPolynomial` on page 3-0 property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial for the RS code as a double-precision integer row vector or as a Galois field row vector whose entries are in the range from 0 to 2^M–1 and represent a generator polynomial in descending order of powers. The length of the generator polynomial must be `CodewordLength`-`MessageLength`+1. This property applies when you set `GeneratorPolynomialSource` on page 3-0 to `Property`.

The default is the result of `rsgenpoly(7,3,[],[],'double')`, which corresponds to `[1 3 1 2 3]`.

When you use this object to generate code, you must set the generator polynomial to a double-precision integer row vector.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to `true` to perform a generator polynomial check. The default is `true`. This check verifies that the specified generator polynomial is valid.

For larger codes, disabling the check accelerates processing time. As a best practice, perform the check at least once before setting this property to `false`. This property applies when you set `GeneratorPolynomialSource` on page 3-0 to `Property`.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as `Auto` or `Property`. The default is `Auto`.

When you set this property to `Auto`, the object uses a primitive polynomial of degree M = `ceil`(log2(`CodewordLength` on page 3-0 +1)).

When you set`PrimitivePolynomialSource` to `Property`, you must specify a polynomial using `PrimitivePolynomial` on page 3-0 .

`PrimitivePolynomial`

Primitive polynomial

Specify the primitive polynomial that defines the finite field `gf`(2^M) corresponding to the integers that form messages and codewords. The default is the result of `fliplr`(`de2bi`(`primpoly`(*3*))), which is `[1 0 1 1]` or the polynomial $x^3 + x + 1$. Specify this property as a double-precision, binary, row vector that represents a primitive polynomial over `gf`(2) of degree M in descending order of powers. This property applies when you set `PrimitivePolynomialSource` on page 3-0 to `Property`.

`PuncturePatternSource`

Source of puncture pattern

Specify the source of the puncture pattern as `None` or `Property`. The default is `None`.

If you set this property to `None`, the object does not apply puncturing to the code. If you set it to `Property`, the object punctures the code based on a puncture pattern vector specified in `PuncturePattern` on page 3-0 .

`PuncturePattern`

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector of length (`CodewordLength` on page 3-0 -`MessageLength` on page 3-0). The default is `[ones(2,1); zeros(2,1)]`. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword. This property applies when you set `PuncturePatternSource` on page 3-0 to `Property`.

`ErasuresInputPort`

Enable erasures input

Set this property to `true` to specify a vector of erasures as an input to the `step` method. The default is `false`. The erasures input must be a double-precision or logical binary column vector that indicates which symbols of the input codewords to erase.

The length of the erasures vector is explained in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91.

When this property is set to `false`, the object assumes no erasures.

`NumCorrectedErrorsOutputPort`

Enable number of corrected errors output

Set this property to `true` to obtain the number of corrected errors as an output to the `step` method. The default is `true`. A nonnegative value in the *i*-th element of the error output vector, denotes the number of corrected errors in the *i*-th input codeword. A value of `-1` in the *i*-th element of the error output vector indicates that a decoding error occurred for that codeword. A decoding error occurs when an input codeword has more errors than the error correction capability of the RS code.

`OutputDataType`

Data type of output

Specify the output data type as `Same as input`, `double`, or `logical`. The default is `Same as input`. This property applies when you set `BitInput` on page 3-0 to `true`.

Methods

step	Decode data using a Reed-Solomon decoder

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
    enc = comm.RSEncoder;
    mod = comm.DPSKModulator('BitInput',false);
    chan = comm.AWGNChannel(...
                'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
    demod = comm.DPSKDemodulator('BitOutput',false);
    hDdecec = comm.RSDecoder;
    errorRate = comm.ErrorRate('ComputationDelay',3);

    for counter = 1:20
      data = randi([0 7], 30, 1);
      encodedData = step(enc, data);
      modSignal = step(mod, encodedData);
      receivedSignal = step(chan, modSignal);
      demodSignal = step(demod, receivedSignal);
      receivedSymbols = step(hDdecec, demodSignal);
      errorStats = step(errorRate, data, receivedSymbols);
    end

  fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.115578
Number of errors = 69
```

Estimate BER of 8-PSK in AWGN with Reed-Solomon Coding

Transmit Reed-Solomon encoded data using 8-PSK over an AWGN channel. Demodulate and decode the received signal and collect error statistics. Plot the bit error rate estimate.

Define the example parameters.

```
M = 8;  % Modulation order
bps = log2(M); % Bits per symbol
N = 7; % RS codeword length
K = 5; % RS message length
```

Create modulator, demodulator, AWGN channel, and error rate objects.

```
pskModulator = comm.PSKModulator('ModulationOrder',M,'BitInput',true);
pskDemodulator = comm.PSKDemodulator('ModulationOrder',M,'BitOutput',true);
awgnChannel = comm.AWGNChannel('BitsPerSymbol',bps);
errorRate = comm.ErrorRate;
```

Create a (7,5) Reed-Solomon encoder and decoder pair which accepts bit inputs.

```
rsEncoder = comm.RSEncoder('BitInput',true,'CodewordLength',N,'MessageLength',K);
rsDecoder = comm.RSDecoder('BitInput',true,'CodewordLength',N,'MessageLength',K);
```

Set the range of E_b/N_0 values. Initialize the error statistics matrix.

```
ebnoVec = (3:0.5:8)';
errorStats = zeros(length(ebnoVec),3);
```

Estimate the bit error rate for each E_b/N_0 value. The simulation runs until either 100 errors or 10^7 bits is encountered. The main simulation loop processing includes encoding, modulation, demodulation, and decoding.

```
for i = 1:length(ebnoVec)
    awgnChannel.EbNo = ebnoVec(i);
    reset(errorRate)
    while errorStats(i,2) < 100 && errorStats(i,3) < 1e7
        data = randi([0 1],1500,1); % Generate binary data
        encData = rsEncoder(data); % RS encode
        modData = pskModulator(encData); % Modulate
        rxSig = awgnChannel(modData); % Pass signal through AWGN
        rxData = pskDemodulator(rxSig); % Demodulate
        decData = rsDecoder(rxData); % RS decode
        errorStats(i,:) = errorRate(data,decData); % Collect error statistics
    end
end
```

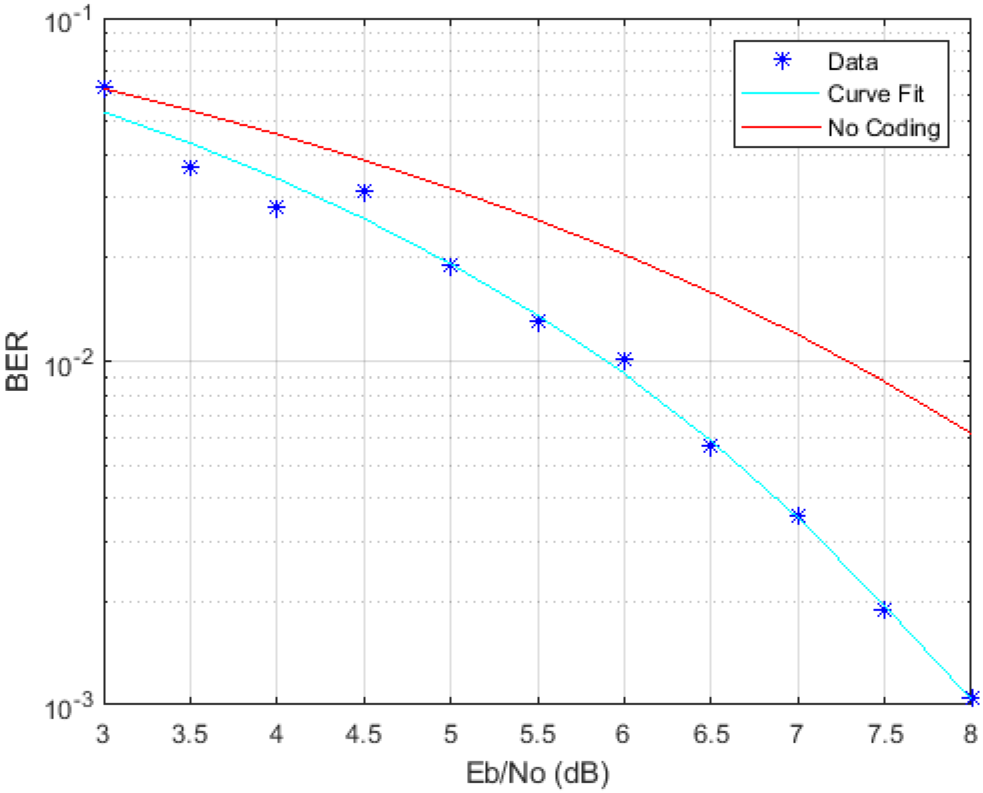
Fit a curve to the BER data using `berfit`. Generate an estimate of 8-PSK performance without coding using the `berawgn` function.

```
berCurveFit = berfit(ebnoVec,errorStats(:,1));
berNoCoding = berawgn(ebnoVec,'psk',8,'nondiff');
```

Plot the BER data, the BER curve fit, and the estimated performance without RS coding.

```
semilogy(ebnoVec,errorStats(:,1),'b*', ...
ebnoVec,berCurveFit,'c-',ebnoVec,berNoCoding,'r')
ylabel('BER')
```

```
xlabel('Eb/No (dB)')
legend('Data','Curve Fit','No Coding')
grid
```



The (7,5) RS code improves the E_b/N_0 required to achieve a 10^{-2} bit error rate by, approximately, 1.4 dB.

Transmit a Shortened RS-encoded, 256-QAM-modulated Symbol Stream

Transmit a shortened RS-encoded, 256-QAM-modulated symbol stream through an AWGN channel. Then demodulate, decode, and count errors.

Set the parameters for the Reed-Solomon code, where N is the codeword length, K is the nominal message length, and S is the shortened message length. Set the modulation order, M, and the number of frames, L.

```
N = 255;
K = 239;
S = 188;
M = 256;
L = 50;
```

Create an AWGN channel System object and an error rate System object.

```
awgnChan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'EbNo',15,'BitsPerSymbol',log2(M));
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Create the Reed-Solomon generator polynomial from the DVB-T standard.

```
gp = rsgenpoly(N,K,[],0);
```

Create a Reed-Solomon encoder and decoder pair using the shortened message length, `S`, and the DVB-T generator polynomial, `gp`.

```
enc = comm.RSEncoder(N,K,gp,S);
dec = comm.RSDecoder(N,K,gp,S);
```

Generate random symbol frames whose length equals one message block. Encode, modulate, apply AWGN, demodulate, decode, and collect statistics.

```
for counter = 1:L
    data = randi([0 1],S,log2(M));
    encodedData = step(enc,bi2de(data));
    modSignal = qammod(encodedData,M,'UnitAveragePower',true);
    rxSignal = awgnChan(modSignal);
    demodSignal = qamdemod(rxSignal,M,'UnitAveragePower',true);
    rxBits = dec(demodSignal);
    dataOut = de2bi(rxBits);
    errorStats = errorRate(data(:),dataOut(:));
end
```

Display the error rate and number of errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 2.01e-02
Number of errors = 1509
```

Reed-Solomon Coding with Erasures

This example shows how to configure the `comm.RSEncoder` and `comm.RSDecoder` System objects to perform Reed-Solomon (RS) block coding with erasures when simulating a communications system. RS decoders can correct both errors and erasures. A receiver that identifies the most unreliable symbols in a given codeword can generate erasures. When a receiver erases a symbol, it replaces that symbol with a zero. The receiver then passes a flag to the decoder, indicating that the symbol is an erasure, not a valid code symbol. In addition, an encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures the exact same way when it decodes a symbol. Puncturing also has the added benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance, given the same demodulator input energy per bit to noise power spectral density ratio (Eb/N0). Note that puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword.

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator and obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncoded Eb/N0 ratio is set EbNoUncoded = 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5e6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses a (63,53) RS code operating with a 64-QAM modulation scheme. This code can correct (63-53)/2 = 5 errors, or it can alternatively correct (63-53) = 10 erasures. For each codeword at the output of the 64-QAM demodulator, the receiver determines the six least reliable symbols using the helperRSCodingGetErasures.m helper function. The indices that point to the location of these unreliable symbols are passed as an input to the RS decoder. The RS decoder treats these symbols as erasures resulting in an error correction capability of (10-6)/2 = 2 errors per codeword.

Create a `comm.RSEncoder` System object and set the `BitInput` property to false to specify that the encoder inputs and outputs are integer symbols.

```
N = 63;  % Codeword length
K = 53;  % Message length
rsEncoder = comm.RSEncoder(N,K,'BitInput',false);
numErasures = 6;
```

Create a `comm.RSDecoder` System object matching the configuration of the `comm.RSEncoder` object.

```
rsDecoder = comm.RSDecoder(N,K,'BitInput',false);
```

Set the `ErasuresInputPort` property to true to specify erasures as an input to the decoder object.

```
rsDecoder.ErasuresInputPort = true;
```

Set the `NumCorrectedErrorsOutputPort` property to true so that the decoder outputs the number of corrected errors. A non negative value in the error output denotes the number of corrected errors in the input codeword. A value of –1 in the error output indicates a decoding error. A decoding error occurs when the input codeword has more errors than the error correction capability of the RS code.

```
rsDecoder.NumCorrectedErrorsOutputPort = true;
```

Run Stream Processing Loop

Simulate the communications system for an uncoded Eb/N0 ratio of 15 dB. The uncoded Eb/N0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded Eb/N0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded Eb/N0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. Set the `EbNo` property of the AWGN channel object to the computed coded Eb/N0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/N);
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K symbols in the [0 N] range.

```
    data = randi([0 N],K,1);
```

Encode the message word. The encoded word, `encData`, is `(N-numPunc)` symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1),reshape(de2bi(demodData,log2(M))',[],1
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1),reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0017
```

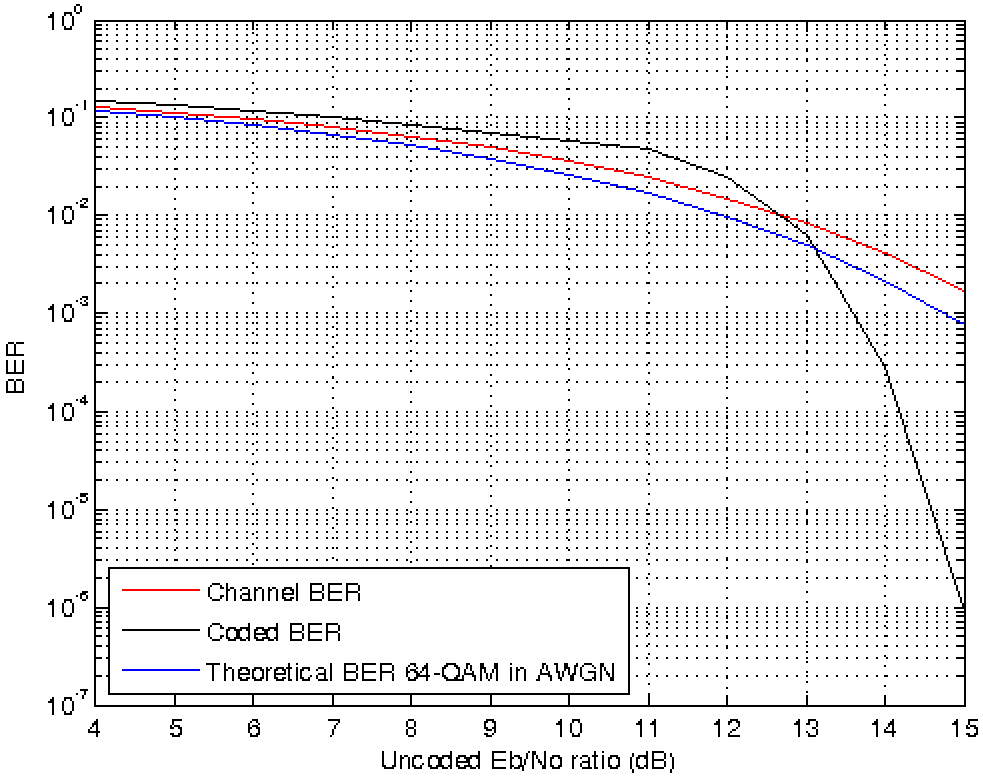
```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 0
```

```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 882
```

You can add a for loop around the processing loop above to run simulations for a set of Eb/N0 values. Simulations were run offline for uncoded Eb/N0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50e6. The results from the simulation are shown. The channel BER is worse than the theoretical 64-QAM BER because Eb/N0 is reduced by the code rate.



Summary

This example utilized several System objects to simulate a 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS decoder to decode symbols with erasures. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Reed-Solomon Coding with Erasures and Punctures

This example shows how to configure the `comm.RSEncoder` and `comm.RSDecoder` System objects to perform Reed-Solomon (RS) block coding with erasures and puncture codes when simulating a communications system. An encoder can generate punctures to remove specific parity symbols from its output. given the puncture pattern, the decoder inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures in exactly the same way when it decodes. Puncturing has the added benefit of making the code rate more flexible, at the expense of some error correction capability.

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures and puncturing by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator. This example obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncoded E_b/N_0 ratio, `EbNoUncoded` is set to 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5×10^6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses the same (63,53) RS code operating with a 64-QAM modulation scheme that is configured for erasures and code puncturing. The RS algortihm decodes receiver-generated erasures and corrects encoder-generated punctures. For each codeword, the sum of the punctures and erasures cannot exceed twice the error-correcting capability of the code.

Create a `comm.RSEncoder` System object and set the `BitInput` property to false to specify that the encoder inputs and outputs are integer symbols.

```
N = 63;  % Codeword length
K = 53;  % Message length
rsEncoder = comm.RSEncoder(N,K,'BitInput',false);
numErasures = 6;
```

Create a `comm.RSDecoder` System object matching the configuration of the `comm.RSEncoder` object. Then set the `ErasuresInputPort` property to true to specify erasures as an input to the decoder object.

```
rsDecoder = comm.RSDecoder(N,K,'BitInput',false);
rsDecoder.ErasuresInputPort = true;
```

To enable code puncturing, set the `PuncturePatternSource` property to `'Property'` and set the `PuncturePattern` property to the desired puncture pattern vector. The same puncture vector must be specified in both the encoder and decoder. This example punctures two symbols from each codeword. Values of `1` in the puncture pattern vector indicate nonpunctured symbols, and values of `0` indicate punctured symbols.

```
numPuncs = 2;
rsEnc.PuncturePatternSource = 'Property';
rsEnc.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];

rsDec.PuncturePatternSource = 'Property';
rsDec.PuncturePattern = rsEnc.PuncturePattern;
```

Run Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded E_b/N_0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded E_b/N_0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. Since the length of the codewords generated by the RS encoder is reduced by the number of punctures specified in the puncture pattern vector, the value of the coded E_b/N_0 ratio needs to be adjusted to account for these punctures. In this example, The number of output symbols is (N - numPuncs) and the uncoded E_b/N_0 ratio relates to the coded E_b/N_0 as shown below. Set the EbNo property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/(N - numPuncs));
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K symbols in the [0 N] range.

```
    data = randi([0 N],K,1);
```

Encode the message word. The encoded word, encData, is (N – numPunc) symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1),reshape(de2bi(demodData,log2(M))',[],1
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1),reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0015
```

```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 0
```

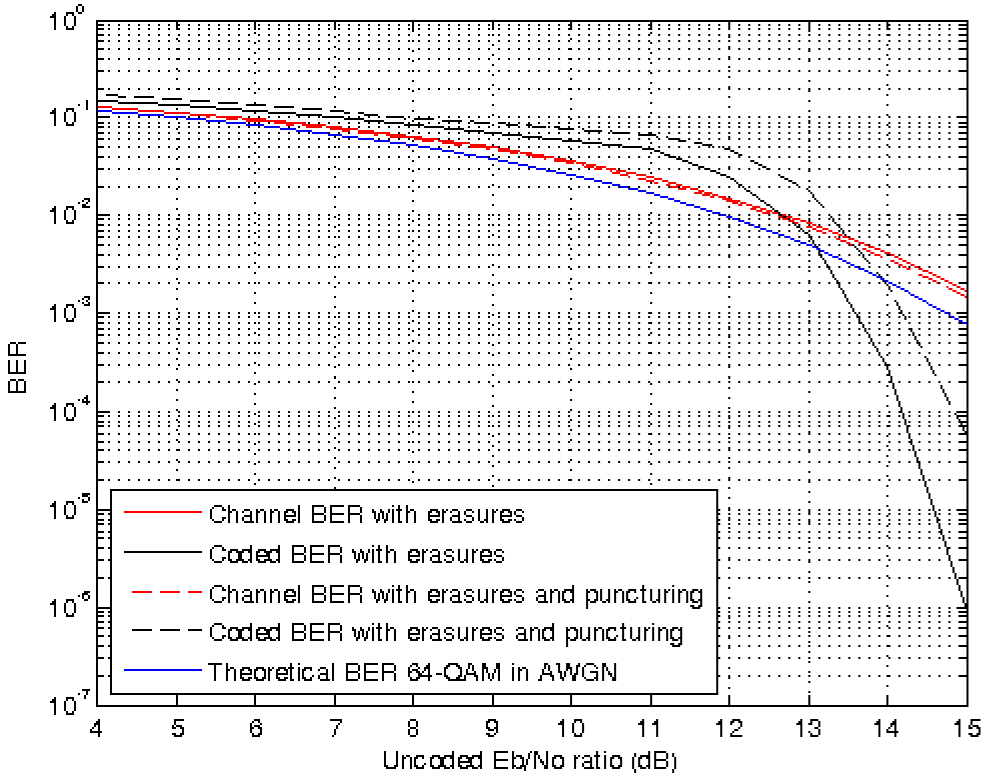
```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 632
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. This figure compares results achieved for

- RS coding with only erasures
- RS coding with erasures and puncturing
- Theoretical BER for 64-QAM

The coded E_b/N_0 is slightly higher than the channel E_b/N_0, so the channel BER is slightly better in the punctured case. On the other hand, the coded BER is worse in the punctured case, because the two punctures reduce the error correcting capability of the code by one, leaving it able to correct only (10 – 6 – 2) / 2 = 1 error per codeword.



Summary

This example utilized functions and System objects to simulate a 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS encoder/decoder System objects to obtain punctured codes. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Reed-Solomon Coding with Erasures, Punctures, and Shortening

This example shows how to configure the `comm.RSEncoder` and `comm.RSDecoder` System objects to perform Reed-Solomon (RS) block coding to shorten the (63,53) code to a (28,18) code. The simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder is presented.

The effects of RS coding with erasures, puncturing, and shortening are analyzed by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator and obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder. Puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword. Puncturing has the benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance for the same demodulator input E_b/N_0.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncodedE_b/N_0 ratio is set `EbNoUncoded` = 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5×10^6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses a (63,53) RS code operating with a 64-QAM modulation scheme. The RS coding operation includes erasures, puncturing, and code shortening. This example shows how to shorten the (63,53) code to a (28,18) code.

To shorten a (63,53) code by 10 symbols to a (53,43) code, you can simply enter 53 and 43 for the `CodewordLength` and `MessageLength` properties, respectively (since $2\lceil\log_2(53 + 1)\rceil - 1 = 63$). However, to shorten it by 35 symbols to a (28,18) code, you must explicitly specify that the symbols belong to the Galois field GF(26). Otherwise, the RS blocks will assume that the code is shortened from a (31,21) code (since $2\lceil\log_2(28 + 1)\rceil - 1 = 31$).

Create a pair of `comm.RSEncoder` and `comm.RSDecoder` System objects so that they perform block coding with a (28,18) code shortened from a (63,53) code that is configured to input and output integer symbols. Configure the decoder to accept an erasure input and two punctures. For each codeword at the output of the 64-QAM demodulator, the receiver determines the six least reliable symbols using the helperRSCodingGetErasures.m helper function. The indices that point to the location of these unreliable symbols are passed as an input to the RS decoder.

```
N = 63;  % Codeword length
K = 53;  % Message length
S = 18;  % Shortened message length
numErasures = 6;
numPuncs = 2;
rsEncoder = comm.RSEncoder(N, K, 'BitInput', false);
rsDecoder = comm.RSDecoder(N, K, 'BitInput', false, 'ErasuresInputPort', true);
rsEncoder.PuncturePatternSource = 'Property';
rsEncoder.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];
rsDecoder.PuncturePatternSource = 'Property';
rsDecoder.PuncturePattern = rsEncoder.PuncturePattern;
```

Set the shortened codeword length and message length values.

```
rsEncoder.ShortMessageLength = S;
rsDecoder.ShortMessageLength = S;
```

Specify the field of $GF(2^6)$ in the RS encoder/decoder System objects, by setting the `PrimitivePolynomialSource` property to `'Property'` and the `PrimitivePolynomial` property to a 6th degree primitive polynomial.

```
primPolyDegree = 6;
rsEncoder.PrimitivePolynomialSource = 'Property';
rsEncoder.PrimitivePolynomial = de2bi(primpoly(primPolyDegree,'nodisplay'),'left-msb');

rsDecoder.PrimitivePolynomialSource = 'Property';
rsDecoder.PrimitivePolynomial = de2bi(primpoly(primPolyDegree,'nodisplay'),'left-msb');
```

Run Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded E_b/N_0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded E_b/N_0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. he value of the coded E_b/N_0 ratio needs to be adjusted to account for shortened and punctured codewords. The number of output symbols is (N - numPuncs - S) and the uncoded E_b/N_0 ratio relates to the coded E_b/N_0 as shown below. Set the EbNo property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(S/(N - numPuncs - K + S));
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K-S symbols in the [0 (2^primPolyDegree)-1] range.

```
    data = randi([0 2^primPolyDegree-1],S,1);
```

Encode the shortened message word. The encoded word encData is (N-numPuncs-S) symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
```

```
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1), ...
        reshape(de2bi(demodData,log2(M))',[],1));
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1), ...
        reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER, and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0036
```

```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 9.6599e-05
```

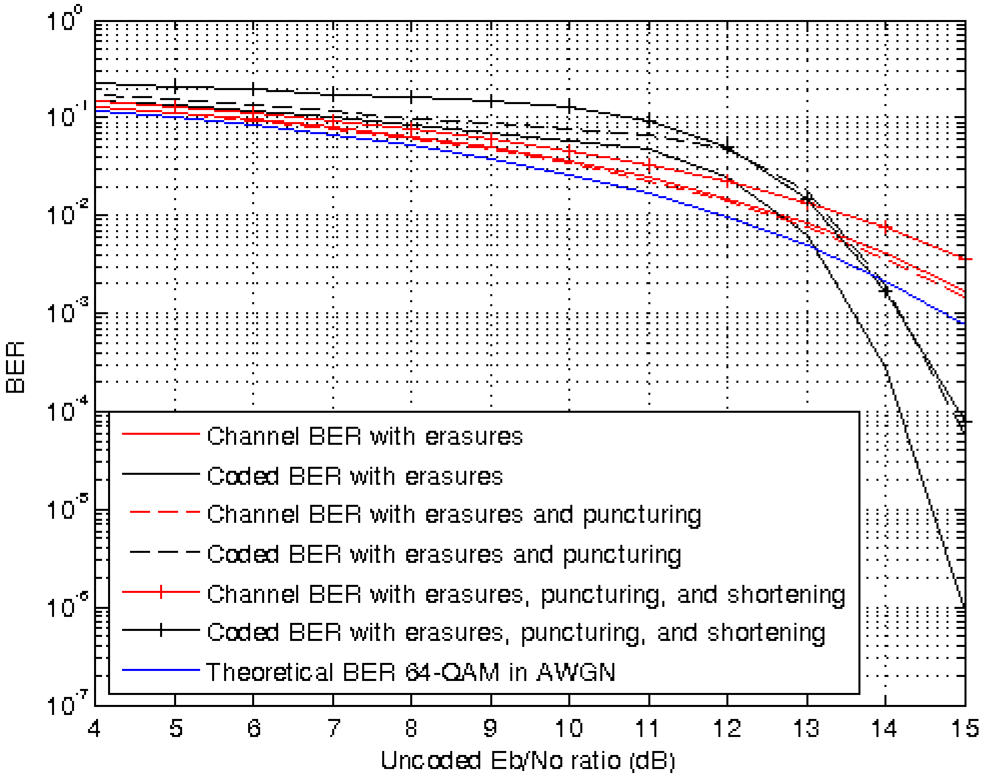
```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 1436
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. This figure compares results achieved for

- RS coding with only erasures
- RS coding with erasures and puncturing
- RS coding with erasures, puncturing, and shotening
- Theoretical BER for 64-QAM

The channel BER with shortening because the coded E_b/N_0 is ddegraded. This degraded coded E_b/N_0 occurs because the code rate of the shortened code is lower than that of the nonshortened code. Shortening also results in degraded coded BER, most noticably at lower E_b/N_0 values.



Summary

This example utilized several System objects to simulate a 64-QAM communications system over an AWGN channel with a shortened RS block code. It showed how to configure the RS decoder to shorten a (63,53) code to a (28,18) code. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHDecoder` | `comm.RSEncoder` | `primpoly` | `rsdec` | `rsgenpoly`

Topics

“Reed-Solomon Codes”

Introduced in R2012a

step

System object: `comm.RSDecoder`
Package: `comm`

Decode data using a Reed-Solomon decoder

Syntax

```
[Y,ERR] = step(H,X)
Y = step(H,X)
Y = step(H,X,ERASURES)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`[Y,ERR] = step(H,X)` decodes the encoded input data, `X`, into the output vector `Y` and returns the number of corrected symbols in output vector `ERR`. The value of the `BitInput` property determines whether `X` is a vector of integers or bits with a numeric, logical, or fixed-point data type. The input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to `true`. A value of `-1` in the *i*-th element of the error output vector indicates that a decoding error occurred for that codeword.

`Y = step(H,X)` decodes the encoded data, `X`, into the output vector `Y`. This syntax applies when you set the `NumCorrectedErrorsOutputPort` property to `false`.

`Y = step(H,X,ERASURES)` uses the binary column input vector, `ERASURES`, to erase the symbols of the input codewords. The elements in `ERASURES` must be of data type double or logical. Values of 1 in the `ERASURES` vector correspond to erased symbols, and values of 0 correspond to non-erased symbols. This syntax applies when you set the ErasuresInputPort property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.RSEncoder

Package: comm

Encode data using Reed-Solomon encoder

Description

The RSEncoder object creates a Reed-Solomon code with message and codeword lengths you specify.

To encode data using a Reed-Solomon encoding scheme:

1. Define and set up your Reed-Solomon encoder object. See "Construction" on page 3-1340.
2. Call step to encode data according to the properties of comm.RSEncoder. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj,x) and y = obj(x) perform equivalent operations.

Construction

enc = comm.RSEncoder creates a block encoder System object, enc. This object performs Reed-Solomon (RS) encoding.

enc = comm.RSEncoder(N,K) creates an RS encoder object, enc, with the CodewordLength property set to N and the MessageLength property set to K.

enc = comm.RSEncoder(N,K,GP) creates an RS encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, and the GeneratorPolynomial property set to GP.

enc = comm.RSEncoder(N,K,GP,S) creates an RS encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, and the ShortMessageLength property set to S.

enc = comm.RSEncoder(N,K,GP,S,Name,Value) creates an RS encoder object, enc, with the CodewordLength property set to N, the MessageLength property set to K, the GeneratorPolynomial property set to GP, the ShortMessageLength property set to S, and each specified property Name set to the specified Value.

enc = comm.RSEncoder(Name,Value) creates an RS encoder object, enc, with each specified property name set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

Note The input and output signal lengths are listed in "Input and Output Signal Lengths in BCH and RS System Objects" on page 3-91 on the comm.BCHDecoder reference page.

BitInput

Assume that input is bits

Specify whether the input comprises bits or integers. The default is `false`.

When you set this property to `false`, the `step` method input data value must be a numeric, column vector of integers. Each symbol that forms the input message and output codewords is an integer between 0 and 2^M–1. These integers correspond to an element of the finite Galois field `gf`(2^M). M is the degree of the primitive polynomial that you specify with the `PrimitivePolynomialSource` on page 3-0 and `PrimitivePolynomial` on page 3-0 properties.

When you set this property to `true`, the input value must be a numeric, column vector of bits. The encoded data output result is a column vector of bits.

CodewordLength

Codeword length

Specify the codeword length of the RS code as a double-precision positive integer scalar value. The default is `7`.

For a full-length RS code, the value of this property must be 2^M–1, where M is an integer such that $3 \leq M \leq 16$.

MessageLength

Message length

Specify the message length as a double-precision positive integer scalar value. The default is `3`.

ShortMessageLengthSource

Short message length source

Specify the source of the shortened message as `Auto` or `Property`. When this property is set to `Auto`, the RS code is defined by the `CodewordLength` on page 3-0 , `MessageLength` on page 3-0 , `GeneratorPolynomial` on page 3-0 , and `PrimitivePolynomial` on page 3-0 properties. When `ShortMessageLengthSource` is set to `Property`, you must specify the `ShortMessageLength` on page 3-0 property, which is used with the other properties to define the RS code. The default is `Auto`.

ShortMessageLength

Shortened message length

Specify the length of the shortened message as a double-precision positive integer scalar whose value must be less than or equal to `MessageLength` on page 3-0 .
When `ShortMessageLength` < `MessageLength`, the RS code is shortened. The default is `3`.

GeneratorPolynomialSource

Source of generator polynomial

Specify the source of the generator polynomial as `Auto` or `Property`. The default is `Auto`.

When you set this property to Auto, the object chooses the generator polynomial automatically. The object calculates the generator polynomial based on the value of the PrimitivePolynomial on page 3-0 property.

When you set GeneratorPolynomialSource to Property, you must specify a generator polynomial using the GeneratorPolynomial on page 3-0 property.

GeneratorPolynomial

Generator polynomial

Specify the generator polynomial for the RS code as a double-precision integer row vector or as a Galois row vector. The Galois row vector entries must be in the range from 0 to 2^M-1. These entries must represent a generator polynomial in descending order of powers. Each coefficient is an element of the Galois field gf(2^M), represented in integer format. The length of the generator polynomial must be CodewordLength on page 3-0 – MessageLength on page 3-0 + 1.

The default is the result of rsgenpoly(7,3,[],[],'double'), which evaluates to a gf(2^3) array with elements [1 3 1 2 3]. This property applies when you set GeneratorPolynomialSource on page 3-0 to Property.

CheckGeneratorPolynomial

Enable generator polynomial checking

Set this property to true to perform a generator polynomial check. The default is true. This check verifies that the specified generator polynomial is valid. For larger codes, disabling the check speeds up processing. As a best practice, perform the check at least once before setting this property to false. This property applies when GeneratorPolynomialSource on page 3-0 is set to Property.

PrimitivePolynomialSource

Source of primitive polynomial

Specify the source of the primitive polynomial as Auto or Property. The default is Auto.

When you set this property to Auto, the object uses a primitive polynomial of degree M = ceil(log2(CodewordLength on page 3-0 +1)).

When you set this property to Property, you must specify a polynomial using the PrimitivePolynomial on page 3-0 property.

PrimitivePolynomial

Primitive polynomial

Specify the primitive polynomial that defines the finite field gf(2^M) corresponding to the integers that form messages and codewords. Specify this property as a double-precision, binary row vector that represents a primitive polynomial over gf(2) of degree M in descending order of powers.

If CodewordLength on page 3-0 is less than 2^M–1, the object uses a shortened RS code. The default is the result of fliplr(de2bi(primpoly(3))), which is [1 0 1 1] or the polynomial $x^3 + x + 1$.

This property applies when you set PrimitivePolynomialSource on page 3-0 to Property.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as `None` or `Property`. The default is `None`.

If you set this property to `None`, the object does not apply puncturing to the code. If you set this property to `Property`, the object punctures the code based on a puncture pattern vector specified in the `PuncturePattern` on page 3-0 property.

PuncturePattern

Puncture pattern vector

Specify the pattern used to puncture the encoded data as a double-precision, binary column vector with a length of (`CodewordLength` on page 3-0 – `MessageLength` on page 3-0). The default is `[ones(2,1); zeros(2,1)]`. Zeros in the puncture pattern vector indicate the position of the parity symbols that are punctured or excluded from each codeword. This property applies when you set the `PuncturePatternSource` on page 3-0 property to `Property`.

OutputDataType

Data type of output

Specify the output data type as `Same as input`, `double`, or `logical`. The default is `Same as input`. This property applies when you set the `BitInput` on page 3-0 property to `true`.

Methods

step	Encode data using a Reed-Solomon encoder

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Transmit an RS-encoded, 8-DPSK-modulated symbol stream

Transmit an RS-encoded, 8-DPSK-modulated symbol stream through an AWGN channel. Then, demodulate, decode, and count errors.

```
enc = comm.RSEncoder;
mod = comm.DPSKModulator('BitInput',false);
chan = comm.AWGNChannel(...
            'NoiseMethod','Signal to noise ratio (SNR)','SNR',10);
demod = comm.DPSKDemodulator('BitOutput',false);
hDdecec = comm.RSDecoder;
errorRate = comm.ErrorRate('ComputationDelay',3);

for counter = 1:20
  data = randi([0 7], 30, 1);
  encodedData = step(enc, data);
```

```
        modSignal = step(mod, encodedData);
        receivedSignal = step(chan, modSignal);
        demodSignal = step(demod, receivedSignal);
        receivedSymbols = step(hDdecec, demodSignal);
        errorStats = step(errorRate, data, receivedSymbols);
    end

  fprintf('Error rate = %f\nNumber of errors = %d\n', ...
      errorStats(1), errorStats(2))
```

```
Error rate = 0.115578
Number of errors = 69
```

Estimate BER of 8-PSK in AWGN with Reed-Solomon Coding

Transmit Reed-Solomon encoded data using 8-PSK over an AWGN channel. Demodulate and decode the received signal and collect error statistics. Plot the bit error rate estimate.

Define the example parameters.

```
M = 8;  % Modulation order
bps = log2(M); % Bits per symbol
N = 7; % RS codeword length
K = 5; % RS message length
```

Create modulator, demodulator, AWGN channel, and error rate objects.

```
pskModulator = comm.PSKModulator('ModulationOrder',M,'BitInput',true);
pskDemodulator = comm.PSKDemodulator('ModulationOrder',M,'BitOutput',true);
awgnChannel = comm.AWGNChannel('BitsPerSymbol',bps);
errorRate = comm.ErrorRate;
```

Create a (7,5) Reed-Solomon encoder and decoder pair which accepts bit inputs.

```
rsEncoder = comm.RSEncoder('BitInput',true,'CodewordLength',N,'MessageLength',K);
rsDecoder = comm.RSDecoder('BitInput',true,'CodewordLength',N,'MessageLength',K);
```

Set the range of E_b/N_0 values. Initialize the error statistics matrix.

```
ebnoVec = (3:0.5:8)';
errorStats = zeros(length(ebnoVec),3);
```

Estimate the bit error rate for each E_b/N_0 value. The simulation runs until either 100 errors or 10^7 bits is encountered. The main simulation loop processing includes encoding, modulation, demodulation, and decoding.

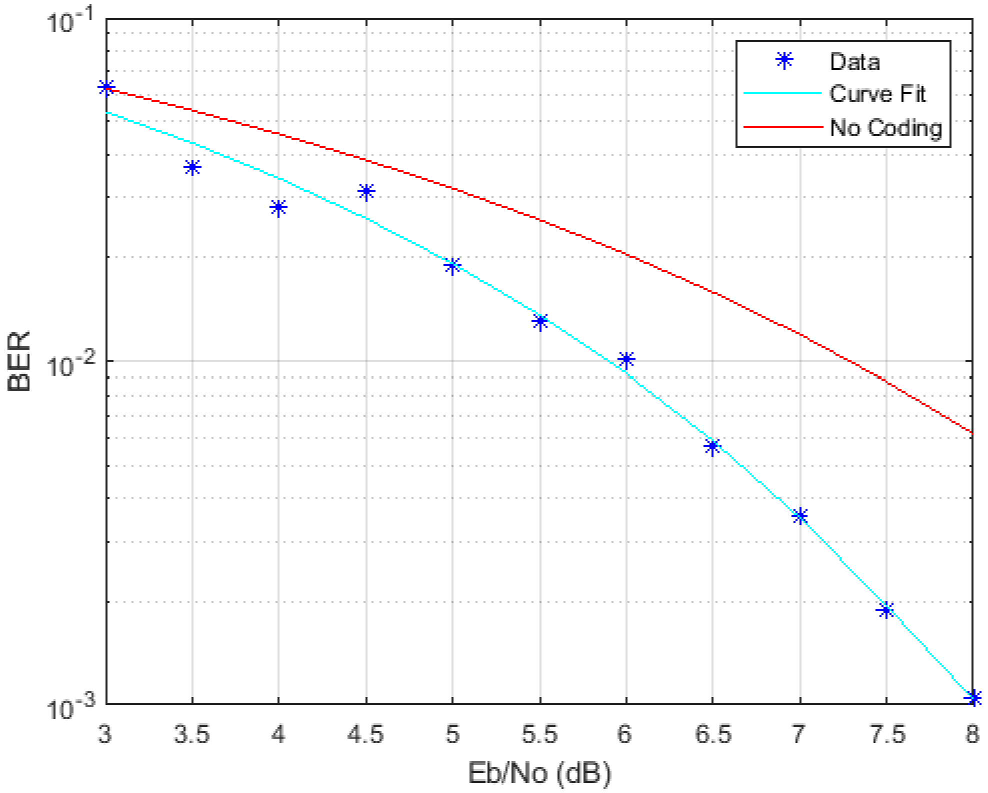
```
for i = 1:length(ebnoVec)
    awgnChannel.EbNo = ebnoVec(i);
    reset(errorRate)
    while errorStats(i,2) < 100 && errorStats(i,3) < 1e7
        data = randi([0 1],1500,1); % Generate binary data
        encData = rsEncoder(data); % RS encode
        modData = pskModulator(encData); % Modulate
        rxSig = awgnChannel(modData); % Pass signal through AWGN
        rxData = pskDemodulator(rxSig); % Demodulate
        decData = rsDecoder(rxData); % RS decode
        errorStats(i,:) = errorRate(data,decData); % Collect error statistics
    end
end
```

Fit a curve to the BER data using `berfit`. Generate an estimate of 8-PSK performance without coding using the `berawgn` function.

```
berCurveFit = berfit(ebnoVec,errorStats(:,1));
berNoCoding = berawgn(ebnoVec,'psk',8,'nondiff');
```

Plot the BER data, the BER curve fit, and the estimated performance without RS coding.

```
semilogy(ebnoVec,errorStats(:,1),'b*', ...
ebnoVec,berCurveFit,'c-',ebnoVec,berNoCoding,'r')
ylabel('BER')
xlabel('Eb/No (dB)')
legend('Data','Curve Fit','No Coding')
grid
```



The (7,5) RS code improves the E_b/N_0 required to achieve a 10^{-2} bit error rate by, approximately, 1.4 dB.

Transmit a Shortened RS-encoded, 256-QAM-modulated Symbol Stream

Transmit a shortened RS-encoded, 256-QAM-modulated symbol stream through an AWGN channel. Then demodulate, decode, and count errors.

Set the parameters for the Reed-Solomon code, where N is the codeword length, K is the nominal message length, and S is the shortened message length. Set the modulation order, M, and the number of frames, L.

```
N = 255;
K = 239;
S = 188;
M = 256;
L = 50;
```

Create an AWGN channel System object and an error rate System object.

```
awgnChan = comm.AWGNChannel('NoiseMethod','Signal to noise ratio (Eb/No)', ...
    'EbNo',15,'BitsPerSymbol',log2(M));
errorRate = comm.ErrorRate('ComputationDelay',3);
```

Create the Reed-Solomon generator polynomial from the DVB-T standard.

```
gp = rsgenpoly(N,K,[],0);
```

Create a Reed-Solomon encoder and decoder pair using the shortened message length, S, and the DVB-T generator polynomial, gp.

```
enc = comm.RSEncoder(N,K,gp,S);
dec = comm.RSDecoder(N,K,gp,S);
```

Generate random symbol frames whose length equals one message block. Encode, modulate, apply AWGN, demodulate, decode, and collect statistics.

```
for counter = 1:L
    data = randi([0 1],S,log2(M));
    encodedData = step(enc,bi2de(data));
    modSignal = qammod(encodedData,M,'UnitAveragePower',true);
    rxSignal = awgnChan(modSignal);
    demodSignal = qamdemod(rxSignal,M,'UnitAveragePower',true);
    rxBits = dec(demodSignal);
    dataOut = de2bi(rxBits);
    errorStats = errorRate(data(:),dataOut(:));
end
```

Display the error rate and number of errors.

```
fprintf('Error rate = %5.2e\nNumber of errors = %d\n', ...
    errorStats(1), errorStats(2))
```

```
Error rate = 2.01e-02
Number of errors = 1509
```

Reed-Solomon Coding with Erasures

This example shows how to configure the comm.RSEncoder and comm.RSDecoder System objects to perform Reed-Solomon (RS) block coding with erasures when simulating a communications system.

RS decoders can correct both errors and erasures. A receiver that identifies the most unreliable symbols in a given codeword can generate erasures. When a receiver erases a symbol, it replaces that symbol with a zero. The receiver then passes a flag to the decoder, indicating that the symbol is an erasure, not a valid code symbol. In addition, an encoder can generate punctures for which specific parity symbols are always removed from its output. The decoder, which knows the puncture pattern, inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures the exact same way when it decodes a symbol. Puncturing also has the added benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance, given the same demodulator input energy per bit to noise power spectral density ratio (Eb/N0). Note that puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword.

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator and obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncoded Eb/N0 ratio is set EbNoUncoded = 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5e6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses a (63,53) RS code operating with a 64-QAM modulation scheme. This code can correct (63-53)/2 = 5 errors, or it can alternatively correct (63-53) = 10 erasures. For each codeword at the output of the 64-QAM demodulator, the receiver determines the six least reliable symbols using the helperRSCodingGetErasures.m helper function. The indices that point to the location of these unreliable symbols are passed as an input to the RS decoder. The RS decoder treats these symbols as erasures resulting in an error correction capability of (10-6)/2 = 2 errors per codeword.

Create a `comm.RSEncoder` System object and set the `BitInput` property to false to specify that the encoder inputs and outputs are integer symbols.

```
N = 63;  % Codeword length
K = 53;  % Message length
rsEncoder = comm.RSEncoder(N,K,'BitInput',false);
numErasures = 6;
```

Create a `comm.RSDecoder` System object matching the configuration of the `comm.RSEncoder` object.

```
rsDecoder = comm.RSDecoder(N,K,'BitInput',false);
```

Set the `ErasuresInputPort` property to true to specify erasures as an input to the decoder object.

```
rsDecoder.ErasuresInputPort = true;
```

Set the `NumCorrectedErrorsOutputPort` property to true so that the decoder outputs the number of corrected errors. A non negative value in the error output denotes the number of corrected errors in the input codeword. A value of –1 in the error output indicates a decoding error. A decoding error occurs when the input codeword has more errors than the error correction capability of the RS code.

```
rsDecoder.NumCorrectedErrorsOutputPort = true;
```

Run Stream Processing Loop

Simulate the communications system for an uncoded Eb/N0 ratio of 15 dB. The uncoded Eb/N0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded Eb/N0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded Eb/N0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. Set the `EbNo` property of the AWGN channel object to the computed coded Eb/N0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/N);
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K symbols in the [0 N] range.

```
    data = randi([0 N],K,1);
```

Encode the message word. The encoded word, `encData`, is `(N-numPunc)` symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1),reshape(de2bi(demodData,log2(M))',[],1
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1),reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0017
```

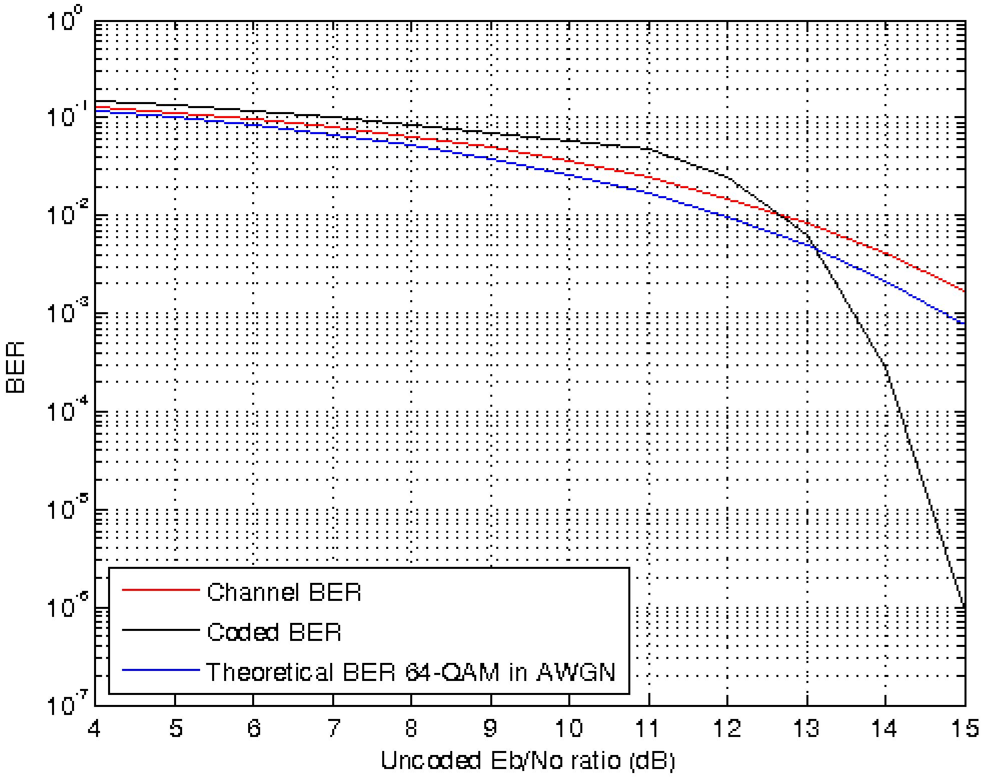
```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 0
```

```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 882
```

You can add a for loop around the processing loop above to run simulations for a set of Eb/N0 values. Simulations were run offline for uncoded Eb/N0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50e6. The results from the simulation are shown. The channel BER is worse than the theoretical 64-QAM BER because Eb/N0 is reduced by the code rate.



Summary

This example utilized several System objects to simulate a 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS decoder to decode symbols with erasures. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Reed-Solomon Coding with Erasures and Punctures

This example shows how to configure the `comm.RSEncoder` and `comm.RSDecoder` System objects to perform Reed-Solomon (RS) block coding with erasures and puncture codes when simulating a communications system. An encoder can generate punctures to remove specific parity symbols from its output. given the puncture pattern, the decoder inserts zeros in the puncture positions and treats those symbols as erasures. The decoder treats encoder-generated punctures and receiver-generated erasures in exactly the same way when it decodes. Puncturing has the added benefit of making the code rate more flexible, at the expense of some error correction capability.

This example shows the simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder. It includes analysis of RS coding with erasures and puncturing by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator. This example obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncoded E_b/N_0 ratio, `EbNoUncoded` is set to 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5×10^6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses the same (63,53) RS code operating with a 64-QAM modulation scheme that is configured for erasures and code puncturing. The RS algortihm decodes receiver-generated erasures and corrects encoder-generated punctures. For each codeword, the sum of the punctures and erasures cannot exceed twice the error-correcting capability of the code.

Create a `comm.RSEncoder` System object and set the `BitInput` property to false to specify that the encoder inputs and outputs are integer symbols.

```
N = 63;  % Codeword length
K = 53;  % Message length
rsEncoder = comm.RSEncoder(N,K,'BitInput',false);
numErasures = 6;
```

Create a `comm.RSDecoder` System object matching the configuration of the `comm.RSEncoder` object. Then set the `ErasuresInputPort` property to true to specify erasures as an input to the decoder object.

```
rsDecoder = comm.RSDecoder(N,K,'BitInput',false);
rsDecoder.ErasuresInputPort = true;
```

To enable code puncturing, set the `PuncturePatternSource` property to `'Property'` and set the `PuncturePattern` property to the desired puncture pattern vector. The same puncture vector must be specified in both the encoder and decoder. This example punctures two symbols from each codeword. Values of `1` in the puncture pattern vector indicate nonpunctured symbols, and values of `0` indicate punctured symbols.

```
numPuncs = 2;
rsEnc.PuncturePatternSource = 'Property';
rsEnc.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];

rsDec.PuncturePatternSource = 'Property';
rsDec.PuncturePattern = rsEnc.PuncturePattern;
```

Run Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded E_b/N_0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded E_b/N_0 ratio. If you input `K` symbols to the encoder and obtain `N` output symbols, then the energy relation is given by the `K/N` rate. Since the length of the codewords generated by the RS encoder is reduced by the number of punctures specified in the puncture pattern vector, the value of the coded E_b/N_0 ratio needs to be adjusted to account for these punctures. In this example, The number of output symbols is (`N - numPuncs`) and the uncoded E_b/N_0 ratio relates to the coded E_b/N_0 as shown below. Set the `EbNo` property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(K/(N - numPuncs));
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K symbols in the [0 `N`] range.

```
    data = randi([0 N],K,1);
```

Encode the message word. The encoded word, `encData`, is (`N – numPunc`) symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1),reshape(de2bi(demodData,log2(M))',[],1
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1),reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0015
```

```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 0
```

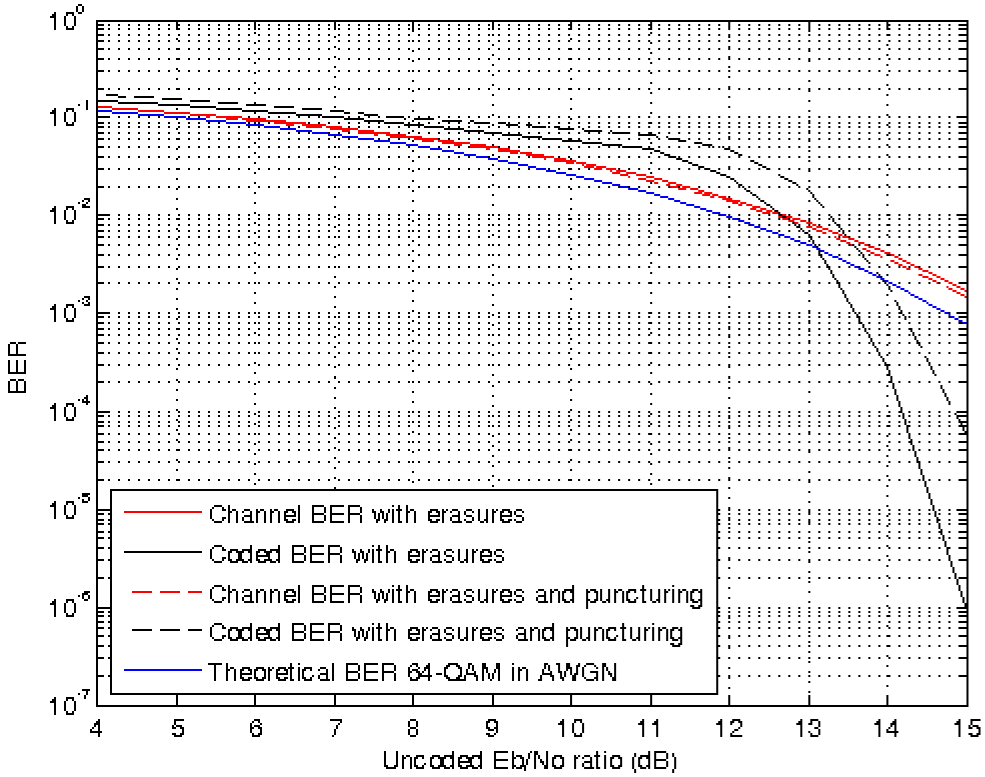
```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 632
```

You can add a for loop around the processing loop above to run simulations for a set of E_b/N_0 values. Simulations were run offline for uncoded E_b/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. This figure compares results achieved for

- RS coding with only erasures
- RS coding with erasures and puncturing
- Theoretical BER for 64-QAM

The coded E_b/N_0 is slightly higher than the channel E_b/N_0, so the channel BER is slightly better in the punctured case. On the other hand, the coded BER is worse in the punctured case, because the two punctures reduce the error correcting capability of the code by one, leaving it able to correct only (10 – 6 – 2) / 2 = 1 error per codeword.



Summary

This example utilized functions and System objects to simulate a 64-QAM communications system over an AWGN channel with RS block coding. It showed how to configure the RS encoder/decoder System objects to obtain punctured codes. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Reed-Solomon Coding with Erasures, Punctures, and Shortening

This example shows how to configure the `comm.RSEncoder` and `comm.RSDecoder` System objects to perform Reed-Solomon (RS) block coding to shorten the (63,53) code to a (28,18) code. The simulation of a communication system consisting of a random source, an RS encoder, a 64-QAM modulator, an AWGN channel, a 64-QAM demodulator, and an RS decoder is presented.

The effects of RS coding with erasures, puncturing, and shortening are analyzed by comparing the channel bit error rate (BER) performance versus the coded BER performance. This example obtains Channel BER by comparing inputs for the QAM modulator to outputs from the QAM demodulator and obtains Coded BER by comparing inputs for the RS encoder to outputs from the RS decoder. Puncturing is the removal of parity symbols from a codeword, and shortening is the removal of message symbols from a codeword. Puncturing has the benefit of making the code rate more flexible, at the expense of some error correction capability. Shortened codes achieve the same code rate flexibility without degrading the error correction performance for the same demodulator input E_b/N_0.

Initialization

The helperRSCodingConfig.m helper function initializes simulation parameters, and configures the `comm.AWGNChannel` and `comm.ErrorRate` System objects used to simulate the communications system. The uncodedE_b/N_0 ratio is set `EbNoUncoded` = 15 dB. Criteria to stop the simulation stop are defined to stop the simulation if 500 errors occur or a maximum 5×10^6 bits are transmitted.

```
helperRSCodingConfig;
```

Configure RS Encoder/Decoder

This example uses a (63,53) RS code operating with a 64-QAM modulation scheme. The RS coding operation includes erasures, puncturing, and code shortening. This example shows how to shorten the (63,53) code to a (28,18) code.

To shorten a (63,53) code by 10 symbols to a (53,43) code, you can simply enter 53 and 43 for the `CodewordLength` and `MessageLength` properties, respectively (since $2\lceil \log_2(53 + 1)\rceil - 1 = 63$). However, to shorten it by 35 symbols to a (28,18) code, you must explicitly specify that the symbols belong to the Galois field GF(26). Otherwise, the RS blocks will assume that the code is shortened from a (31,21) code (since $2\lceil \log_2(28 + 1)\rceil - 1 = 31$).

Create a pair of `comm.RSEncoder` and `comm.RSDecoder` System objects so that they perform block coding with a (28,18) code shortened from a (63,53) code that is configured to input and output integer symbols. Configure the decoder to accept an erasure input and two punctures. For each codeword at the output of the 64-QAM demodulator, the receiver determines the six least reliable symbols using the helperRSCodingGetErasures.m helper function. The indices that point to the location of these unreliable symbols are passed as an input to the RS decoder.

```
N = 63;  % Codeword length
K = 53;  % Message length
S = 18;  % Shortened message length
numErasures = 6;
numPuncs = 2;
rsEncoder = comm.RSEncoder(N, K, 'BitInput', false);
rsDecoder = comm.RSDecoder(N, K, 'BitInput', false, 'ErasuresInputPort', true);
rsEncoder.PuncturePatternSource = 'Property';
rsEncoder.PuncturePattern = [ones(N-K-numPuncs,1); zeros(numPuncs,1)];
rsDecoder.PuncturePatternSource = 'Property';
rsDecoder.PuncturePattern = rsEncoder.PuncturePattern;
```

Set the shortened codeword length and message length values.

```
rsEncoder.ShortMessageLength = S;
rsDecoder.ShortMessageLength = S;
```

Specify the field of GF(2^6) in the RS encoder/decoder System objects, by setting the `PrimitivePolynomialSource` property to `'Property'` and the `PrimitivePolynomial` property to a 6th degree primitive polynomial.

```
primPolyDegree = 6;
rsEncoder.PrimitivePolynomialSource = 'Property';
rsEncoder.PrimitivePolynomial = de2bi(primpoly(primPolyDegree,'nodisplay'),'left-msb');

rsDecoder.PrimitivePolynomialSource = 'Property';
rsDecoder.PrimitivePolynomial = de2bi(primpoly(primPolyDegree,'nodisplay'),'left-msb');
```

Run Stream Processing Loop

Simulate the communications system for an uncoded E_b/N_0 ratio of 15 dB. The uncoded E_b/N_0 is the ratio that would be measured at the input of the channel if there was no coding in the system.

The signal going into the AWGN channel is the encoded signal, so you must convert the uncoded E_b/N_0 values so that they correspond to the energy ratio at the encoder output. This ratio is the coded E_b/N_0 ratio. If you input K symbols to the encoder and obtain N output symbols, then the energy relation is given by the K/N rate. he value of the coded E_b/N_0 ratio needs to be adjusted to account for shortened and punctured codewords. The number of output symbols is (N - numPuncs - S) and the uncoded E_b/N_0 ratio relates to the coded E_b/N_0 as shown below. Set the EbNo property of the AWGN channel object to the computed coded E_b/N_0 value.

```
EbNoCoded = EbNoUncoded + 10*log10(S/(N - numPuncs - K + S));
channel.EbNo = EbNoCoded;
```

Loop until the simulation reaches the target number of errors or the maximum number of transmissions.

```
chanErrorStats = zeros(3,1);
codedErrorStats = zeros(3,1);
correctedErrors = 0;
while (codedErrorStats(2) < targetErrors) && ...
        (codedErrorStats(3) < maxNumTransmissions)
```

The data symbols transmit one message word at a time. Each message word has K-S symbols in the [0 (2^primPolyDegree)-1] range.

```
    data = randi([0 2^primPolyDegree-1],S,1);
```

Encode the shortened message word. The encoded word encData is (N-numPuncs-S) symbols long.

```
    encData = rsEncoder(data);
```

Modulate encoded data and add noise. then demodulate channel output.

```
    modData = qammod(encData,M);
    chanOutput = channel(modData);
    demodData = qamdemod(chanOutput,M);
```

Use the helperRSCodingGetErasures.m helper function to find the 6 least reliable symbols and generate an erasures vector. The length of the erasures vector must be equal to the number of symbols in the demodulated codeword. A one in the ith element of the vector erases the ith symbol in the codeword. Zeros in the vector indicate no erasures.

```
    erasuresVec = helperRSCodingGetErasures(chanOutput,numErasures);
```

Decode the data. Accumulate the number of corrected errors using the cumulative sum object.

```
    [estData,errs] = rsDecoder(demodData,erasuresVec);
    if (errs >= 0)
```

```
        correctedErrors = cumulativeSum(errs);
    end
```

When computing the channel and coded BERs, convert integers to bits.

```
    chanErrorStats(:,1) = ...
        chanBERCalc(reshape(de2bi(encData,log2(M))',[],1), ...
        reshape(de2bi(demodData,log2(M))',[],1));
    codedErrorStats(:,1) = ...
        codedBERCalc(reshape(de2bi(data,log2(M))',[],1), ...
        reshape(de2bi(estData,log2(M))',[],1));
end
```

The error rate measurement objects, `chanBERCalc` and `codedBERCalc`, output 3-by-1 vectors containing BER measurement updates, the number of errors, and the total number of bit transmissions. Display the channel BER, the coded BER, and the total number of errors corrected by the RS decoder.

```
chanBitErrorRate = chanErrorStats(1)
```

```
chanBitErrorRate = 0.0036
```

```
codedBitErrorRate = codedErrorStats(1)
```

```
codedBitErrorRate = 9.6599e-05
```

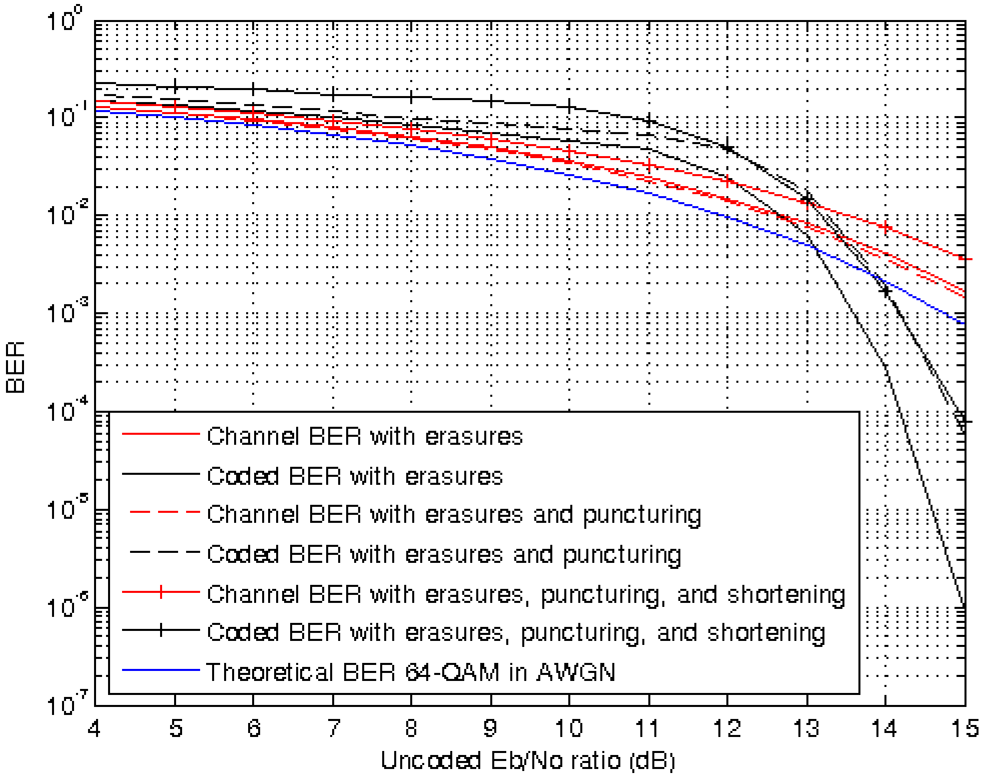
```
totalCorrectedErrors = correctedErrors
```

```
totalCorrectedErrors = 1436
```

You can add a for loop around the processing loop above to run simulations for a set of E_{b}/N_0 values. Simulations were run offline for uncoded E_{b}/N_0 values in 4:15 dB, target number of errors equal to 5000, and maximum number of transmissions equal to 50×10^6. This figure compares results achieved for

- RS coding with only erasures
- RS coding with erasures and puncturing
- RS coding with erasures, puncturing, and shotening
- Theoretical BER for 64-QAM

The channel BER with shortening because the coded E_{b}/N_0 is ddegraded. This degraded coded E_{b}/N_0 occurs because the code rate of the shortened code is lower than that of the nonshortened code. Shortening also results in degraded coded BER, most noticably at lower E_{b}/N_0 values.



Summary

This example utilized several System objects to simulate a 64-QAM communications system over an AWGN channel with a shortened RS block code. It showed how to configure the RS decoder to shorten a (63,53) code to a (28,18) code. System performance was measured using channel and coded BER curves obtained using error rate measurement System objects.

Helper functions used in this example:

- helperRSCodingConfig.m
- helperRSCodingGetErasures.m

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.BCHEncoder` | `comm.RSDecoder` | `primpoly` | `rsenc` | `rsgenpoly`

Topics

“Reed-Solomon Codes”

Introduced in R2012a

step

System object: `comm.RSEncoder`
Package: `comm`

Encode data using a Reed-Solomon encoder

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` encodes the numeric column input data vector, `X`, and returns the encoded data, `Y`. The value of the `BitInput` property determines whether `X` is a vector of integers or bits with a numeric, logical, or fixed-point data type. The input and output length of the step function equal the values listed in the table in “Input and Output Signal Lengths in BCH and RS System Objects” on page 3-91.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.Scrambler

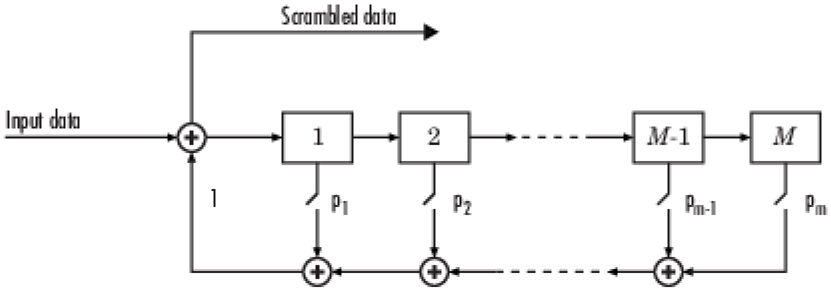
Package: `comm`

Scramble input signal

Description

The comm.Scrambler object scrambles a scalar or column vector input signal.

This schematic shows the scrambler operation. The adders operate modulo *N*, where *N* is the value specified by the Calculation base property.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Polynomial property, you specify the on or off state for each switch in the scrambler.

To scramble an input signal:

1. Create the `comm.Scrambler` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
scrambler = comm.Scrambler
scrambler = comm.Scrambler(base,poly,cond)
scrambler = comm.Scrambler( ___ ,Name,Value)
```

Description

`scrambler = comm.Scrambler` creates a scrambler System object. This object scrambles the input data by using a linear feedback shift register that you specify with the Polynomial property.

`scrambler = comm.Scrambler(base,poly,cond)` creates the scrambler object with the CalculationBase property set to `base`, the Polynomial property set to `poly`, and the InitialConditions property set to `cond`.

Example: `comm.Scrambler(8,'1 + z^-2 + z^-3 + z^-5 + z^-7',[0 3 2 2 5 1 7])` sets the calculation base to 8, and the scrambler polynomial and initial conditions as specified.

`scrambler = comm.Scrambler( ___ ,Name,Value)` sets properties using one or more name-value pairs and either of the previous syntaxes. Enclose each property name in single quotes.

Example: `comm.Scrambler('CalculationBase',2)`

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`CalculationBase` — Range of input data
4 (default) | nonnegative integer

Range of input data used in the scrambler for modulo operations, specified as a nonnegative integer. The input and output of this object are integers from `0` to `CalculationBase` – `1`.

Data Types: `double`

`Polynomial` — Connections for linear feedback shift registers
`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Connections for linear feedback shift registers in the scrambler, specified as a character vector, integer vector, or binary vector. The `Polynomial` property defines if each switch in the scrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the scrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + ...$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that have a coefficient of 1. In this case, the order of the scramble polynomial is one less than the binary vector length.

Example: `'1 + z^-6 + z^-8'`, `[0 -6 -8]`, and `[1 0 0 0 0 0 1 0 1]` all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Data Types: `double` | `char`

`InitialConditionsSource` — Initial conditions source
`'Property'` (default) | `'Input port'`

- `'Property'` – Specify scrambler initial conditions by using the InitialConditions property.
- `'Input port'` – Specify scrambler initial conditions by using an additional input argument, initcond, when calling the object.

Data Types: `char`

`InitialConditions` — Initial conditions of scrambler registers
`[0 1 2 3]` (default) | nonnegative integer vector

Initial conditions of scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of `InitialConditions` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

`ResetInputPort` — Scrambler state reset port
`false` (default) | `true`

Scrambler state reset port, specified as `false` or `true`. If `ResetInputPort` is `true`, you can reset the scrambler object by using an additional input argument, reset, when calling the object.

Dependencies

This property is available when InitialConditionsSource is set to `'Property'`.

Usage

Syntax

```
scrambledOut = scrambler(signal)
scrambledOut = scrambler(signal,initcond)
scrambledOut = scrambler(signal,reset)
```

Description

`scrambledOut = scrambler(signal)` scrambles the input signal. The output is the same data type and length as the input vector.

`scrambledOut = scrambler(signal,initcond)` provides an additional input with values specifying the initial conditions of the linear feedback shift register.

This syntax applies when you set the InitialConditionsSource property of the object to `'Input port'`.

`scrambledOut = scrambler(signal,reset)` provides an additional input indicating whether to reset the state of the scrambler.

This syntax applies when you set InitialConditionsSource to `'Property'` and ResetInputPort to `true`.

Input Arguments

`signal` — Input signal
column vector

Input signal, specified as a column vector.

Example: `scrambledOut = scrambler([0 1 1 0 1])`

Data Types: `double` | `logical`

`initcond` — Initial register conditions
nonnegative integer column vector

Initial scrambler register conditions when the simulation starts, specified as a nonnegative integer column vector. The length of `initcond` must equal the order of the Polynomial property. The vector element values must be integers from `0` to CalculationBase – `1`.

Example: `scrambledOut = scrambler(signal,[0 1 1 0])` corresponds to possible initial register states for a scrambler with a polynomial order of 4 and a calculation base of 2 or higher.

Data Types: `double`

`reset` — Reset initial state of scrambler
scalar

Reset the initial state of the scrambler when the simulation starts, specified as a scalar. When the value of `reset` is nonzero, the object is reset before it is called.

Example: `scrambledOut = scrambler(signal,0)` scrambles the input signal without resetting the scrambler states.

Data Types: `double`

Output Arguments

`out` — Scrambled output
column vector

Scrambled output, returned as a column vector with the same data type and length as signal.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Scramble and Descramble Data

Scramble and descramble 8-ary data using `comm.Scrambler` and `comm.Descrambler` System objects™ having a calculation base of 8.

Create scrambler and descrambler objects, specifying the calculation base, polynomial, and initial conditions using input arguments. The scrambler and descrambler polynomials are specified with different but equivalent data formats.

```
N = 8;
scrambler = comm.Scrambler(N,'1 + z^-2 + z^-3 + z^-5 + z^-7', ...
    [0 3 2 2 5 1 7]);
descrambler = comm.Descrambler(N,[1 0 1 1 0 1 0 1], ...
    [0 3 2 2 5 1 7]);
```

Scramble and descramble random integers. Display the original data, scrambled data, and descrambled data sequences.

```
data = randi([0 N-1],5,1);
scrData = scrambler(data);
deScrData = descrambler(scrData);
[data scrData deScrData]
```

```
ans = 5×3

     6     7     6
     7     5     7
     1     7     1
     7     0     7
     5     3     5
```

Verify that the descrambled data matches the original data.

```
isequal(data,deScrData)
```

```
ans = logical
   1
```

Scramble and Descramble Data with Changing Initial Conditions

Scramble and descramble quaternary data while changing the initial conditions between function calls.

Create scrambler and descrambler System objects having a calculation base of 4. Set the `InitialConditionsSource` property to `'Input port'` so you can set the initial conditions as an argument to the object.

```
N = 4;
scrambler = comm.Scrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
descrambler = comm.Descrambler(N,'1 + z^-3','InitialConditionsSource','Input port');
```

Preallocate memory for the error vector which will be used to store errors output by the `symerr` function.

```
errVec = zeros(10,1);
```

Scramble and descramble random integers while changing the initial conditions, `initCond`, each time the loop executes. Use the `symerr` function to determine if the scrambling and descrambling operations result in symbol errors.

```
for k = 1:10
    initCond = randperm(3)';
    data = randi([0 N-1],5,1);
    scrData = scrambler(data,initCond);
    deScrData = descrambler(scrData,initCond);
    errVec(k) = symerr(data,deScrData);
end
```

Examine `errVec` to verify that the output from the descrambler matches the original data.

```
errVec
```

```
errVec = 10×1

     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
```

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.Descrambler` | `comm.PNSequence`

Blocks

Scrambler

Introduced in R2012a

comm.SphereDecoder

Package: `comm`

Decode input using sphere decoder

Description

The `Sphere Decoder` System object decodes the symbols sent over N_T antennas using the sphere decoding algorithm.

To decode input symbols using a sphere decoder:

1. Define and set up your sphere decoder object. See "Construction" on page 3-1367.
2. Call `step` to decode input symbols according to the properties of `comm.SphereDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.SphereDecoder` creates a System object, `H`. This object uses the sphere decoding algorithm to find the maximum-likelihood solution for a set of received symbols over a MIMO channel with N_T transmit antennas and N_R receive antennas.

`H = comm.SphereDecoder(Name,Value)` creates a sphere decoder object, `H`, with the specified property name set to the specified value. Name must appear inside single quotes (''). You can specify several name-value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

`H = comm.SphereDecoder(CONSTELLATION,BITTABLE)` creates a sphere decoder object, `H`, with the Constellation property set to `CONSTELLATION`, and the `BitTable` property set to `BITTABLE`.

Properties

`Constellation`

Signal constellation per transmit antenna

Specify the constellation as a complex column vector containing the constellation points to which the transmitted bits are mapped. The default setting is a QPSK constellation with an average power of `1`. The length of the vector must be a power of two. The object assumes that each transmit antenna uses the same constellation.

`BitTable`

Bit mapping used for each constellation point.

Specify the bit mapping for the symbols that the `Constellation` property specifies as a numerical matrix. The default is `[0 0; 0 1; 1 0; 1 1]`, which matches the default `Constellation` property value.

The matrix size must be `[ConstellationLength bitsPerSymbol]`. `ConstellationLength` represents the length of the `Constellation` property. `bitsPerSymbol` represents the number of bits that each symbol encodes.

`InitialRadius`

Initial search radius of the decoding algorithm.

Specify the initial search radius for the decoding algorithm as either `Infinity` | `ZF Solution`. The default is `Infinity`.

When you set this property to `Infinity`, the object sets the initial search radius to `Inf`.

When you set this property to `ZF Solution`, the object sets the initial search radius to the zero-forcing solution. This calculation uses the pseudo-inverse of the input channel when decoding. Large constellations and/or antenna counts can benefit from the initial reduction in the search radius. In most cases, however, the extra computation of the `ZF Solution` will not provide a benefit.

`DecisionType`

Specify the decoding decision method as either `Soft` | `Hard`. The default is `Soft`.

When you set this property to `Soft`, the decoder outputs log-likelihood ratios (LLRs), or soft bits.

When you set this property to `Hard`, the decoder converts the soft LLRs to bits. The hard-decision output logical array follows the mapping of a zero for a negative LLR and one for all other values.

Methods

step	Decode received symbols using sphere decoding algorithm

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Decode Using a Sphere Decoder

Modulate a set of bits using 16-QAM constellation. Transmit the signal as two parallel streams over a MIMO channel. Decode using a sphere decoder with perfect channel knowledge.

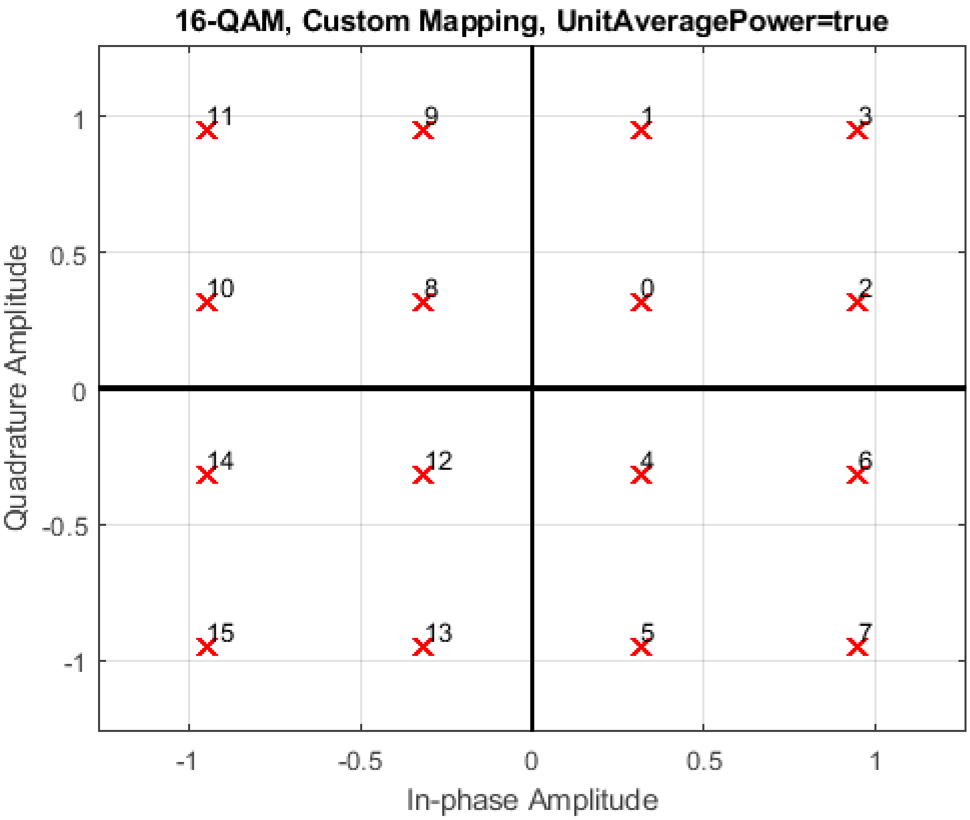
Specify the modulation order, the number of transmitted bits, the Eb/No ratio, and the symbol mapping.

```
bps = 4; % Bits per symbol
M = 2^bps; % Modulation order
nBits = 1e3*bps;
```

```
ebno = 10;
symMap = [11 10 14 15 9 8 12 13 1 0 4 5 3 2 6 7];
```

Generate and display the symbol mapping of the 16-QAM modulator by using the `qammod` function and the custom symbol map.

```
sym = qammod(symMap(1:M)',M,symMap,'UnitAveragePower',true,'PlotConstellation',true);
```



Convert the decimal value of the symbol map to binary bits using the left bit as the most significant bit (msb). The `M`-by-`bps` matrix `bitTable` is used by the sphere decoder.

```
bitTable = de2bi(symMap,bps,'left-msb');
```

Create a 2x2 MIMO Channel System object with `PathGainsOutputPort` set to `true` to use the path gains as a channel estimate. To ensure the repeatability of results, set the object to use the global random number stream.

```
mimo = comm.MIMOChannel( ...
    'PathGainsOutputPort',true, ...
    'RandomStream','Global stream');
```

Create an AWGN Channel System object.

```
awgnChan = comm.AWGNChannel('EbNo',ebno,'BitsPerSymbol',bps);
```

Create a Sphere Decoder System object that processes bits using hard-decision decoding. Configure using the custom bit table and symbol map.

```
sphDec = comm.SphereDecoder('Constellation',sym, ...
    'BitTable',bitTable,'DecisionType','Hard');
```

Create an error rate System object.

```
berRate = comm.ErrorRate;
```

Set the global random number generator seed.

```
rng(37)
```

Generate a random data stream.

```
data = randi([0 1],nBits,1);
```

Modulate the data and reshape it into two streams to be used with the 2x2 MIMO channel.

```
modData = qammod(data,M,symMap,'InputType','bit','UnitAveragePower',true);
modData = reshape(modData,[],2);
```

Pass the modulated data through the MIMO fading channel and add AWGN.

```
[fadedSig,pathGains] = mimo(modData);
rxSig = awgnChan(fadedSig);
```

Decode the received signal using `pathGains` as a perfect channel estimate.

```
decodedData = sphDec(rxSig,squeeze(pathGains));
```

Convert the decoded hard-decision data, which is a logical matrix, into a double column vector to enable the calculation of error statistics. Calculate and display the bit error rate and the number of errors.

```
dataOut = double(decodedData(:));
errorStats = berRate(data,dataOut);
errorStats(1:2)
```

```
ans = 2×1

    0.0380
  152.0000
```

Algorithm

This object implements a soft-output max-log a posteriori probability (APP) MIMO detector by means of a soft-output Schnorr-Euchner sphere decoder (SESD), implemented as single tree search (STS) tree traversal. The algorithm assumes the same constellation and bit table on all of the transmit antennas. Given as inputs, the received symbol vector and the estimated channel matrix, the algorithm outputs the log-likelihood ratios (LLRs) of the transmitted bits.

The algorithm assumes a MIMO system model with N_{T} transmit antennas and N_{R} receive antennas where N_{T} symbols are simultaneously sent, expressed as:

$y = Hs + n.$

where y is the received symbols, H is the MIMO channel matrix, s is the transmitted symbol vector, and n is the thermal noise.

The MIMO detector seeks the maximum-likelihood (ML) solution, $\widehat{s}_{ML}$, such that:

$$\widehat{s}_{ML} = \underset{s \in o}{\operatorname{argmin}} \|y - Hs\|^2$$

where O is the complex-valued constellation from which the N_{T} elements of s are chosen.

Soft detection also computes a log-likelihood ratio (LLR) for each bit that serves as a measure of the reliability of the estimate for each bit. The LLR is calculated as using the max-log approximation:

$$L(x_{j,b}) = \underbrace{\min_{s \in x_{j,b}^{(0)}} \|y - Hs\|^2}_{\lambda^{ML}} - \underbrace{\min_{s \in x_{j,b}^{(1)}} \|y - Hs\|^2}_{\lambda_{j,b}^{\overline{ML}}}$$

where

- $L(x_{j,b})$ is the LLR estimate for each bit.
- $x_{j,b}$ is each sent bit, the bth bit of the jth symbol.
- $x_{j,b}^{(0)}$ and $x_{j,b}^{(1)}$ are the disjoint sets of vector symbols that have the bth bit in the label of the jth scalar symbol equal to 0 and 1, respectively. The two λ symbols denotes the distance calculated as norm squared., specifically:
  - λ^{ML} is the distance $\widehat{s}_{ML}$.
  - $\lambda_{j,b}^{\overline{ML}}$ is the distance to the counter-hypothesis, which denotes the binary complement of the bth bit in the binary label of the jth entry of $\widehat{s}_{ML}$, specifically the minimum of the symbol set $x_{j,b}^{\left(x_{j,b}^{\overline{ML}}\right)}$, which contains all of the possible vectors for which the bth bit of the jth entry is flipped compared to the same entry of $\widehat{s}_{ML}$.

Based on whether $x_{j,b}^{\left(x_{j,b}^{ML}\right)}$ is `0` or `1`, the LLR estimate for bit $x_{j,b}$ is computed as follows:

$$L(x_{j,b}) = \begin{cases} \lambda^{ML} - \lambda_{j,b}^{\overline{ML}}, & x_{j,b}^{ML} = 0 \\ \lambda_{j,b}^{\overline{ML}} - \lambda^{ML}, & x_{j,b}^{ML} = 1 \end{cases}$$

The design of a decoder strives to efficiently find $\widehat{s}_{ML}$, λ^{ML}, and $\lambda_{j,b}^{\overline{ML}}$.

This search can be converted into a tree search by means of the sphere decoding algorithms. To this end, the channel matrix is decomposed into $H = QR$ by means of a QR decomposition. Left-multiplying y by Q^{H}, the problem can be reformulated as:

$$\lambda^{ML} = \underset{s \in o}{\arg\min} \|\bar{y} - Rs\|^2$$

$$\lambda_{j,b}^{\overline{ML}} = \underset{s \in x_{j,b}^{\left(x_{j,b}^{\overline{ML}}\right)}}{\text{argmin}} \|\bar{y} - Rs\|^2$$

Using this reformulated problem statement, the triangular structure of R can be exploited to arrange a tree structure such that each of the leaf nodes corresponds to a possible s vector and the partial distances to the nodes in the tree can be calculated cumulatively adding to the partial distance of the parent node.

In the STS algorithm, the λ^{ML} and $\lambda_{j,b}^{\overline{ML}}$ metrics are searched concurrently. The goal is to have a list containing the metric λ^{ML}, along with the corresponding bit sequence x^{ML} and the metrics $x_{j,b}^{\left(x_{j,b}^{\overline{ML}}\right)}$ of all counter-hypotheses. The sub-tree originating from a given node is searched only if the result can lead to an update of either λ^{ML} or $\lambda_{j,b}^{\overline{ML}}$.

The STS algorithm flow can be summarized as:

1. If when reaching a leaf node, a new ML hypothesis is found $\left(d(x) < \lambda^{ML}\right)$, all $\lambda_{j,b}^{\overline{ML}}$ for which $x_{j,b} = x_{j,b}^{\overline{ML}}$ are set to λ^{ML} which now turns into a valued counter-hypothesis. Then, λ^{ML} is set to the current distance, $d(x)$.
2. If $d(x) \geq \lambda^{ML}$, only the counter-hypotheses have to be checked. For all j and b for which $\left(d(x) < \lambda^{ML}\right)$ and $x_{j,b} = x_{j,b}^{\overline{ML}}$, the decoder updates $\lambda_{j,b}^{\overline{ML}}$ to be $d(x)$.
3. A sub-tree is pruned if the partial distance of the node is bigger than the current $\lambda_{j,b}^{\overline{ML}}$ which may be affected when traversing the subtree.
4. The STS concludes once all of the tree nodes have been visited once or pruned.

Limitations

- The output LLR values are not scaled by the noise variance. For coded links employing iterative coding (LDPC or turbo) or MIMO OFDM with Viterbi decoding, the output LLR values should be scaled by the channel state information to achieve better performance.

Selected Bibliography

[1] Studer, C., A. Burg, and H. Bölcskei. “Soft-Output Sphere Decoding: Algorithms and VLSI Implementation”. *IEEE Journal of Selected Areas in Communications*. Vol. 26, No. 2, February 2008, pp. 290–300.

[2] Cho, Y. S., et.al. "MIMO-OFDM Wireless communications with MATLAB," IEEE Press, 2011.

[3] Hochwald, B.M., S. ten Brink. “Achieving near-capacity on a multiple-antenna channel”, IEEE Transactions on Communications, Vol. 51, No. 3, Mar 2003, pp. 389-399.

[4] Agrell, E., T. Eriksson, A. Vardy, K. Zeger. “Closest point search in lattices”, IEEE Transactions on Information Theory, Vol. 48, No. 8, Aug 2002, pp. 2201-2214.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Sphere Decoder | `comm.LTEMIMOChannel` | `comm.MIMOChannel` | `comm.OSTBCCombiner`

Introduced in R2013a

step

System object: `comm.SphereDecoder`
Package: `comm`

Decode received symbols using sphere decoding algorithm

Syntax

```
Y = step(H, RXSYMBOLS, CHAN)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H, RXSYMBOLS, CHAN)` decodes the received symbols, RXSYMBOLS, using the sphere decoding algorithm. The algorithm can be employed to decode Ns channel realizations in one call, where in each channel realization, Nr symbols are received.

The inputs are:

`RXSYMBOLS`: a [Ns Nr] complex double matrix containing the received symbols.

`CHAN`: a [Ns Nt Nr] or [1 Nt Nr] complex double matrix representing the fading channel coefficients of the flat-fading MIMO channel. For the [Ns Nt Nr] case, the object applies each channel matrix to each Nr symbol set. For the block fading case, i.e., when the size of `CHAN` is [1 Nt Nr], the same channel is applied to all of the received symbols.

The output `Y`, which depends on the setting of the `DecisionType` property, is a double matrix containing the Log-Likelihood Ratios (LLRs) of the decoded bits or the bits themselves. For both cases, the size of the output is [Ns*bitsPerSymbol Nt], where `bitsPerSymbol` represents the number of bits per transmitted symbol, as determined by the `BitTable` property.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.SymbolSynchronizer

Package: `comm`

Correct symbol timing clock skew

Description

The `comm.SymbolSynchronizer` System object corrects symbol timing clock skew between a single-carrier transmitter and receiver for PAM, PSK, QAM, and OQPSK modulation schemes. For more information, see “Symbol Synchronization Overview” on page 3-1393.

Note The input signal operates on a sample-rate basis and the output signal operates on a symbol-rate basis.

To correct symbol timing clock skew:

1. Create the `comm.SymbolSynchronizer` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
symbolSync = comm.SymbolSynchronizer
symbolSync = comm.SymbolSynchronizer(Name,Value)
```

Description

`symbolSync = comm.SymbolSynchronizer` creates a symbol synchronizer System object for correcting the clock skew between a single-carrier transmitter and receiver.

`symbolSync = comm.SymbolSynchronizer(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.SymbolSynchronizer('Modulation','OQPSK')` configures the symbol synchronizer System object for an OQPSK-modulated input signal. Enclose each property name in quotes.

Tunable `DampingFactor`, `NormalizedLoopBandwidth`, and `DetectorGain` properties enable you to optimize synchronizer performance in your simulation loop without releasing the object.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

Name-Value Pair Arguments

`Modulation` — Modulation type

`'PAM/PSK/QAM'` (default) | `'OQPSK'`

Modulation type, specified as `'PAM/PSK/QAM'` or `'OQPSK'`.

Tunable: No

Data Types: `char` | `string`

`TimingErrorDetector` — Timing error detector method

`Zero-Crossing (decision-directed)` (default) | `Gardner (non-data-aided)` | `Early-Late (non-data-aided)` | `Mueller-Muller (decision-directed)`

Timing error detector method, specified as `Zero-Crossing (decision-directed)`, `Gardner (non-data-aided)`, `Early-Late (non-data-aided)`, or `Mueller-Muller (decision-directed)`. This property assigns the timing error detection scheme used in the synchronizer. For more information, see "Timing Error Detection (TED)" on page 3-1394.

Tunable: No

Data Types: `char` | `string`

`SamplesPerSymbol` — Samples per symbol

`2` (default) | integer greater than 1

Samples per symbol, specified as an integer greater than 1.

Tunable: No

Data Types: `double`

`DampingFactor` — Damping factor of loop filter

`1` (default) | positive scalar

Damping factor of the loop filter, specified as a positive scalar. For more information, see "Loop Filter" on page 3-1397.

Tunable: Yes

Data Types: `double` | `single`

`NormalizedLoopBandwidth` — Normalized bandwidth of loop filter

`0.01` (default) | scalar in the range (0, 1)

Normalized bandwidth of the loop filter, specified as a scalar in the range (0, 1). The loop bandwidth is normalized to the sample rate of the input signal. For more information, see "Loop Filter" on page 3-1397.

Note To ensure the symbol synchronizer locks, set the `NormalizedLoopBandwidth` property to a value less than `0.1`.

Tunable: Yes

Data Types: `double` | `single`

`DetectorGain` — Phase detector gain

`2.7` (default) | positive scalar

Phase detector gain, specified as a positive scalar.

Tunable: Yes

Data Types: `double` | `single`

Usage

Note For versions earlier than R2016b, use the `step` function to run the System object™ algorithm. The arguments to `step` are the object you created, followed by the arguments shown in this section.

For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Syntax

`symbols = symbolSync(samples)`

Description

`symbols = symbolSync(samples)` corrects symbol timing clock skew between a single-carrier transmitter and receiver based on the input samples and outputs synchronized symbols.

- The input operates on a sample-rate basis and the output signal operates on a symbol-rate basis.
- You can tune the `DampingFactor`, `NormalizedLoopBandwidth`, and `DetectorGain` properties to improve the synchronizer performance.

Input Arguments

`samples` — Input samples

scalar (default) | column vector

Input samples, specified as a scalar or column vector of a PAM-, PSK-, QAM-, or OQPSK-modulated single-carrier signal.

Data Types: `double` | `single`
Complex Number Support: Yes

Output Arguments

`symbols` — Synchronized symbols

column vector

Synchronized symbols, returned as a variable-sized column vector. The output symbols inherit the data type from the input samples. For an input with dimensions N_{samp}-by-1, this output has dimensions N_{sym}-by-1. N_{sym} is approximately equal to N_{samp} divided by N_{sps}, where N_{sps} is equal to the

`SamplesPerSymbol` property value. If the output exceeds the maximum output size of $\left\lceil \frac{N_{\text{samp}}}{N_{\text{sps}}} \times 1.1 \right\rceil$, it is truncated.

`timingErr` — Estimated timing error
scalar in the range [0, 1] | column vector of elements in the range [0, 1]

Estimated timing error for each input sample, returned as a scalar in the range [0, 1] or column vector of elements in the range [0, 1]. The estimated timing error is normalized to the input sample rate. `timingErr` has the same data type and size as input `samples`.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Specific to `comm.SymbolSynchronizer`

clone	Create duplicate System object
isLocked	Determine if System object is in use

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Correct Symbol Timing Error of QPSK-Modulated Signal

Correct a fixed symbol timing error on a noisy QPSK-modulated signal. Check the bit error rate (BER) of the synchronized received signal.

Initialize simulation parameters.

```
M = 4;          % Modulation order for QPSK
nSym = 5000;    % Number of symbols in a packet
sps = 4;        % Samples per symbol
timingErr = 2; % Samples of timing error
snr = 15;       % Signal-to-noise ratio (dB)
```

Create root raised cosine (RRC) transmit and receive filter System objects.

```
txfilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',sps);
rxfilter = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',sps,'DecimationFactor',2);
```

Create a symbol synchronizer System object to correct the timing error.

```
symbolSync = comm.SymbolSynchronizer;
```

Generate random M-ary symbols and apply QPSK modulation.

```
data = randi([0 M-1],nSym,1);
modSig = pskmod(data,M,pi/4);
```

Create a delay object to introduce a fixed timing error of 2 samples. Because the transmit RRC filter outputs 4 samples per symbol, 1 sample is equivalent to a 1/4 symbol through the fixed delay and channel.

```
fixedDelay = dsp.Delay(timingErr);
fixedDelaySym = ceil(fixedDelay.Length/sps); % Round fixed delay to nearest integer in symbols
```

Filter the modulated signal through a transmit RRC filter by using the `txfilter` object. Apply a signal timing error by using the `fixedDelay` object.

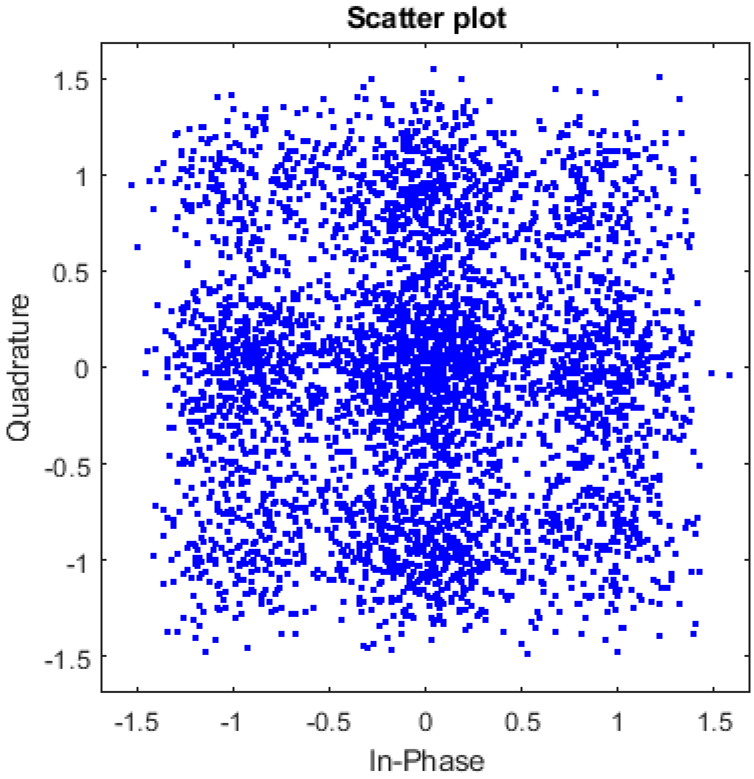
```
txSig = txfilter(modSig);
delaySig = fixedDelay(txSig);
```

Pass the delayed signal through an AWGN channel with a 15 dB signal-to-noise ratio.

```
rxSig = awgn(delaySig,snr,'measured');
```

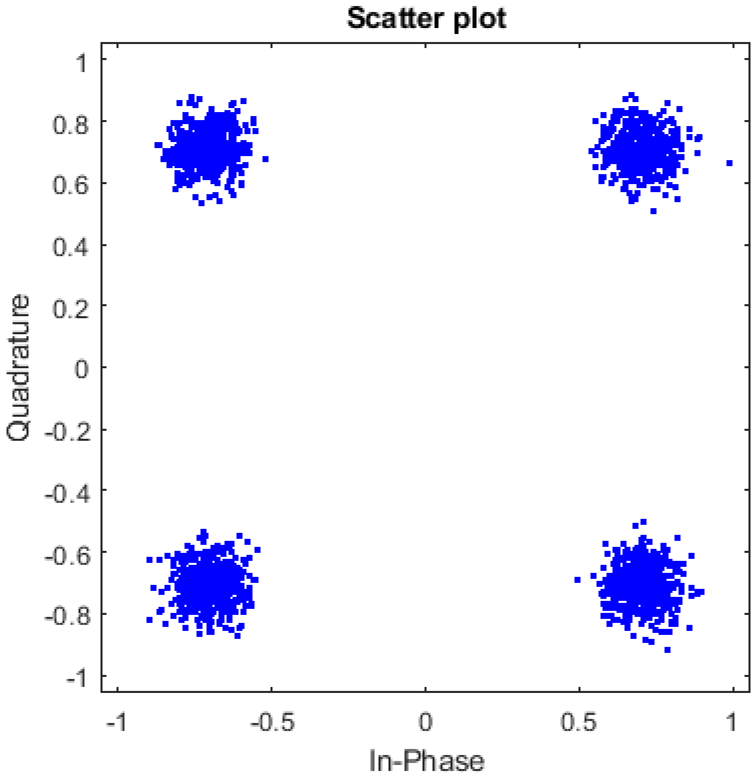
Filter the modulated signal through a receive RRC filter by using the `rxfilter` object. Display the scatter plot. Due to the timing error, the received signal does not align with the expected QPSK reference constellation.

```
rxSample = rxfilter(rxSig);
scatterplot(rxSample(1001:end),2)
```



Correct the symbol timing error by using the `symbolSync` object. Display the scatter plot. The synchronized signal now aligns with the expected QPSK constellation.

```
rxSync = symbolSync(rxSample);
scatterplot(rxSync(1001:end),2)
```



Demodulate the QPSK signal.

```
recData = pskdemod(rxSync,M,pi/4);
```

Compute, in symbols, the total system delay due to the fixed delay and the transmit and receive RRC filters.

```
sysDelay = dsp.Delay(fixedDelaySym + txfilter.FilterSpanInSymbols/2 + ...
    rxfilter.FilterSpanInSymbols/2);
```

Compute the BER, taking into account the system delay.

```
[numErr,ber] = biterr(sysDelay(data),recData)
```

```
numErr = 12
```

```
ber = 0.0012
```

Correct Symbol Timing Error of BPSK-Modulated Signal

Correct a fixed symbol timing error on a noisy BPSK transmission signal. Check the bit error rate (BER) of the synchronized received signal.

Initialize simulation parameters.

```
M = 2;          % Modulation order for BPSK
nSym = 20000;   % Number of symbols in a packet
sps = 4;        % Samples per symbol
timingErr = 2;  % Samples of timing error
snr = 15;       % Signal-to-noise ratio (dB)
```

Create root raised cosine (RRC) transmit and receive filter System objects.

```
txfilter = comm.RaisedCosineTransmitFilter(...
    'OutputSamplesPerSymbol',sps);
rxfilter = comm.RaisedCosineReceiveFilter(...
    'InputSamplesPerSymbol',sps,'DecimationFactor',1);
```

Create a symbol synchronizer System object™ to correct the timing error.

```
symbolSync = comm.SymbolSynchronizer(...
    'SamplesPerSymbol',sps, ...
    'NormalizedLoopBandwidth',0.01, ...
    'DampingFactor',1.0, ...
    'TimingErrorDetector','Early-Late (non-data-aided)');
```

Generate random data symbols and apply BPSK modulation.

```
data = randi([0 M-1],nSym,1);
modSig = pskmod(data,M);
```

Create a delay object to introduce a fixed timing error of 2 samples. Because the transmit RRC filter outputs 4 samples per symbol, 1 sample is equivalent to a 1/4 symbol through the fixed delay and channel.

```
fixedDelay = dsp.Delay(timingErr);
fixedDelaySym = ceil(fixedDelay.Length/sps); % Round fixed delay to nearest integer in symbols
```

Filter the modulated signal through a transmit RRC filter by using the `txfilter` object. Apply a signal timing error by using the `fixedDelay` object.

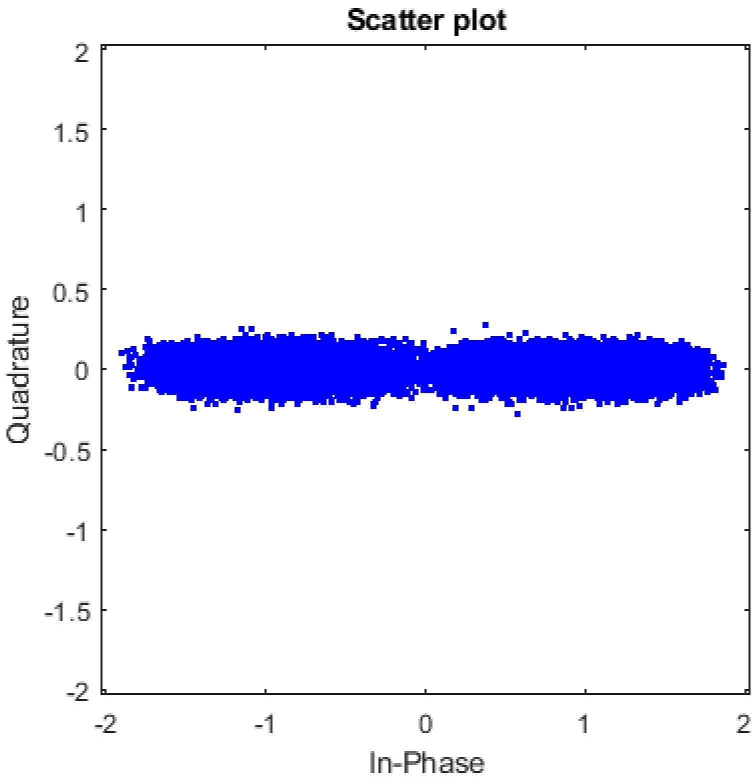
```
txSig = txfilter(modSig);
delayedSig = fixedDelay(txSig);
```

Pass the delayed signal through an AWGN channel.

```
rxSig = awgn(delayedSig,snr,'measured');
```

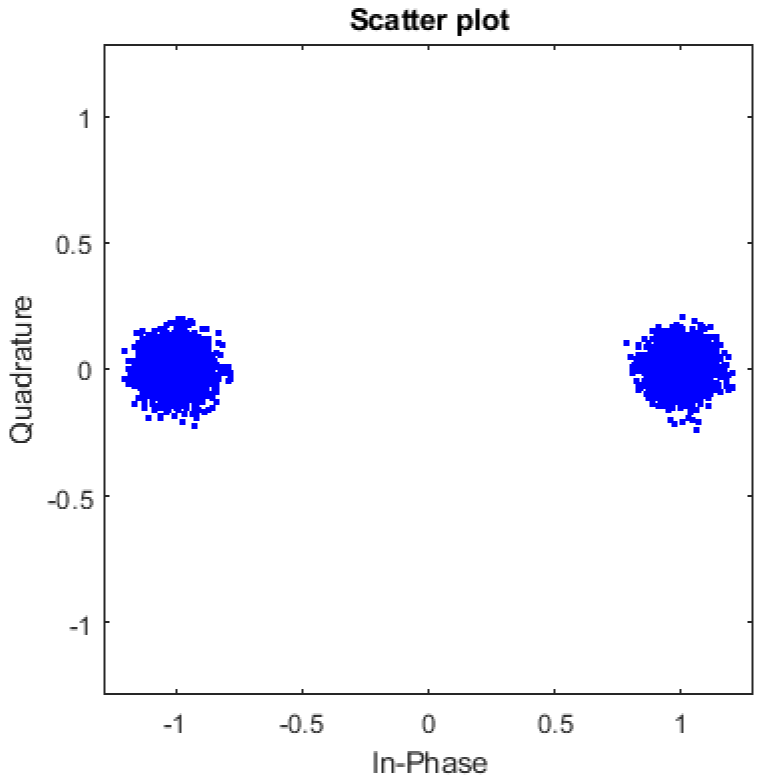
Filter the modulated signal through a receive RRC filter by using the `rxfilter` object. Display the scatter plot. Due to the timing error, the received signal does not align with the expected BPSK reference constellation.

```
rxSample = rxfilter(rxSig);
scatterplot(rxSample(10000:end),2)
```



Correct the symbol timing error by using the `symbolSync` object. Display the scatter plot. The synchronized signal now aligns with the expected BPSK constellation.

```
rxSync = symbolSync(rxSample);
scatterplot(rxSync(10000:end),2)
```



Demodulate the BPSK signal.

```
recData = pskdemod(rxSync,M);
```

Compute, in symbols, the total system delay due to the fixed delay and the transmit and receive RRC filters.

```
sysDelay = dsp.Delay(fixedDelaySym + txfilter.FilterSpanInSymbols/2 + ...
    rxfilter.FilterSpanInSymbols/2);
```

Compute the BER, taking into account the system delay.

```
[numErr1,ber1] = biterr(sysDelay(data),recData)
```

```
numErr1 = 8
```

```
ber1 = 4.0000e-04
```

Correct Symbol Timing and Doppler Offsets

Correct symbol timing and frequency offset errors by using the `comm.SymbolSynchronizer` and `comm.CarrierSynchronizer` System objects.

Configuration

Initialize simulation parameters.

```
M = 16;             % Modulation order
nSym = 2000;        % Number of symbols in a packet
sps = 2;            % Samples per symbol
spsFilt = 8;        % Samples per symbol for filters and channel
spsSync = 2;        % Samples per symbol for synchronizers
lenFilt = 10;       % RRC filter length
```

Create a matched pair of root raised cosine (RRC) filter System objects for transmitter and receiver.

```
txfilter = comm.RaisedCosineTransmitFilter('FilterSpanInSymbols',lenFilt, ...
    'OutputSamplesPerSymbol',spsFilt,'Gain',sqrt(spsFilt));
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',lenFilt, ...
    'InputSamplesPerSymbol',spsFilt,'DecimationFactor',spsFilt/2,'Gain',sqrt(1/spsFilt));
```

Create a phase-frequency offset System object to introduce a 100 Hz Doppler shift.

```
doppler = comm.PhaseFrequencyOffset('FrequencyOffset',100, ...
    'PhaseOffset',45,'SampleRate',1e6);
```

Create a variable delay System object to introduce timing offsets.

```
varDelay = dsp.VariableFractionalDelay;
```

Create carrier and symbol synchronizer System objects to correct for Doppler shift and timing offset, respectively.

```
carrierSync = comm.CarrierSynchronizer('SamplesPerSymbol',spsSync);
symbolSync = comm.SymbolSynchronizer( ...
    'TimingErrorDetector','Early-Late (non-data-aided)', ...
    'SamplesPerSymbol',spsSync);
```

Create constellation diagram System objects to view the results.

```
refConst = qammod(0:M-1,M,'UnitAveragePower',true);
cdReceive = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsFilt,'Title','Received Signal');
cdDoppler = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsSync,'Title','Frequency Corrected Signal');
cdTiming = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'SamplesPerSymbol',spsSync,'Title','Frequency and Timing Synchronized Signal');
```

Main Processing Loop

The main processing loop:

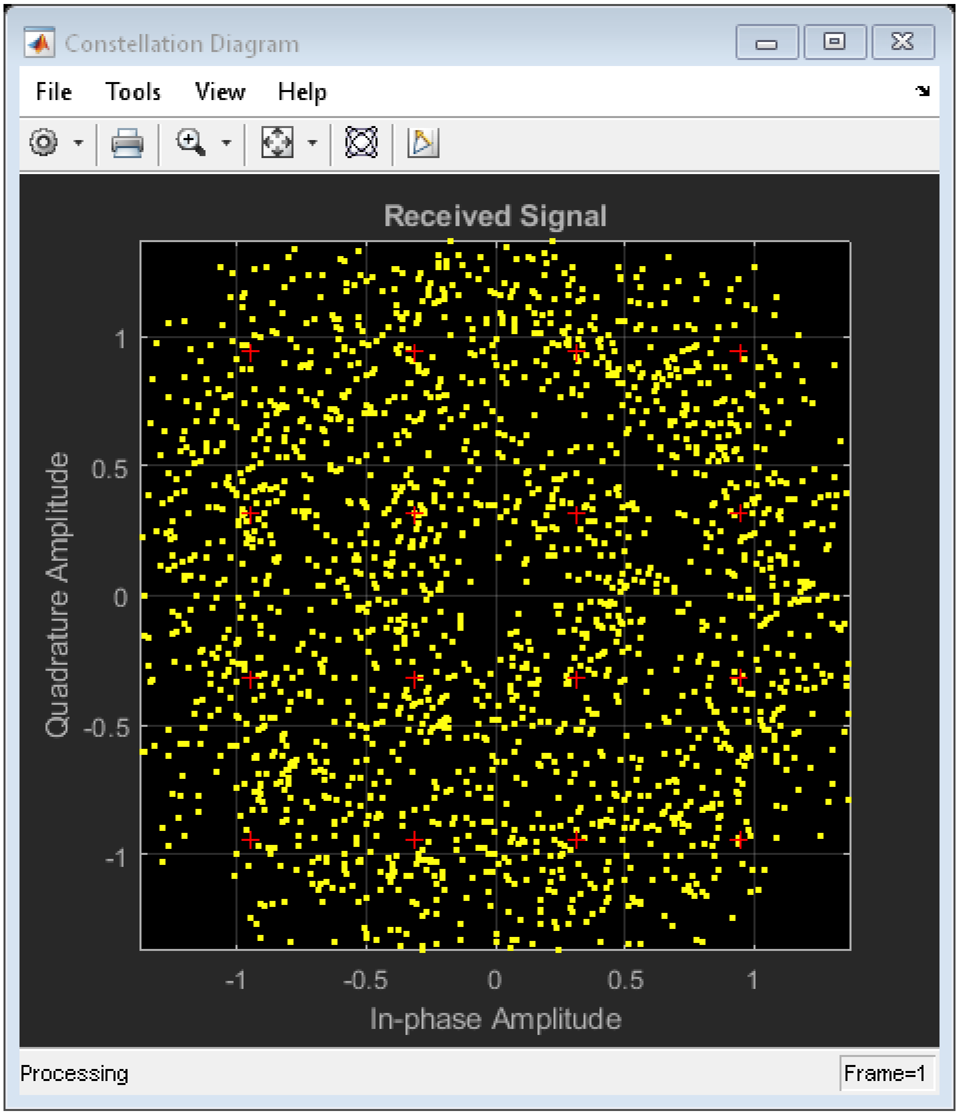
- Generates random symbols and apply QAM modulation.
- Filters the modulated signal.
- Applies frequency and timing offsets.
- Passes the transmitted signal through an AWGN channel.
- Filters the received signal.
- Corrects the Doppler shift.
- Corrects the timing offset.

```
for k = 1:15
    data = randi([0 M-1],nSym,1);
    modSig = qammod(data,M,'UnitAveragePower',true);
    txSig = txfilter(modSig);

    txDoppler = doppler(txSig);
    txDelay = varDelay(txDoppler,k/15);

    rxSig = awgn(txDelay,25);

    rxFiltSig = rxfilter(rxSig);
    rxCorr = carrierSync(rxFiltSig);
    rxData = symbolSync(rxCorr);
end
```

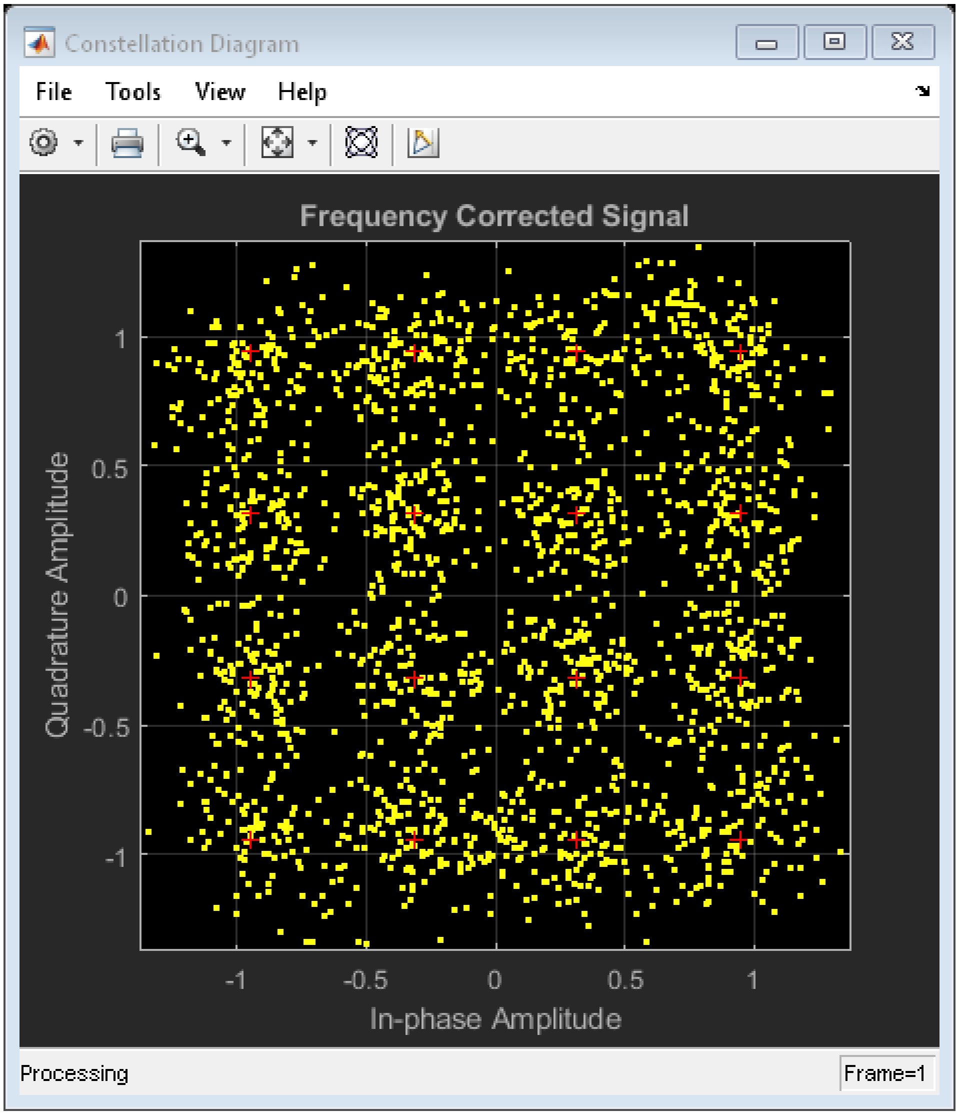
Visualization

Plot the constellation diagrams of the received signal, frequency corrected signal, and frequency and timing synchronized signal. Specific constellation points cannot be identified in the received signal and can be only partially identified in the frequency corrected signal. However, the timing and frequency synchronized signal aligns with the expected QAM constellation points.

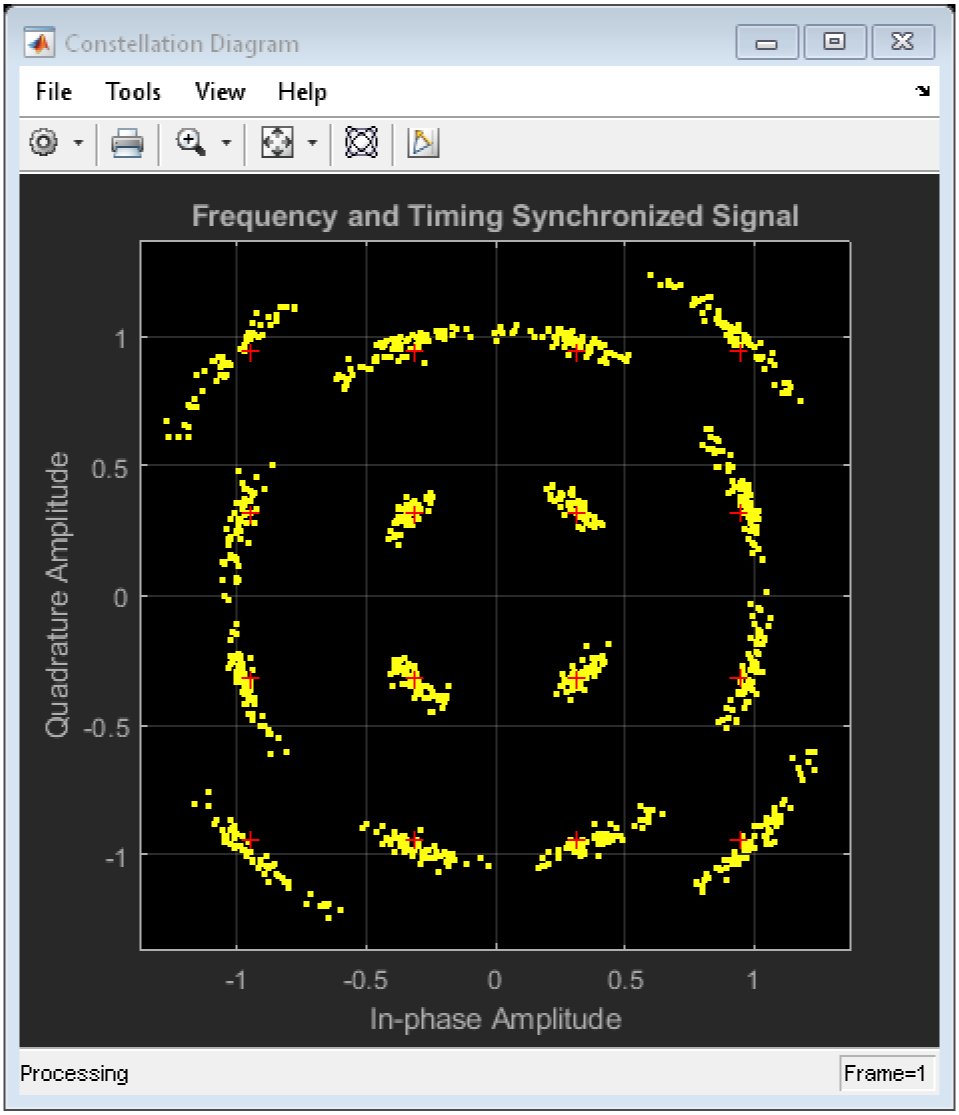
```
cdReceive(rxSig)
```



```
cdDoppler(rxCorr)
```



```
cdTiming(rxData)
```



Timing Error for Noisy 8-PSK Signal

Correct a monotonically increasing symbol timing error on a noisy 8-PSK signal. Display the normalized timing error.

Initialize simulation parameters.

```
M = 8;              % Modulation order
nSym = 5000;        % Number of symbol in a packet
sps = 2;            % Samples per symbol
nSamp = sps*nSym; % Number of samples in a packet
```

Create root raised cosine (RRC) transmit and receive filter System objects.

```
txfilter = comm.RaisedCosineTransmitFilter( ...
    'OutputSamplesPerSymbol',sps);
```

```
rxfilter = comm.RaisedCosineReceiveFilter( ...
    'InputSamplesPerSymbol',sps, ...
    'DecimationFactor',1);
```

Create a variable fractional delay System object™ to introduce a monotonically increasing timing error.

```
varDelay = dsp.VariableFractionalDelay;
```

Create a symbol synchronizer System object to correct the timing error.

```
symbolSync = comm.SymbolSynchronizer(...
    'TimingErrorDetector','Mueller-Muller (decision-directed)', ...
    'SamplesPerSymbol',sps);
```

Generate random 8-ary symbols and apply 8-PSK modulation.

```
data = randi([0 M-1],nSym,1);
modSig = pskmod(data,M,pi/8);
```

Filter the modulated signal through a raised cosine transmit filter and apply a monotonically increasing timing delay.

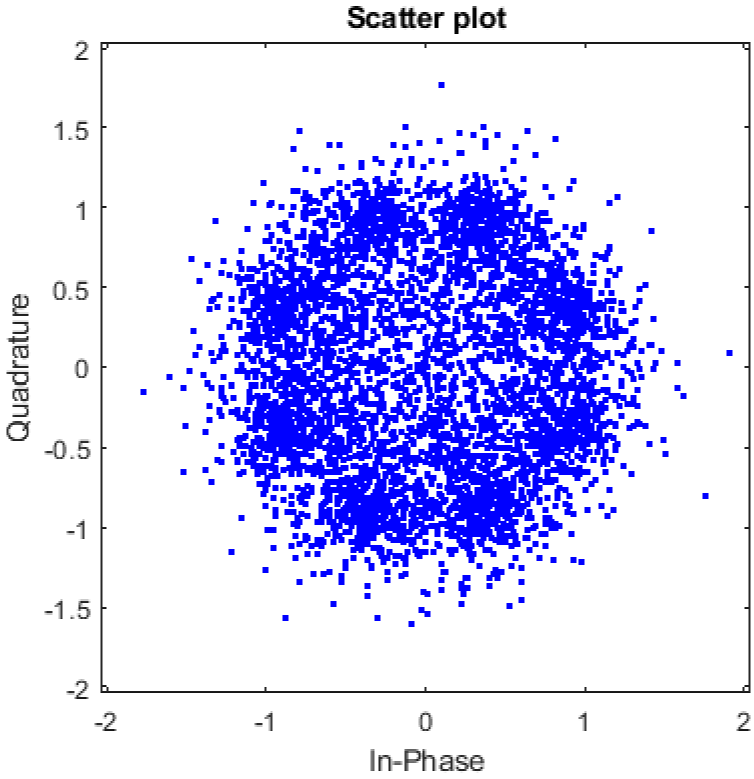
```
vdelay = (0:1/nSamp:1-1/nSamp)';
txSig = txfilter(modSig);
delaySig = varDelay(txSig,vdelay);
```

Pass the delayed signal through an AWGN channel with a 15 dB signal-to-noise ratio.

```
rxSig = awgn(delaySig,15,'measured');
```

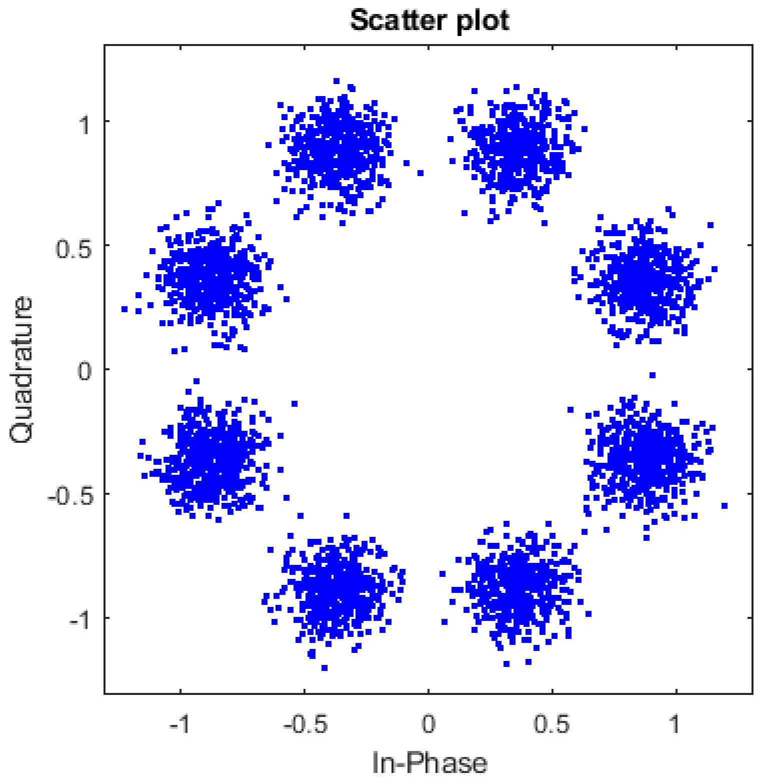
Filter the modulated signal through a receive RRC filter. Display the scatter plot. Due to the timing error, the received signal does not align with the expected 8-PSK reference constellation.

```
rxSample = rxfilter(rxSig);
scatterplot(rxSample,sps)
```



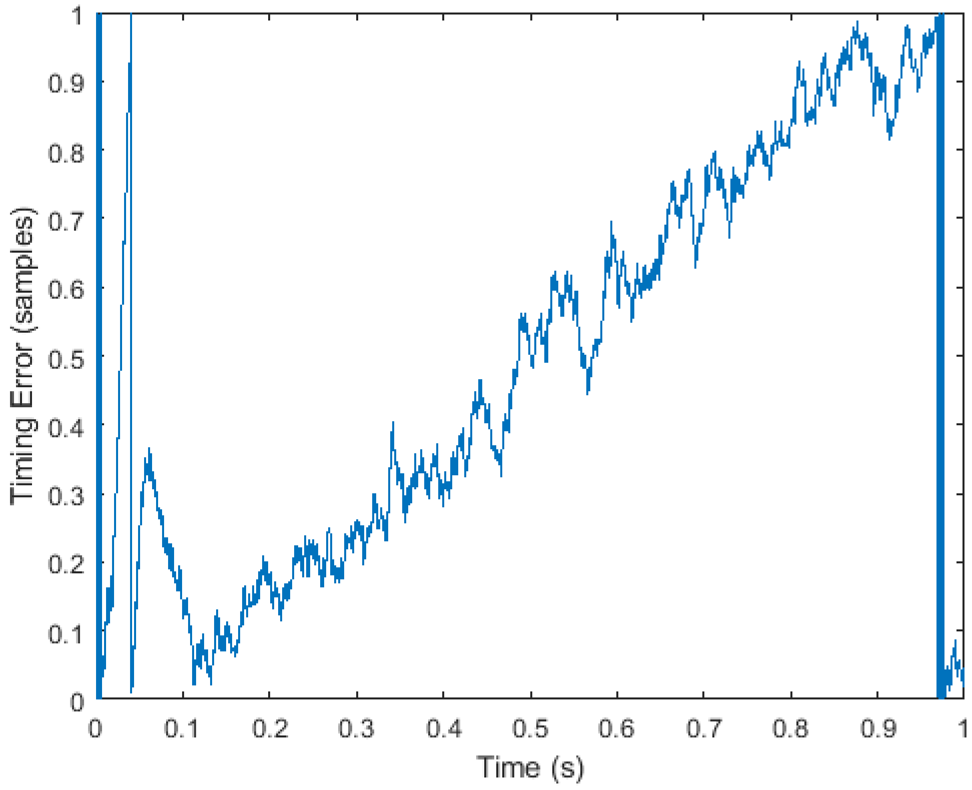
Correct the symbol timing error by using the `symbolSync` object. Display the scatter plot. The synchronized signal now aligns with the expected 8-PSK constellation.

```
[rxSym,tError] = symbolSync(rxSample);
scatterplot(rxSym(1001:end))
```



Plot the timing error estimate. Over time, the normalized timing error increases to 1 sample.

```
figure
plot(vdelay,tError)
xlabel('Time (s)')
ylabel('Timing Error (samples)')
```



More About

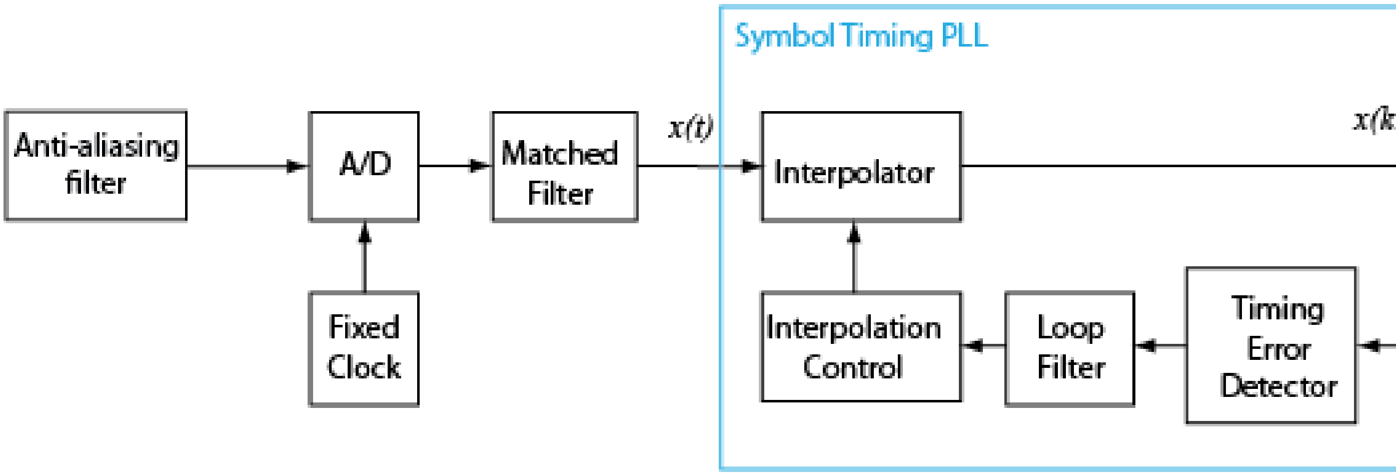
Symbol Synchronization Overview

The symbol timing synchronizer algorithm is based on a phased lock loop (PLL) algorithm that consists of four components:

- A timing error detector (TED)
- An interpolator
- An interpolation controller
- A loop filter

For OQPSK modulation, the in-phase and quadrature signal components are first aligned (as in QPSK modulation) using a state buffer to cache the last half symbol of the previous input. After initial alignment, the remaining synchronization process is the same as for QPSK modulation.

This block diagram shows an example of a timing synchronizer. In the figure, the symbol timing PLL operates on $x(t)$, the received sample signal after matched filtering. The symbol timing PLL outputs the symbol signal, $x(kT_s + \hat{\tau})$, after correcting for the clock skew between the transmitter and receiver.



Timing Error Detection (TED)

The symbol timing synchronizer supports non-data-aided TED and decision-directed TED methods. This table shows the timing estimate expressions for the TED method options.

TED Method	Expression
Zero-crossing (decision-directed)	$e(k) = x((k-1/2)T_s + \hat{\tau})[\hat{a}_0(k-1) - \hat{a}_0(k)] + y((k-1/2)T_s + \hat{\tau})[\hat{a}_1(k-1) - \hat{a}_1(k)]$
Gardner (non-data-aided)	$e(k) = x((k-1/2)T_s + \hat{\tau})[x((k-1)T_s + \hat{\tau}) - x(kT_s + \hat{\tau})] + y((k-1/2)T_s + \hat{\tau})[y((k-1)T_s + \hat{\tau}) - y(kT_s + \hat{\tau})]$
Early-late (non-data-aided)	$e(k) = x(kT_s + \hat{\tau})[x((k+1/2)T_s + \hat{\tau}) - x((k-1/2)T_s + \hat{\tau})] + y(kT_s + \hat{\tau})[y((k+1/2)T_s + \hat{\tau}) - y((k-1/2)T_s + \hat{\tau})]$
Mueller-Muller (decision-directed)	$e(k) = \hat{a}_0(k-1)x(kT_s + \hat{\tau}) - \hat{a}_0(k)x((k-1)T_s + \hat{\tau}) + \hat{a}_1(k-1)y(kT_s + \hat{\tau}) - \hat{a}_1(k)y((k-1)T_s + \hat{\tau})$

Non-data-aided TED uses received samples without any knowledge of the transmitted signal or the results of the channel estimation. Non-data-aided TED is used to estimate the timing error for signals with modulation schemes that have constellation points aligned with the in-phase or quadrature axis. Examples of signals suitable for the Gardner or early-late methods include QPSK-modulated signals with a zero phase offset that has points at {1+0*i*, 0+1*i*, -1+0*i*, 0−1*i*} and BPSK-modulated signals with a zero phase offset.

- **Gardner method** — The Gardner method is a non-data-aided feedback method that is independent of carrier phase recovery. It is used for baseband systems and modulated carrier systems. More specifically, this method is used for systems that use a linear modulation type with Nyquist pulses that have an excess bandwidth between approximately 40% and 100%. Examples include systems that use PAM, PSK, QAM, or OQPSK modulation and that shape the signal using raised cosine filters whose rolloff factor is between 0.4 and 1. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth increases (or rolloff factor increases in the case of a raised cosine filter). The Gardner method is similar to the early-late gate method.

- **Early-late method** — The early-late method is a non-data-aided feedback method. It is used for systems that use a linear modulation type such as PAM, PSK, QAM, or OQPSK modulation. For example, systems using a raised cosine filter with Nyquist pulses. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth of the pulse increases (or rolloff factor increases in the case of a raised cosine filter).

The early-late method is similar to the Gardner method. The Gardner method performs better in systems with high SNR values because it has lower self noise than the early-late method.

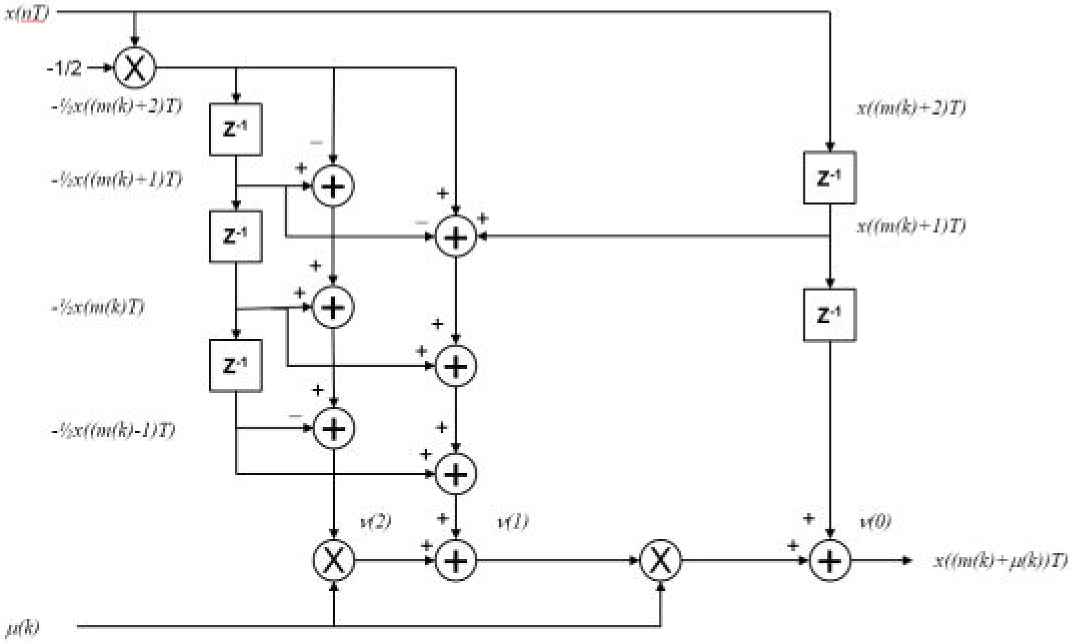
Decision-directed TED uses the `sign` function to estimate the in-phase and quadrature components of received samples, which result in lower computational complexity than non-data-aided TED.

- **Zero-crossing method** — The zero-crossing method is a decision-directed technique that requires 2 samples per symbol at the input to the synchronizer. It is used in low-SNR conditions for all values of excess bandwidth and in moderate-SNR conditions for moderate excess bandwidth factors in the approximate range [0.4, 0.6].
- **Mueller-Muller method** — The Mueller-Muller method is a decision-directed feedback method that requires prior recovery of the carrier phase. When the input signal has Nyquist pulses (for example, when using a raised cosine filter), the Mueller-Muller method has no self noise. For narrowband signaling in the presence of noise, the performance of the Mueller-Muller method improves as the excess bandwidth factor of the pulse decreases.

Because the decision-directed methods (zero-crossing and Mueller-Muller) estimate timing error based on the sign of the in-phase and quadrature components of signals passed to the synchronizer, they are not recommended for constellations that have points with either a zero in-phase or a quadrature component. $x(kT_S + \hat{\tau})$ and $y(kT_S + \hat{\tau})$ are the in-phase and quadrature components of the input signals to the timing error detector, where $\hat{\tau}$ is the estimated timing error. The Mueller-Muller method coefficients $\hat{a}_0(k)$ and $\hat{a}_1(k)$ are the estimates of $x(kT_S + \hat{\tau})$ and $y(kT_S + \hat{\tau})$. The timing estimates are made by applying the `sign` function to the in-phase and quadrature components and are used for only the decision-directed TED methods.

Interpolator

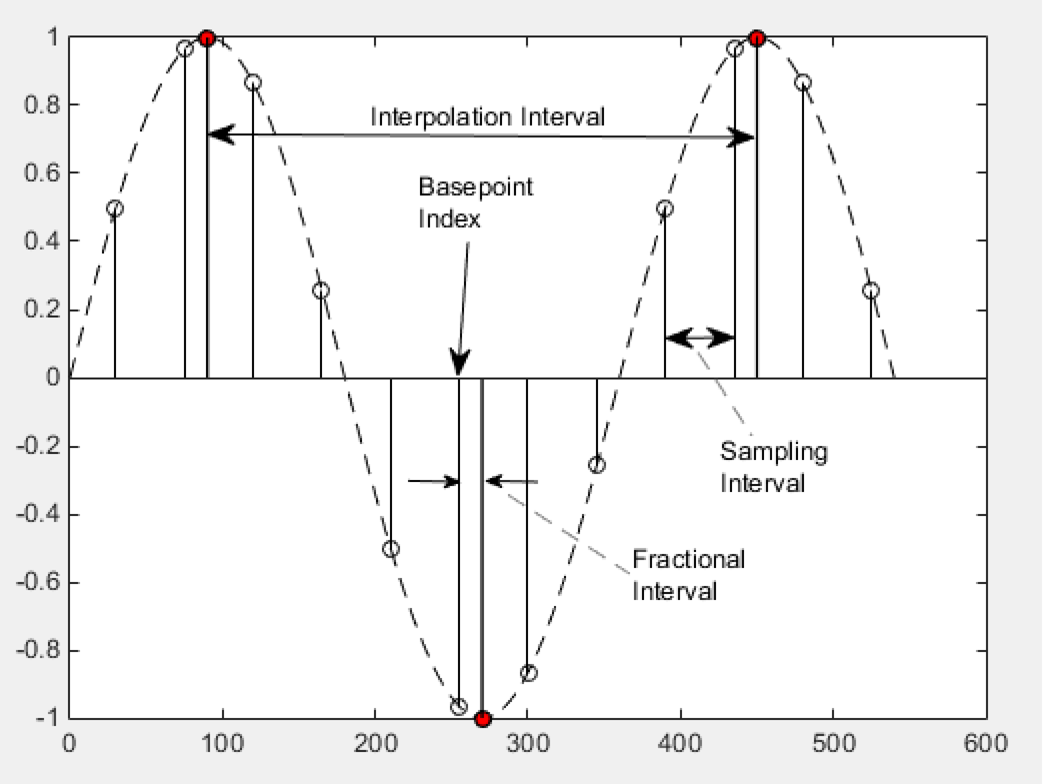
The time delay is estimated from the fixed-rate samples of the matched filter, which are asynchronous with the symbol rate. Because the resulting samples are not aligned with the symbol boundaries, an interpolator is used to "move" the samples. Because the time delay is unknown, the interpolator must be adaptive. Moreover, because the interpolant is a linear combination of the available samples, it can be thought of as the output of a filter.



The interpolator uses a piecewise parabolic interpolator with a Farrow structure and coefficient *α* set to 1/2 (see Rice, Michael, *Digital Communications: A Discrete-Time Approach*).

Interpolation Control

Interpolation control provides the interpolator with the basepoint index and fractional interval. The basepoint index is the sample index nearest to the interpolant. The fractional interval is the ratio of the time between the interpolant and its basepoint index and the interpolation interval.



Interpolation is performed for every sample, and a strobe signal is used to determine if the interpolant is output. The synchronizer uses a modulo-1 counter interpolation control to provide the strobe and the fractional interval for use with the interpolator.

Loop Filter

The synchronizer uses a proportional-plus integrator (PI) loop filter. The proportional gain, K_1, and the integrator gain, K_2, are calculated by

$$K_1 = \frac{-4\zeta\theta}{\left(1 + 2\zeta\theta + \theta^2\right)K_p}$$

and

$$K_2 = \frac{-4\theta^2}{\left(1 + 2\zeta\theta + \theta^2\right)K_p} .$$

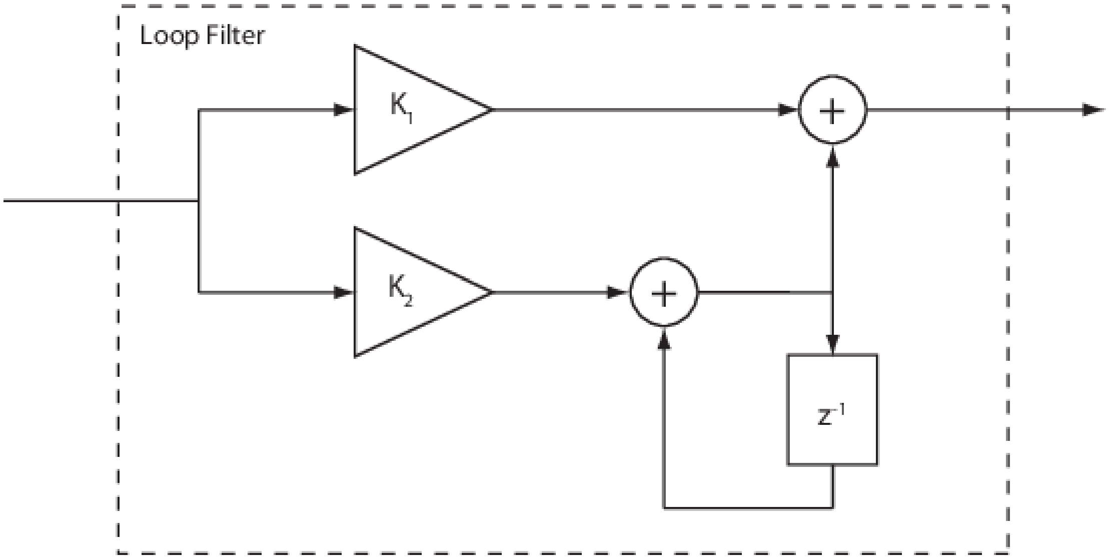
The interim term, θ, is given by

$$\theta = \frac{\frac{B_n T_s}{N}}{\zeta + \frac{1}{4\zeta}} ,$$

where:

- N is the number of samples per symbol.

- ζ is the damping factor.
- B_nT_s is the normalized loop bandwidth.
- K_p is the detector gain.



References

[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2008.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.CarrierSynchronizer`

Blocks

Symbol Synchronizer

Introduced in R2015a

comm.ThermalNoise

Package: `comm`

Add thermal noise to signal

Description

The `ThermalNoise` object simulates the effects of thermal noise on a complex, baseband signal.

To add thermal noise to a complex, baseband signal:

1. Define and set up your thermal noise object. See “Construction” on page 3-1399.
2. Call `step` to add thermal noise according to the properties of `comm.ThermalNoise`.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`tn = comm.ThermalNoise` creates a receiver thermal noise System object, `H`. This object adds thermal noise to the complex, baseband input signal.

`tn = comm.ThermalNoise(Name,Value)` creates a receiver thermal noise object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

Properties

`NoiseMethod` — Method used to set noise power

`'Noise temperature'` (default) | `'Noise figure'` | `'Noise factor'`

Method used to set the noise power, specified as `'Noise temperature'`, `'Noise figure'`, or `'Noise factor'`.

`NoiseTemperature` — Receiver noise temperature

`290` (default) | nonnegative real scalar

Receiver noise temperature, specified in degrees K as a nonnegative real scalar. This property is available when `NoiseMethod` is equal to `'Noise temperature'`. Noise temperature is typically used to characterize satellite receivers because the input noise temperature can vary and is often less than 290 K. Tunable.

`NoiseFigure` — Noise figure

`3.01` (default) | nonnegative real scalar

Noise figure, specified in dB as a nonnegative real scalar. This property is available when `NoiseMethod` is equal to `'Noise figure'`. Noise figure describes the performance of a receiver and does not include the effect of the antenna. It is defined only for an input noise temperature of 290 K. The noise figure is the dB equivalent of the noise factor. Tunable.

`NoiseFactor` — Noise factor

2 (default) | real scalar ≥ 1

Noise factor, specified as a real scalar greater than or equal to 1. This property is available when `NoiseMethod` is equal to `'Noise factor'`. Noise factor describes the performance of a receiver and does not include the effect of the antenna. It is defined only for an input noise temperature of 290 K. The noise factor is the linear equivalent of the noise figure. Tunable.

`SampleRate` — Sample rate

1 (default) | positive real scalar

Sample rate, specified as in Hz as a positive real scalar. The object computes the variance of the noise added to the input signal as kT×`SampleRate`. The value k is Boltzmann's constant and T is the noise temperature specified explicitly or implicitly via one of the noise methods.

`Add290KAntennaNoise` — Add 290 K antenna noise

`false` (default) | `true`

Add 290 K antenna noise to the input signal, specified as a logical scalar. To add 290 K antenna noise, set this property to `true`. This property is available when `NoiseMethod` is equal to `'Noise factor'` or `'Noise figure'`.

The total noise applied to the input signal is the sum of the circuit noise and the antenna noise.

Methods

step	Add receiver thermal noise

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Add Thermal Noise to QPSK Signal

Create a thermal noise object having a noise temperature of 290 K and a sample rate of 5 MHz.

```
thNoise = comm.ThermalNoise('NoiseTemperature',290,'SampleRate',5e6);
```

Generate QPSK-modulated data having an output power of 20 dBm.

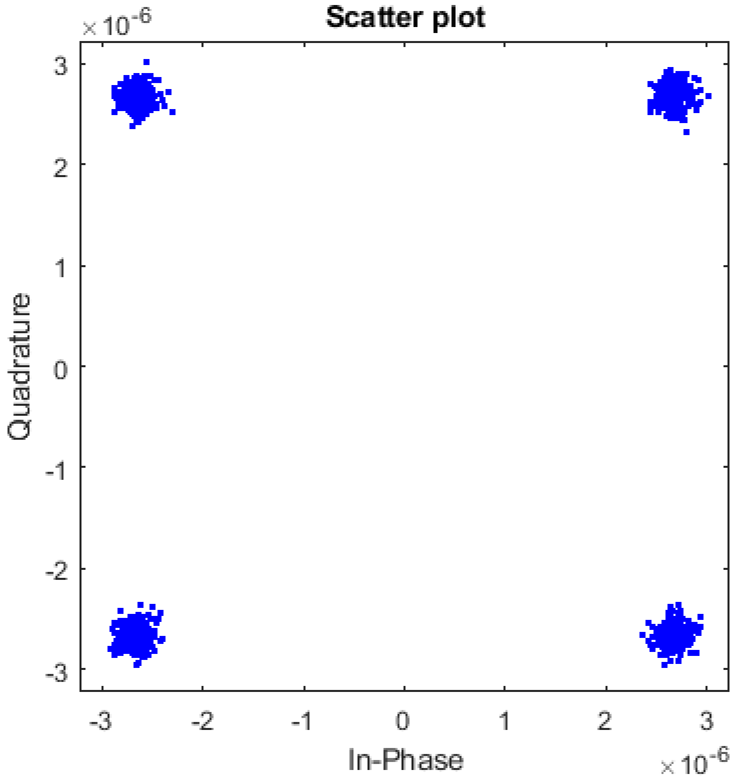
```
data = randi([0 3],1000,1);
modData = 0.3162*pskmod(data,4,pi/4);
```

Attenuate the signal by the free space path loss assuming a 1000 m link distance and a carrier frequency of 2 GHz.

```
fsl = (4*pi*1000*2e9/3e8)^2;
rxData = modData/sqrt(fsl);
```

Add thermal noise to the signal. Plot the noisy constellation.

```
noisyData = thNoise(rxData);
scatterplot(noisyData)
```



Add Antenna and Receiver Thermal Noise to 16-QAM Signal

Create a thermal noise object having a 5 dB noise figure and a 10 MHz sample rate. Specify that the 290 K antenna noise be included.

```
thermalNoise = comm.ThermalNoise('NoiseMethod','Noise figure', ...
    'NoiseFigure',5, ...
    'SampleRate',10e6, ...
    'Add290KAntennaNoise',true);
```

Generate QPSK-modulated data having a 1 W output power.

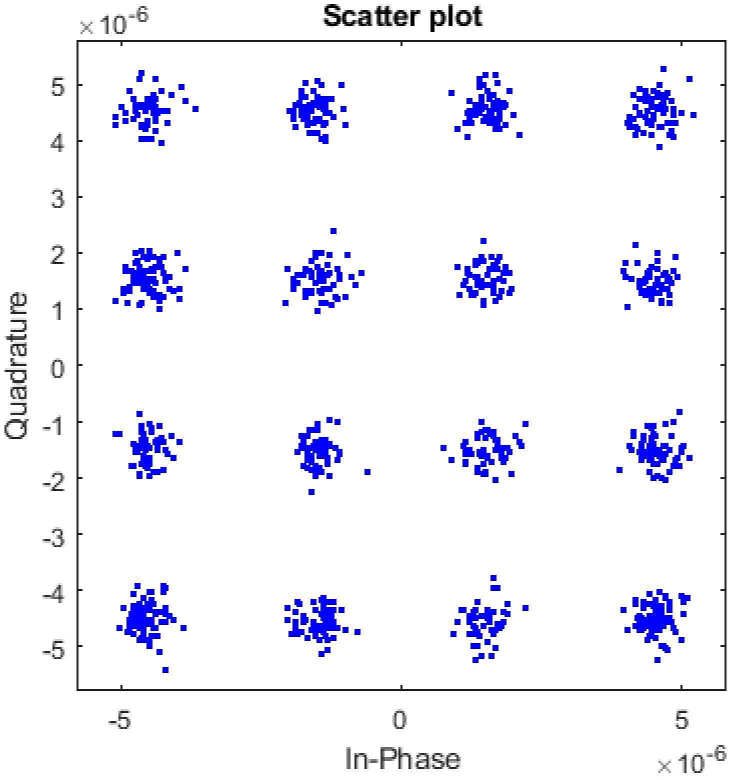
```
data = randi([0 15],1000,1);
modSig = qammod(data,16,'UnitAveragePower',true);
```

Attenuate the signal by the free space path loss assuming a 1 km link distance and a 5 GHz carrier frequency.

```
fsl = (4*pi*1000*5e9/3e8)^2;
rxSig = modSig/sqrt(fsl);
```

Add thermal noise to the signal and plot its constellation.

```
noisySig = thermalNoise(rxSig);
scatterplot(noisySig)
```



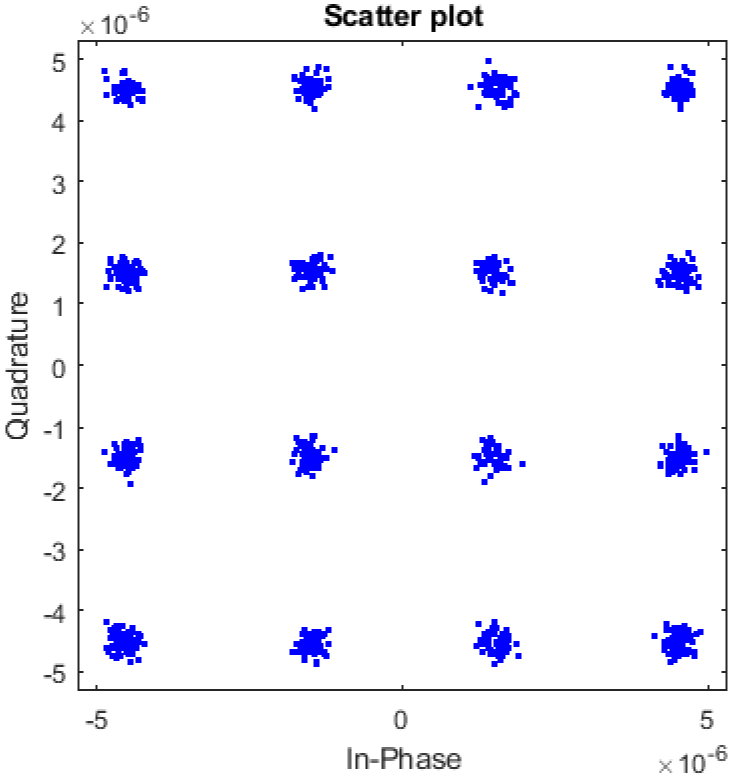
Estimate the SNR.

```
mer = comm.MER;
snrEst1 = mer(rxSig,noisySig)
```

```
snrEst1 = 22.6611
```

Decrease the noise figure to 0 dB, and plot the resultant received signal. Because antenna noise is included, the signal is not completely noiseless.

```
thermalNoise.NoiseFigure = 0;
noisySig = thermalNoise(rxSig);
scatterplot(noisySig)
```



Estimate the SNR. The SNR is 5 dB higher than in the first case, which is expected given the 5 dB decrease in the noise figure.

```
snrEst2 = mer(rxSig,noisySig)
```

```
snrEst2 = 27.8658
```

```
snrEst2 - snrEst1
```

```
ans = 5.2047
```

Algorithms

Wireless receiver performance is often expressed as a noise factor or figure. The noise factor is defined as the ratio of the input signal-to-noise ratio, S_i/N_i to the output signal-to-noise ratio, S_o/N_o, such that

$$F = \frac{S_i/N_i}{S_o/N_o} \; .$$

Given receiver gain G and receiver noise power N_{ckt}, the noise factor can be expressed as

$$\begin{aligned} F &= \frac{S_i/N_i}{GS_i/(N_{ckt} + GN_i)} \\ &= \frac{N_{ckt} + GN_i}{GN_i} \; . \end{aligned}$$

The IEEE defines the noise factor assuming that noise temperature at the input is T_0, where T_0 = 290 K. The noise factor is then

$$\begin{aligned} F &= \frac{N_{ckt} + GN_i}{GN_i} \\ &= \frac{GkBT_{ckt} + GkBT_0}{GkBT_0} \\ &= \frac{T_{ckt} + T_0}{T_0} \ . \end{aligned}$$

T_{ckt} is the equivalent input noise temperature of the receiver and is expressed as

$$T_{ckt} = T_0(F - 1) \ .$$

The overall noise temperature of an antenna and receiver, T_{sys}, is

$$T_{sys} = T_{ant} + T_{ckt} \, ,$$

where T_{ant} is the antenna noise temperature.

The noise figure, NF, is the dB equivalent of the noise factor and can be expressed as

$$NF = 10\log_{10}(F) \ .$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.AWGNChannel`

Introduced in R2012a

step

System object: `comm.ThermalNoise`
Package: `comm`

Add receiver thermal noise

Syntax

```
Y = step(H,X)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` adds thermal noise to the complex, baseband input signal, `X`, and outputs the result in `Y`. The input signal `X` must be a complex, double or single precision data type column vector or scalar.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.TurboDecoder

Package: `comm`

Decode input signal using parallel concatenated decoding scheme

Description

The `comm.TurboDecoder` System object uses a parallel concatenated decoding scheme to decode a coded input signal. The input signal is typically the soft-decision output from the baseband demodulation operation. For more information, see “Parallel Concatenated Convolutional Decoding Scheme” on page 3-1413.

To decode an input signal using a parallel concatenated decoding scheme:

1. Create the `comm.TurboDecoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
turbodec = comm.TurboDecoder
turbodec = comm.TurboDecoder(trellis,interlvrindices,numiter)
turbodec = comm.TurboDecoder(___,Name,Value)
```

Description

`turbodec = comm.TurboDecoder` creates a turbo decoder System object. This object uses the *a-posteriori* probability (APP) constituent decoder to iteratively decode the parallel-concatenated convolutionally encoded input data.

`turbodec = comm.TurboDecoder(trellis,interlvrindices,numiter)` creates a turbo decoder System object with the `TrellisStructure`, `InterleaverIndices`, and `numiter`, respectively. The `trellis` input must be specified as described by the `TrellisStructure` property. The `interlvrindices` input must be specified as described by the `InterleaverIndices` property. The `numiter` input must be specified as described by the `NumIterations` property.

`turbodec = comm.TurboDecoder(___,Name,Value)` sets properties using one or more name-value pairs in addition to any input argument combination from previous syntaxes. Enclose each property name in quotes. For example, `comm.TurboDecoder('InterleaverIndicesSource','Input port')` configures a turbo decoder System object with the interleaver indices to be supplied as an input argument to the System object when it is called.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`TrellisStructure` — Trellis description of constituent convolutional code

`poly2trellis(4,[13 15],13)` (default) | structure

Trellis description of the constituent convolutional code, specified as a structure that contains the trellis description for a rate K/N code. K is the number of input bit streams, and N is the number of output bit streams.

Note K must be 1 for the turbo coder. For more information, see “Coding Rate” on page 3-1414.

You can either use the `poly2trellis` function to create the trellis structure or create it manually. For more about this structure, see “Trellis Description of a Convolutional Code” and the `istrellis` function.

The trellis structure contains these fields.

`numInputSymbols` — Number of symbols input to encoder

2^K

Number of symbols input to the encoder, specified as an integer equal to 2^K, where K is the number of input bit streams.

Data Types: `double`

`numOutputSymbols` — Number of symbols output from encoder

2^N

Number of symbols output from the encoder, specified as an integer equal to 2^N, where N is the number of output bit streams.

Data Types: `double`

`numStates` — Number of states in encoder

power of 2

Number of states in the encoder, specified as a power of 2.

Data Types: `double`

`nextStates` — Next states

matrix of integers

Next states for all combinations of current states and current inputs, specified as a matrix of integers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

`outputs` — Outputs

matrix of octal numbers

Outputs for all combinations of current states and current inputs, specified as a matrix of octal numbers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

Data Types: `struct`

`InterleaverIndicesSource` — Source of interleaver indices

`'Property'` (default) | `'Input port'`

Source of interleaver indices, specified as `'Property'` or `'Input port'`.

- When you set this property to `'Input port'`, the object executes using the input interleaver indices, `interlvrindices`, set when you call the object. The vector length and values for the interleaver indices and coded input signal can change with each call to the object.
- When you set this property to `'Property'`, the object uses the interleaver indices that you specify for the `InterleaverIndices` property.

Data Types: `char` | `string`

`InterleaverIndices` — Interleaver indices

`(64:-1:1).'` (default) | column vector of integers

Interleaver indices, specified as a column vector of integers. The vector must be of length L, where L is the length of the decoded output message, `decmsg`. Each element of the vector must be an integer in the range [1, L] and must be unique. The interleaver indices define the mapping used to permute the input bits at the decoder.

Tunable: Yes

Dependencies

To enable this property, set the `InterleaverIndicesSource` property to `'Property'`.

Data Types: `double`

`Algorithm` — Decoding algorithm

`'True APP'` (default) | `'Max*'` | `'Max'`

Decoding algorithm, specified as `'True APP'`, `'Max*'`, or `'Max'`. When you set this property to `'True APP'`, the object implements true APP decoding. When you set this property to `'Max*'` or `'Max'`, the object uses approximations to increase the speed of the computations. For more information, see “APP Decoder” on page 3-1414.

Data Types: `char` | `string`

`NumScalingBits` — Number of scaling bits

3 (default) | integer in the range [0, 8]

Number of scaling bits, specified as an integer in the range [0, 8]. This property sets the number of bits the constituent decoders use to scale the input data to avoid losing precision during computations. The constituent decoders multiply the input by $2^{\texttt{NumScalingBits}}$ and divide the pre-output by the same factor. For more information, see “APP Decoder” on page 3-1414.

Dependencies

This enable this property, set the Algorithm property to `'Max*'`.

Data Types: `double`

`NumIterations` — Number of decoding iterations

`6` (default) | positive integer

Number of decoding iterations, specified as a positive integer. This property sets the number of decoding iterations used for each call to the object. The object iterates and provides updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the object is the hard-decision output of the final LLR update.

Data Types: `double`

Usage

Syntax

```
decmsg = turbodec(codeword)
decmsg = turbodec(codeword,interlvrindices)
```

Description

`decmsg = turbodec(codeword)` decodes the input codeword using the parallel concatenated convolutional decoding scheme that is specified by the trellis structure and interleaver indices. `turbodec` returns the binary decoded data. For more information, see “Parallel Concatenated Convolutional Decoding Scheme” on page 3-1413.

`decmsg = turbodec(codeword,interlvrindices)` additionally specifies the interleaver indices. To enable this syntax, set the InterleaverIndicesSource property to `'Input port'`. The interleaver indices define the mapping used to permute the input at the decoder.

Input Arguments

`codeword` — Parallel concatenated codeword

column vector

Parallel concatenated codeword, specified as a column vector of length M, where M is the length of the parallel concatenated codeword.

Data Types: `double` | `single`

`interlvrindices` — Interleaver indices

column vector of integers

Interleaver indices, specified as a column vector of integers. The vector must be of length L, where L is the length of the decoded output message, `decmsg`. Each element of the vector must be an integer in the range [1, L] and must be unique. The interleaver indices define the mapping used to permute the input bits at the decoder.

Dependencies

To enable this property, set the `InterleaverIndicesSource` property to `'Input port'`.

Data Types: `double`

Output Arguments

`decmsg` — Decoded message
binary column vector

Decoded message, returned as a binary column vector of length *L*, where *L* is the length of the decoded output message. This output signal is the same as data type of the `codeword` input.

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Transmit and Receive Turbo-Encoded Data over BPSK-Modulated AWGN Channel

Simulate the transmission and reception of BPSK data over an AWGN channel by using turbo encoding and decoding.

Specify simulation parameters, and then compute the effective coding rate and noise variance. For BPSK modulation, E_S/N_0 equals E_b/N_0 because the number of bits per symbol (bps) is 1. To ease reuse of this code for other modulation schemes, calculations in this example include the bps terms. Define the packet length, trellis structure, and number of iterations. Calculate the noise variance using E_S/N_0 and the code rate. Set the random number generator to its default state to ensure that the results are repeatable.

```
modOrd = 2; % Modulation order
bps = log2(modOrd); % Bits per symbol
EbNo = 1; % Energy per bit to noise power spectral density ratio in dB
EsNo = EbNo + 10*log10(bps); % Energy per symbol to noise power spectral density ratio in dB

L = 256; % Input packet length in bits
trellis = poly2trellis(4,[13 15 17],13);
numiter = 4;
n = log2(trellis.numOutputSymbols);
numTails = log2(trellis.numStates)*n;
M = L*(2*n - 1) + 2*numTails; % Output codeword packet length
rate = L/M; % Coding rate

snrdB = EsNo + 10*log10(rate); % Signal to noise ratio in dB
noiseVar = 1./(10.^(snrdB/10)); % Noise variance
```

```
rng default
```

Generate random interleaver indices.

```
intrlvrIndices = randperm(L);
```

Create a turbo encoder and decoder pair. Use the defined trellis structure and random interleaver indices. Configure the decoder to run a maximum of four iterations.

```
turboenc = comm.TurboEncoder(trellis,intrlvrIndices);
turbodec = comm.TurboDecoder(trellis,intrlvrIndices,numiter);
```

Create a BPSK modulator and demodulator pair, where the demodulator outputs soft bits determined using an LLR method.

```
bpskmod = comm.BPSKModulator;
bpskdemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
awgnchan = comm.AWGNChannel('NoiseMethod','Variance','Variance',noiseVar);
errrate = comm.ErrorRate;
```

The main processing loop performs these steps.

1. Generate binary data.
2. Turbo encode the data.
3. Modulate the encoded data.
4. Pass the modulated signal through an AWGN channel.
5. Demodulate the noisy signal by using LLR to output soft bits.
6. Turbo decode the demodulated data. Because the bit mapping from the demodulator is opposite of the mapping expected by the turbo decoder, the decoder input must use the inverse of the demodulated signal.
7. Calculate the error statistics.

```
for frmIdx = 1:100
    data = randi([0 1],L,1);
    encodedData = turboenc(data);
    modSignal = bpskmod(encodedData);
    receivedSignal = awgnchan(modSignal);
    demodSignal = bpskdemod(receivedSignal);
    receivedBits = turbodec(-demodSignal);
    errorStats = errrate(data,receivedBits);
end
```

Display the error data.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', errorStats)
```

```
Bit error rate = 2.34e-04
Number of errors = 6
Total bits = 25600
```

Perform Turbo Coding on 16-QAM Signal in AWGN Channel

Simulate an end-to-end communication link by using a 16-QAM signal and turbo codes in an AWGN channel. Inside a frame processing loop, packet sizes are randomly selected to be 500, 1000, or 1500 bits. Because the packet size varies, the interleaver indices are provided to the turbo encoder and decoder as an input argument of their associated System object.

Set the modulation order and E_b/N_0. Compute the number of bits per symbol and the energy per symbol to noise ratio (E_S/N_0) based on the modulation order and E_b/N_0. To get repeatable results, set the random number generator to its default state.

```
modOrder = 16;
bps = log2(modOrder); % Bits per symbol
EbNo = 2.8; % Energy per bit to noise power spectral density ratio in dB
EsNo = EbNo + 10*log10(bps); % Energy per symbol to noise power spectral density ratio in dB
rng default
```

Create a turbo encoder and decoder pair. Because the packet length varies for each frame, specify that the interleaver indices be supplied by an input argument of the System object when executed. Specify that the decoder perform four iterations.

```
turboEnc = comm.TurboEncoder('InterleaverIndicesSource','Input port');
turboDec = comm.TurboDecoder('InterleaverIndicesSource','Input port','NumIterations',4);
trellis = poly2trellis(4,[13 15 17],13);
n = log2(turboEnc.TrellisStructure.numOutputSymbols);
numTails = log2(turboEnc.TrellisStructure.numStates)*n;
```

Create an error rate object.

```
errRate = comm.ErrorRate;
```

The frame processing loop performs these steps.

1. Select a random packet length, and generate random binary data.
2. Compute the output codeword length and coding rate.
3. Compute the signal to noise ratio (SNR) and noise variance.
4. Generate interleaver indices.
5. Turbo encode the data.
6. Apply 16-QAM modulation, and normalize the average signal power.
7. Pass the modulated signal through an AWGN channel.
8. Demodulate the noisy signal by using an LLR method, output soft bits, and normalize the average signal power.
9. Turbo decode the data. Because the bit mapping order from the demodulator is opposite the mapping order expected by the turbo decoder, the decoder input must use the inverse of the demodulated signal.
10. Calculate the error statistics.

```
numFrames = 100;
for pktIdx = 1:numFrames
    L = 500*randi([1 3],1,1); % Packet length in bits
    data = randi([0 1],L,1);
    M = L*(2*n - 1) + 2*numTails; % Output codeword packet length
    rate = L/M; % Coding rate for current packet
    snrdB = EsNo + 10*log10(rate); % Signal to noise ratio in dB
    noiseVar = 1./(10.^(snrdB/10)); % Noise variance

    intrlvrIndices = randperm(L);
    encodedData = turboEnc(data,intrlvrIndices);
    modSignal = qammod(encodedData,modOrder,'InputType','bit','UnitAveragePower',true);
    receivedSignal = awgn(modSignal,snrdB);
    demodSignal = qamdemod(receivedSignal,modOrder,'OutputType','llr', ...
        'UnitAveragePower',true,'NoiseVariance',noiseVar);
    receivedBits = turboDec(-demodSignal,intrlvrIndices); % Demodulated signal is negated

    errorStats = errRate(data,receivedBits);
end
```

Display the error data.

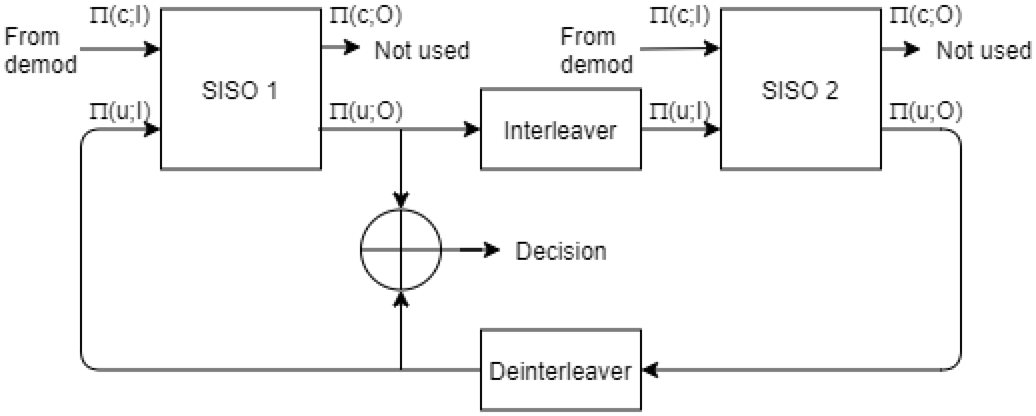
```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n',errorStats)
```

```
Bit error rate = 8.51e-04
Number of errors = 80
Total bits = 94000
```

More About

Parallel Concatenated Convolutional Decoding Scheme

The turbo decoder uses a parallel concatenated convolutional decoding scheme to decode a coded input signal. The parallel concatenated decoding scheme uses an iterative “APP Decoder” on page 3-1414 with two constituent decoders, an interleaver, and a deinterleaver. This figure shows the decoding scheme.



The two constituent decoders use the same trellis structure and decoding algorithm. The soft-input soft-output APP decoders (SISO 1 and SISO 2) output an updated sequence of log-likelihoods of the encoder input bits, π(u;O).The sequence is based on the received sequence of log-likelihoods of the channel (coded) bits, π(c;I), and code parameters.

The decoder iteratively updates these likelihoods for a fixed number of decoding iterations and then outputs the decision bits. The interleaver used in the decoder is identical to the interleaver used in the encoder. The deinterleaver performs the inverse permutation with respect to the interleaver. The decoder does not assume knowledge of the tail bits and excludes these bits from the iterations.

For more information, see “Coding Rate” on page 3-1414.

APP Decoder

This System object implements the soft-input-soft-output APP decoding algorithm according to [1] and [2].

The `'True APP'` option of the `Algorithm` property implements APP decoding as per the equations 20–23 in section V of [1]. To gain speed, the `'Max*'` and `'Max'` values of the `Algorithm` property approximate expressions like $\log \sum_i \exp(a_i)$ by other quantities. The `'Max'` option uses $\max(a_i)$ as the approximation. The `'Max*'` option uses $\max(a_i)$ plus a correction term given by the expression $\ln(1 + \exp(-|a_{i-1} - a_i|))$.

Setting the `Algorithm` property to `'Max*'` enables the `NumScalingBits` property of this System object. This property denotes the number of bits by which this System object scales the data it processes internally (multiplies the input by $2^{\texttt{NumScalingBits}}$ and divides the pre-output by the same factor). Use this property to avoid losing precision during computations.

Coding Rate

Each constituent convolutional coder has a 1/*N* coding rate. The coding rate of the constituent convolutional code is represented as a rate *K*/*N* code. *K* is the number of input bit streams. *N* is the number of output bit streams.

Note *K* must be 1 for the turbo coder.

The decoder accepts an *M*-element column vector input signal and returns an *L*-element column vector output signal. The effective code rate of the turbo decoder is *L*/*M*. *L* is the length of the binary output message, and *M* is the length of the parallel concatenated codeword.

For a given trellis, *M* and *L* are related by $L = (M - 2 \times numTails) / (2 \times N - 1)$, where:

- $numTails = \log_2($`trellis.numStates`$) \times N$
- $N = \log_2($`trellis.numOutputSymbols`$)$. For a rate 1/2 trellis, $N = 2$.

For more information about trellis structures, see `poly2trellis` function.

References

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

[2] Viterbi, A.J. “An Intuitive Justification and a Simplified Implementation of the MAP Decoder for Convolutional Codes.” *IEEE Journal on Selected Areas in Communications* 16, no. 2 (February 1998): 260–64. https://doi.org/10.1109/49.661114.

[3] Berrou, C., A. Glavieux, and P. Thitimajshima. “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes.” *Proceedings of ICC 93 - IEEE International Conference on*

Communications, Geneva, Switzerland, May 1993, 1064–70. https://doi.org/10.1109/icc.1993.397441.

[4] Schlegel, Christian, and Lance Perez. *Trellis and Turbo Coding*. IEEE Press Series on Digital & Mobile Communication. Piscataway, NJ ; Hoboken, NJ: IEEE Press ; Wiley-Interscience, 2004.

[5] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.APPDecoder` | `comm.TurboEncoder` | `comm.ViterbiDecoder`

Functions

`istrellis` | `poly2trellis` | `vitdec`

Blocks

Turbo Decoder

Introduced in R2012a

comm.TurboEncoder

Package: `comm`

Encode input signal using parallel concatenated encoding scheme

Description

The `comm.TurboEncoder` System object applies a parallel concatenated encoding scheme to a binary input message. This coding scheme uses two identical convolutional encoders and appends the termination bits at the end of the encoded data bit stream. For more information, see “Parallel Concatenated Convolutional Encoding Scheme” on page 3-1424.

To encode a binary input message using a parallel concatenated encoding scheme:

1. Create the `comm.TurboEncoder` object and set its properties.
2. Call the object with arguments, as if it were a function.

To learn more about how System objects work, see What Are System Objects?.

Creation

Syntax

```
turboenc = comm.TurboEncoder
turboenc = comm.TurboEncoder(Name,Value)
turboenc = comm.TurboEncoder(trellis,interlvrindices)
```

Description

`turboenc = comm.TurboEncoder` creates a turbo encoder System object. This object performs turbo encoding using the default object configuration.

`turboenc = comm.TurboEncoder(Name,Value)` sets properties using one or more name-value pairs. For example, `comm.TurboEncoder('InterleaverIndicesSource','Input port')` configures a turbo encoder System object with the interleaver indices to be supplied as an input argument to the System object when it is called. Enclose each property name in quotes.

`turboenc = comm.TurboEncoder(trellis,interlvrindices)` creates a turbo encoder System object with the `TrellisStructure` and `InterleaverIndices` properties set to `trellis` and `interlvrindices`, respectively. The `trellis` input must be specified as described by the `TrellisStructure` property. The `interlvrindices` input must be specified as described by the `InterleaverIndices` property.

Properties

Unless otherwise indicated, properties are *nontunable*, which means you cannot change their values after calling the object. Objects lock when you call them, and the `release` function unlocks them.

If a property is *tunable*, you can change its value at any time.

For more information on changing property values, see System Design in MATLAB Using System Objects.

`TrellisStructure` — Trellis description of constituent convolutional code
`poly2trellis(4,[13 15],13)` (default) | structure

Trellis description of the constituent convolutional code, specified as a structure that contains the trellis description for a rate K/N code. K is the number of input bit streams, and N is the number of output bit streams.

Note K must be 1 for the turbo coder. For more information, see “Coding Rate” on page 3-1424.

You can either use the `poly2trellis` function to create the trellis structure or create it manually. For more about this structure, see “Trellis Description of a Convolutional Code” and the `istrellis` function.

The trellis structure contains these fields.

`numInputSymbols` — Number of symbols input to encoder
2^K

Number of symbols input to the encoder, specified as an integer equal to 2^K, where K is the number of input bit streams.

Data Types: `double`

`numOutputSymbols` — Number of symbols output from encoder
2^N

Number of symbols output from the encoder, specified as an integer equal to 2^N, where N is the number of output bit streams.

Data Types: `double`

`numStates` — Number of states in encoder
power of 2

Number of states in the encoder, specified as a power of 2.

Data Types: `double`

`nextStates` — Next states
matrix of integers

Next states for all combinations of current states and current inputs, specified as a matrix of integers. The matrix size must be `numStates`-by-2^K.

Data Types: `double`

`outputs` — Outputs
matrix of octal numbers

Outputs for all combinations of current states and current inputs, specified as a matrix of octal numbers. The matrix size must be `numStates`-by-2^K.

Data Types: double

Data Types: struct

InterleaverIndicesSource — Source of interleaver indices
'Property' (default) | 'Input port'

Source of interleaver indices, specified as 'Property' or 'Input port'.

- When you set this property to 'Input port', the object executes using the input interleaver indices, set when you call the object. The vector length and values for the interleaver indices and binary input message can change with each call to the object.
- When you set this property to 'Property', the object uses the interleaver indices that you specify for the InterleaverIndices property.

Data Types: char | string

InterleaverIndices — Interleaver indices
(64:-1:1).' (default) | column vector of integers

Interleaver indices, specified as a column vector of integers. The vector must be of length *L*, where *L* is the length of the binary input message. Each element of the vector must be an integer in the range [1, *L*] and must be unique. The interleaver indices define the mapping used to permute the input bits at the encoder.

Tunable: Yes

Dependencies

To enable this property, set the InterleaverIndicesSource property to 'Property'.

Data Types: double

Usage

Syntax

```
codeword = turboenc(message)
codeword = turboenc(message,interlvrindices)
```

Description

codeword = turboenc(message) encodes the input message using the parallel concatenated convolutional encoding scheme specified by the trellis structure and interleaver indices. turboenc returns the binary encoded codeword. message and codeword are column vectors of numeric, logical, or unsigned fixed-point values with word length 1 (fi object). For more information, see “Parallel Concatenated Convolutional Encoding Scheme” on page 3-1424.

codeword = turboenc(message,interlvrindices) additionally specifies the interleaver indices. interlvrindices must be a column vector containing integers in the range [1, *L*] with no repeated values. *L* is the length of the binary input message, message. This syntax applies when the InterleaverIndicesSource property is set to 'Input port'. The interleaver indices define the mapping used to permute the input bits at the encoder.

Input Arguments

`message` — Input message
binary column vector

Input message, specified as a binary column vector of length *L*, where *L* is the length of the uncoded input message.

Data Types: `double` | `int8` | `fi(data,0,1)`

Output Arguments

`codeword` — Parallel concatenated codeword
binary column vector

Parallel concatenated codeword, returned as a binary column vector of length *M*, where *M* is the number of bits in the parallel concatenated codeword. This output inherits its data type from the `message` input.

Data Types: `double` | `int8` | `fi(data,0,1)`

Object Functions

To use an object function, specify the System object as the first input argument. For example, to release system resources of a System object named `obj`, use this syntax:

```
release(obj)
```

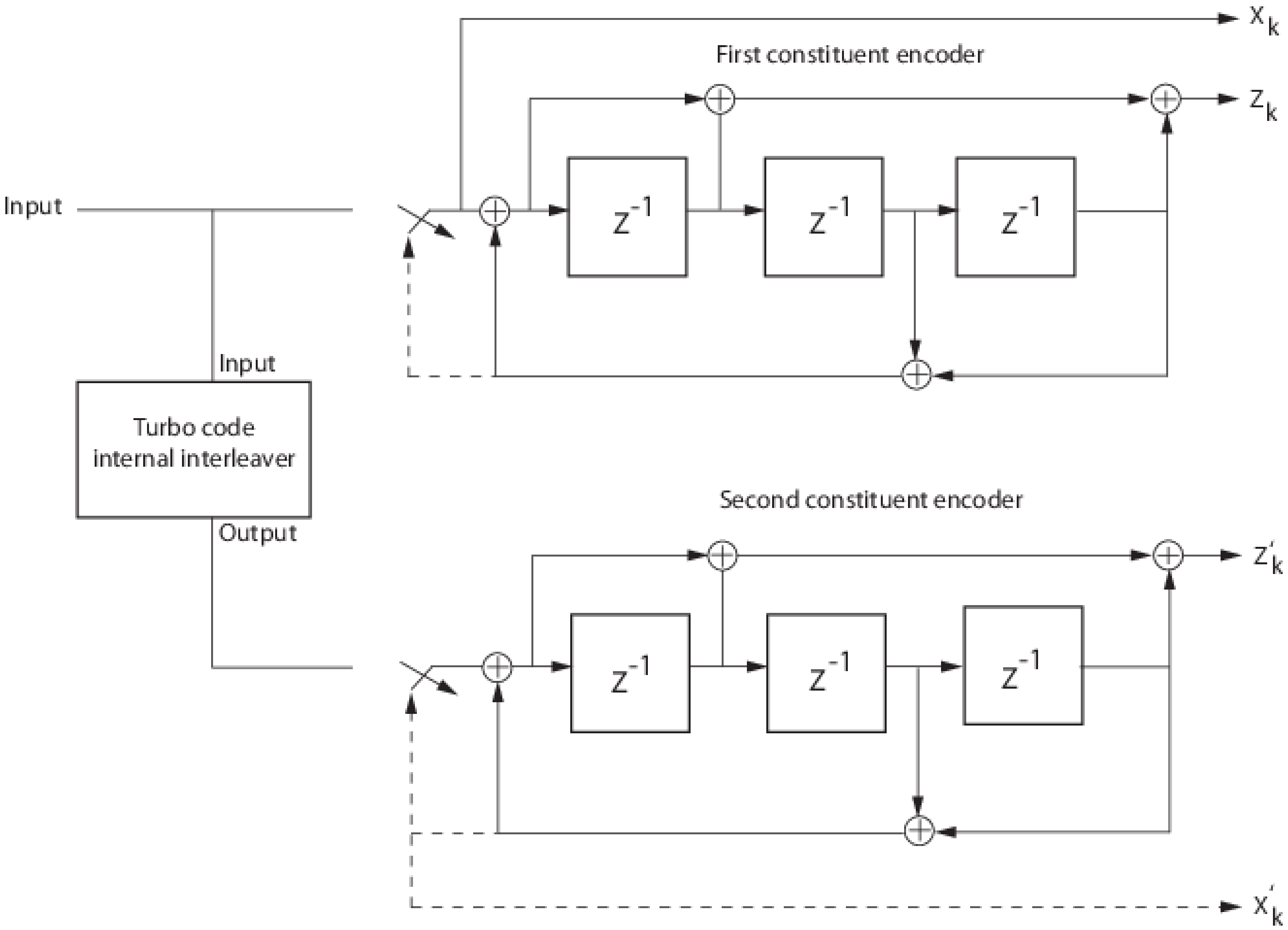
Common to All System Objects

step	Run System object algorithm
release	Release resources and allow changes to System object property values and input characteristics
reset	Reset internal states of System object

Examples

Perform Rate 1/3 Turbo Code Encoding

Encode an input message by using a rate 1/3 turbo encoder configuration with the default trellis structure, `poly2trellis(4,[13 15],13)`, as represented in this figure.


Input
Input
Turbo code internal interleaver
Output
First constituent encoder
Second constituent encoder
z^{-1}
X_k
Z_k
Z'_k
X'_k

For an input message with 64 bits, the codeword output from the encoder is 204 bits. The first 192 bits output correspond to the three 64 bit streams, interlaced as X_k, Z_k, and Z'_k. The systematic bit stream, X_k, and parity bit stream, Z_k, are from the first encoder and the parity bit stream, Z'_k, is from the second encoder. When the switches are in the lower position, signals follow the dashed lines and the last 12 bits correspond to the tail bits from the two encoders. The first group of six bits (three systematic bits and three parity bits) are the output tail bits from the first constituent encoder. The second group of six bits (three systematic bits and three parity bits) are the output tail bits from the second constituent encoder.

Create a turbo encoder using the default settings. Generate a frame of binary message data, and then encode the message data.

```
rng default
turboenc = comm.TurboEncoder;
frameLen = 64; % Frame length
data = randi([0 1],frameLen,1);
encData = turboenc(data);
codewordLen = length(encData);
```

Compute the coding rate. Due to the tail bits, the encoder output code rate is slightly less than 1/3.

```
codingrate = frameLen/codewordLen
codingrate = 0.3137
```

Transmit and Receive Turbo-Encoded Data over BPSK-Modulated AWGN Channel

Simulate the transmission and reception of BPSK data over an AWGN channel by using turbo encoding and decoding.

Specify simulation parameters, and then compute the effective coding rate and noise variance. For BPSK modulation, E_S/N_0 equals E_b/N_0 because the number of bits per symbol (bps) is 1. To ease reuse of this code for other modulation schemes, calculations in this example include the bps terms. Define the packet length, trellis structure, and number of iterations. Calculate the noise variance using E_S/N_0 and the code rate. Set the random number generator to its default state to ensure that the results are repeatable.

```
modOrd = 2; % Modulation order
bps = log2(modOrd); % Bits per symbol
EbNo = 1; % Energy per bit to noise power spectral density ratio in dB
EsNo = EbNo + 10*log10(bps); % Energy per symbol to noise power spectral density ratio in dB

L = 256; % Input packet length in bits
trellis = poly2trellis(4,[13 15 17],13);
numiter = 4;
n = log2(trellis.numOutputSymbols);
numTails = log2(trellis.numStates)*n;
M = L*(2*n - 1) + 2*numTails; % Output codeword packet length
rate = L/M; % Coding rate

snrdB = EsNo + 10*log10(rate); % Signal to noise ratio in dB
noiseVar = 1./(10.^(snrdB/10)); % Noise variance

rng default
```

Generate random interleaver indices.

```
intrlvrIndices = randperm(L);
```

Create a turbo encoder and decoder pair. Use the defined trellis structure and random interleaver indices. Configure the decoder to run a maximum of four iterations.

```
turboenc = comm.TurboEncoder(trellis,intrlvrIndices);
turbodec = comm.TurboDecoder(trellis,intrlvrIndices,numiter);
```

Create a BPSK modulator and demodulator pair, where the demodulator outputs soft bits determined using an LLR method.

```
bpskmod = comm.BPSKModulator;
bpskdemod = comm.BPSKDemodulator('DecisionMethod','Log-likelihood ratio', ...
    'Variance',noiseVar);
```

Create an AWGN channel object and an error rate object.

```
awgnchan = comm.AWGNChannel('NoiseMethod','Variance','Variance',noiseVar);
errrate = comm.ErrorRate;
```

The main processing loop performs these steps.

1. Generate binary data.
2. Turbo encode the data.
3. Modulate the encoded data.
4. Pass the modulated signal through an AWGN channel.
5. Demodulate the noisy signal by using LLR to output soft bits.
6. Turbo decode the demodulated data. Because the bit mapping from the demodulator is opposite of the mapping expected by the turbo decoder, the decoder input must use the inverse of the demodulated signal.
7. Calculate the error statistics.

```
for frmIdx = 1:100
    data = randi([0 1],L,1);
    encodedData = turboenc(data);
    modSignal = bpskmod(encodedData);
    receivedSignal = awgnchan(modSignal);
    demodSignal = bpskdemod(receivedSignal);
    receivedBits = turbodec(-demodSignal);
    errorStats = errrate(data,receivedBits);
end
```

Display the error data.

```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n', errorStats)
```

```
Bit error rate = 2.34e-04
Number of errors = 6
Total bits = 25600
```

Perform Turbo Coding on 16-QAM Signal in AWGN Channel

Simulate an end-to-end communication link by using a 16-QAM signal and turbo codes in an AWGN channel. Inside a frame processing loop, packet sizes are randomly selected to be 500, 1000, or 1500 bits. Because the packet size varies, the interleaver indices are provided to the turbo encoder and decoder as an input argument of their associated System object.

Set the modulation order and E_b/N_0. Compute the number of bits per symbol and the energy per symbol to noise ratio (E_S/N_0) based on the modulation order and E_b/N_0. To get repeatable results, set the random number generator to its default state.

```
modOrder = 16;
bps = log2(modOrder); % Bits per symbol
EbNo = 2.8; % Energy per bit to noise power spectral density ratio in dB
EsNo = EbNo + 10*log10(bps); % Energy per symbol to noise power spectral density ratio in dB
rng default
```

Create a turbo encoder and decoder pair. Because the packet length varies for each frame, specify that the interleaver indices be supplied by an input argument of the System object when executed. Specify that the decoder perform four iterations.

```
turboEnc = comm.TurboEncoder('InterleaverIndicesSource','Input port');
turboDec = comm.TurboDecoder('InterleaverIndicesSource','Input port','NumIterations',4);
trellis = poly2trellis(4,[13 15 17],13);
```

```
n = log2(turboEnc.TrellisStructure.numOutputSymbols);
numTails = log2(turboEnc.TrellisStructure.numStates)*n;
```

Create an error rate object.

```
errRate = comm.ErrorRate;
```

The frame processing loop performs these steps.

1. Select a random packet length, and generate random binary data.
2. Compute the output codeword length and coding rate.
3. Compute the signal to noise ratio (SNR) and noise variance.
4. Generate interleaver indices.
5. Turbo encode the data.
6. Apply 16-QAM modulation, and normalize the average signal power.
7. Pass the modulated signal through an AWGN channel.
8. Demodulate the noisy signal by using an LLR method, output soft bits, and normalize the average signal power.
9. Turbo decode the data. Because the bit mapping order from the demodulator is opposite the mapping order expected by the turbo decoder, the decoder input must use the inverse of the demodulated signal.
10. Calculate the error statistics.

```
numFrames = 100;
for pktIdx = 1:numFrames
    L = 500*randi([1 3],1,1); % Packet length in bits
    data = randi([0 1],L,1);

    M = L*(2*n - 1) + 2*numTails; % Output codeword packet length
    rate = L/M; % Coding rate for current packet
    snrdB = EsNo + 10*log10(rate); % Signal to noise ratio in dB
    noiseVar = 1./(10.^(snrdB/10)); % Noise variance

    intrlvrIndices = randperm(L);
    encodedData = turboEnc(data,intrlvrIndices);
    modSignal = qammod(encodedData,modOrder,'InputType','bit','UnitAveragePower',true);
    receivedSignal = awgn(modSignal,snrdB);
    demodSignal = qamdemod(receivedSignal,modOrder,'OutputType','llr', ...
        'UnitAveragePower',true,'NoiseVariance',noiseVar);
    receivedBits = turboDec(-demodSignal,intrlvrIndices); % Demodulated signal is negated

    errorStats = errRate(data,receivedBits);
end
```

Display the error data.

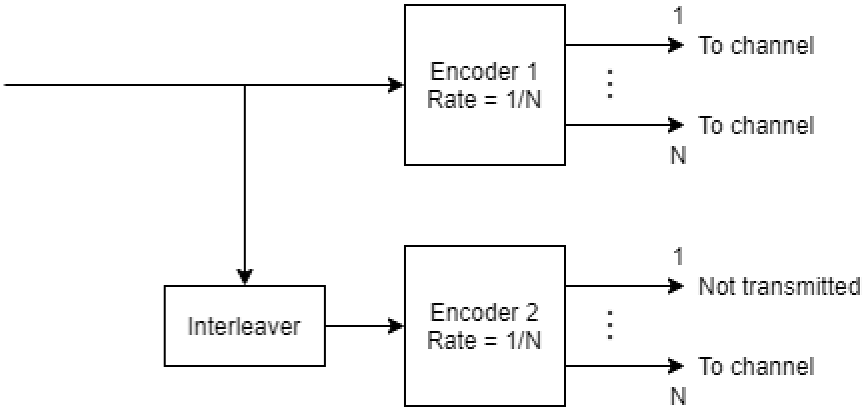
```
fprintf('Bit error rate = %5.2e\nNumber of errors = %d\nTotal bits = %d\n',errorStats)
```

```
Bit error rate = 8.51e-04
Number of errors = 80
Total bits = 94000
```

More About

Parallel Concatenated Convolutional Encoding Scheme

The turbo encoder uses a parallel concatenated convolutional encoding scheme to encode a binary input signal. This figure shows the two identical encoders and one internal interleaver used in the coding scheme. Each constituent encoder is terminated independently by tail bits. The output of the turbo encoder consists of the systematic and parity bit streams of the first encoder and only the parity bit streams of the second encoder.



For example, a rate 1/2 constituent encoder interlaces three streams and multiplexes the tail bits to the end of the encoded data streams.

For more information, see “Coding Rate” on page 3-1424 and “Tail bits” on page 3-1424.

Coding Rate

Each constituent convolutional coder has a 1/*N* coding rate. The coding rate of the constituent convolutional code is represented as a rate *K*/*N* code. *K* is the number of input bit streams. *N* is the number of output bit streams.

Note *K* must be 1 for the turbo coder.

The encoder accepts an *L*-by-1 column vector input signal and returns an *M*-by-1 column vector output signal. The effective code rate of the turbo encoder is *L*/*M*. *L* is the length of the binary input message and *M* is the number of bits in the parallel concatenated codeword.

For a given trellis, *M* and *L* are related by $M = L \times (2 \times N - 1) + 2 \times numTails$, where *numTails* is the number of tail bits. For more information, see “Tail bits” on page 3-1424.

Tail bits

The turbo encoder treats each input independently. For each input message, extra bits are used to set the encoder states to an all-zeros state. Each constituent encoder is terminated independently by tail bits. The turbo encoder output consists of the interlaced systematic and parity streams, with the tail bits multiplexed to the end of the encoded data streams.

The number of tail bits, *numTails*, output by each constituent encoder depends on values in the trellis structure.

- *numTails* = log_2(`trellis.numStates`) × *N*
- *N* = log_2(`trellis.numOutputSymbols`).

For a rate 1/2 trellis, *N* = 2.

For more information about trellis structures, see `poly2trellis`.

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima. “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes.” *Proceedings of ICC 93 - IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, 1064–70. https://doi.org/10.1109/icc.1993.397441.

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

[3] Schlegel, Christian, and Lance Perez. *Trellis and Turbo Coding*. IEEE Press Series on Digital & Mobile Communication. Piscataway, NJ ; Hoboken, NJ: IEEE Press ; Wiley-Interscience, 2004.

[4] 3GPP TS 36.212. "Multiplexing and channel coding." *3rd Generation Partnership Project; Technical Specification Group Radio Access Network; Evolved Universal Terrestrial Radio Access (E-UTRA)*. https://www.3gpp.org.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

Objects

`comm.ConvolutionalEncoder` | `comm.TurboDecoder`

Functions

`convenc` | `istrellis` | `poly2trellis`

Blocks

Turbo Encoder

Introduced in R2012a

comm.ViterbiDecoder

Package: `comm`

Decode convolutionally encoded data using Viterbi algorithm

Description

The `ViterbiDecoder` object decodes input symbols to produce binary output symbols. This object can process several symbols at a time for faster performance. This object processes variable-size signals; however, variable-size signals cannot be applied for erasure inputs.

To decode input symbols and produce binary output symbols:

1. Define and set up your Viterbi decoder object. See “Construction” on page 3-1426.
2. Call `step` to decode input symbols according to the properties of `comm.ViterbiDecoder`. The behavior of `step` is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`H = comm.ViterbiDecoder` creates a Viterbi decoder System object, `H`. This object uses the Viterbi algorithm to decode convolutionally encoded input data.

`H = comm.ViterbiDecoder(Name,Value)` creates a Viterbi decoder object, `H`, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (`Name1`,`Value1`,...,`NameN`,`ValueN`).

`H = comm.ViterbiDecoder(TRELLIS,Name,Value)` creates a Viterbi decoder object, `H`. This object has the `TrellisStructure` property set to `TRELLIS` and the other specified properties set to the specified values.

Properties

`TrellisStructure`

Trellis structure of convolutional code

Specify the trellis as a MATLAB structure that contains the trellis description of the convolutional code. The default is the result of `poly2trellis(7, [171 133])`. Use the `istrellis` function to verify whether a structure is a valid trellis.

`InputFormat`

Input format

Specify the format of the input to the decoder as `Unquantized` | `Hard` | `Soft`. The default is `Unquantized`.

When you set this property to `Unquantized`, the input must be a real vector of double- or single-precision soft values that are unquantized. The object considers negative numbers to be `1`s and positive numbers to be `0`s.

When you set this property to `Hard`, the input must be a vector of hard decision values, which are `0`s or `1`s. The data type of the inputs can be double-precision, single-precision, logical, 8-, 16-, and 32-bit signed integers. You can also use 8-, 16-, and 32-bit unsigned integers.

When you set this property to `Soft`, the input requires a vector of quantized soft values represented as integers between 0 and $2^{SoftInputWordLength} - 1$. The data type of the inputs can be double-precision, single-precision, logical, 8-, 16-, and 32-bit signed integers. You can also use 8-, 16-, and 32-bit unsigned integers. Alternately, you can specify the data type as an unsigned and unscaled fixed point object (fi) with a word length equal to the word length that you specify in the `SoftInputWordLength` property. The object considers negative numbers to be `0`s and positive numbers to be `1`s.

SoftInputWordLength

Soft input word length

Specify the number of bits to represent each quantized soft input value as a positive, integer scalar value. The default is `4` bits. This property applies when you set the `InputFormat` on page 3-0 property to `Soft`.

InvalidQuantizedInputAction

Action when input values are out of range

Specify the action the object takes when input values are out of range as `Ignore` | `Error`. The default is `Ignore`. Set this property to `Error` so that the object generates an error when the quantized input values are out of range. This property applies when you set the `InputFormat` on page 3-0 property to `Hard` or `Soft`.

TracebackDepth

Traceback depth

Specify the number of trellis branches to construct each traceback path as a numeric, integer scalar value. The default is `34`. The traceback depth influences the decoding accuracy and delay. The number of zero symbols that precede the first decoded symbol in the output represent a decoding delay.

When you set the `TerminationMethod` on page 3-0 property to `Continuous`, the decoding delay consists of `TracebackDepth`×*K* zero bits for a rate *K*/*N* convolutional code.

When you set the `TerminationMethod` property to `Truncated` or `Terminated`, there is no output delay.

For more information, see “Traceback and Decoding Delay” on page 3-1431 and “Traceback Depth Estimates” on page 3-1432.

TerminationMethod

Termination method of encoded frame

Specify the termination method as `Continuous` | `Truncated` | `Terminated`. The default is `Continuous`.

In `Continuous` mode, the object saves the internal state metric at the end of each frame for use with the next frame. The object treats each traceback path independently.

In `Truncated` mode, the object treats each frame independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. In `Terminated` mode, the object treats each frame independently, and the traceback path always starts and ends in the all-zeros state.

ResetInputPort

Enable decoder reset input

Set this property to true to enable an additional `step` method input. The default is `false`. When the reset input is a nonzero value, the object resets the internal states of the decoder to initial conditions. This property applies when you set the `TerminationMethod` on page 3-0 property to `Continuous`.

DelayedResetAction

Reset on nonzero input via port

Set this property to true to delay resetting the object output. The default is false. When you set this property to true, the reset of the internal states of the decoder occurs after the object computes the decoded data. When you set this property to false, the reset of the internal states of the decoder occurs before the object computes the decoded data. This property applies when you set the `ResetInputPort` on page 3-0 property to true.

PuncturePatternSource

Source of puncture pattern

Specify the source of the puncture pattern as `None` | `Property`. The default is `None`.

When you set this property to `None`, the object assumes no puncturing. Set this property to `Property` to decode punctured codewords based on a puncture pattern vector specified via the `PuncturePattern` on page 3-0 property.

PuncturePattern

Puncture pattern vector

Specify puncture pattern to puncture the encoded data. The default is `[1; 1; 0; 1; 0; 1]`. The puncture pattern is a column vector of `1`s and `0`s. The `0`s indicate the position to insert dummy bits. The puncture pattern must match the puncture pattern used by the encoder. This property applies when you set the `PuncturePatternSource` on page 3-0 property to `Property`.

ErasuresInputPort

Enable erasures input

Set this property to true to specify a vector of erasures as a step method input. The default is false. The erasures input must be a double-precision or logical, binary, column vector. This vector indicates which symbols of the input codewords to erase. Values of 1 indicate erased bits. The decoder does not update the branch metric for the erasures in the incoming data stream.

The lengths of the step method erasure input and the step method data input must be the same. When you set this property to false, the object assumes no erasures.

OutputDataType

Data type of output

Specify the data type of the output as Full precision | Smallest unsigned integer | double | single | int8 | uint8 | int16 | uint16 | int32 | uint32 | logical. The default is Full precision.

When the input signal is an integer data type, you must have a Fixed-Point Designer user license to use this property in Smallest unsigned integer or Full precision mode.

Fixed-Point Properties

StateMetricDataType

Data type of state metric

Specify the state metric data type as Full precision | Custom. The default is Full precision.

When you set this property to Full precision, the object sets the state metric fixed-point type to numerictype([],16). This property applies when you set the InputFormat on page 3-0 property to Hard or Soft.

When you set the InputFormat property to Hard, the step method data input must be a column vector. This vector comprises unsigned, fixed point numbers (fi objects) of word length 1 to enable fixed-point Viterbi decoding. Based on this input (either a 0 or a 1), the object calculates the internal branch metrics using an unsigned integer of word length *L*. In this case, *L* indicates the number of output bits as specified by the trellis structure.

When you set the InputFormat property to Soft, the step method data input must be a column vector. This vector comprises unsigned, fixed point numbers (fi objects) of word length N. N indicates the number of soft-decision bits specified in the SoftInputWordLength on page 3-0 property.

The step method data inputs must be integers in the range 0 to 2^N–1. The object calculates the internal branch metrics using an unsigned integer of word length $L = (N + Nout - 1)$. In this case, *Nout* represents the number of output bits as specified by the trellis structure.

CustomStateMetricDataType

Fixed-point data type of state metric

Specify the state metric fixed-point type as an unscaled, numerictype object with a signedness of Auto. The default is numerictype([],16). This property applies when you set the StateMetricDataType on page 3-0 property to Custom.

Methods

reset	Reset states of the Viterbi decoder object
step	Decode convolutionally encoded data using Viterbi algorithm

Common to All System Objects	
release	Allow System object property value changes

Examples

Encode and Decode 8-DPSK Modulated Data

Transmit a convolutionally encoded 8-DPSK modulated bit stream through an AWGN channel. Then, demodulate and decode using a Viterbi decoder.

Create the necessary System objects.

```
hConEnc = comm.ConvolutionalEncoder;
hMod = comm.DPSKModulator('BitInput',true);
hChan = comm.AWGNChannel('NoiseMethod', ...
    'Signal to noise ratio (SNR)',...
    'SNR',10);
hDemod = comm.DPSKDemodulator('BitOutput',true);
hDec = comm.ViterbiDecoder('InputFormat','Hard');
hError = comm.ErrorRate('ComputationDelay',3,'ReceiveDelay', 34);
```

Process the data using the following steps:

1 Generate random bits
2 Convolutionally encode the data
3 Apply DPSK modulation
4 Pass the modulated signal through AWGN
5 Demodulate the noisy signal
6 Decode the data using a Viterbi algorithm
7 Collect error statistics

```
for counter = 1:20
    data = randi([0 1],30,1);
    encodedData = step(hConEnc, data);
    modSignal = step(hMod, encodedData);
    receivedSignal = step(hChan, modSignal);
    demodSignal = step(hDemod, receivedSignal);
    receivedBits = step(hDec, demodSignal);
    errors = step(hError, data, receivedBits);
end
```

Display the number of errors.

```
errors(2)
```

```
ans = 3
```

Convolutional Encoding and Viterbi Decoding with a Puncture Pattern Matrix

Encode and decode a sequence of bits using a convolutional encoder and a Viterbi decoder with a defined puncture pattern. Verify that the input and output bits are identical

Define a puncture pattern matrix and reshape it into vector form for use with the Encoder and Decoder objects.

```
pPatternMat = [1 0 1;1 1 0];
pPatternVec = reshape(pPatternMat,6,1);
```

Create convolutional encoder and a Viterbi decoder in which the puncture pattern is defined by `pPatternVec`.

```
ENC = comm.ConvolutionalEncoder(...
    'PuncturePatternSource','Property', ...
    'PuncturePattern',pPatternVec);

DEC = comm.ViterbiDecoder('InputFormat','Hard', ...
    'PuncturePatternSource','Property',...
    'PuncturePattern',pPatternVec);
```

Create an error rate counter with the appropriate receive delay.

```
ERR = comm.ErrorRate('ReceiveDelay',DEC.TracebackDepth);
```

Encode and decode a sequence of random bits.

```
dataIn = randi([0 1],600,1);

dataEncoded = step(ENC,dataIn);

dataOut = step(DEC,dataEncoded);
```

Verify that there are no errors in the output data.

```
errStats = step(ERR,dataIn,dataOut);
errStats(2)
```

```
ans = 0
```

More About

Traceback and Decoding Delay

The traceback depth influences the decoding delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

- For the continuous operating mode, the decoding delay is equal to the number of traceback depth symbols.
- For the truncated or terminated operating mode, the decoding delay is zero. In this case, the traceback depth must be less than or equal to the number of symbols in each input.

Traceback Depth Estimates

As a general estimate, a typical traceback depth value is approximately two to three times (*ConstraintLength* – 1) / (1 – *coderate*). The constraint length of the code, *ConstraintLength*, is equal to (log2(`trellis`.`numStates`) + 1). The *coderate* is equal to (*K* / *N*) × (length(*PuncturePattern*) / sum(*PuncturePattern*).

K is the number of input symbols, *N* is the number of output symbols, and *PuncturePattern* is the puncture pattern vector.

For example, applying this general estimate, results in these approximate traceback depths.

- A rate 1/2 code has a traceback depth of 5(*ConstraintLength* – 1).
- A rate 2/3 code has a traceback depth of 7.5(*ConstraintLength* – 1).
- A rate 3/4 code has a traceback depth of 10(*ConstraintLength* – 1).
- A rate 5/6 code has a traceback depth of 15(*ConstraintLength* – 1).

Algorithms

This object implements the algorithm, inputs, and outputs described on the Viterbi Decoder block reference page. The object properties correspond to the block parameters, except:

- The **Decision type** parameter corresponds to the `InputFormat` on page 3-0 property.
- The **Operation mode** parameter corresponds to the `TerminationMethod` on page 3-0 property.

References

[1] Moision, B. "A Truncation Depth Rule of Thumb for Convolutional Codes." *In Information Theory and Applications Workshop* (January 27 2008-February 1 2008, San Diego, California), 555-557. New York: IEEE, 2008.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.APPDecoder` | `comm.ConvolutionalEncoder`

Topics

“Log-likelihood Ratio (LLR) Demodulation”

Introduced in R2012a

reset

System object: `comm.ViterbiDecoder`
Package: `comm`

Reset states of the Viterbi decoder object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `ViterbiDecoder` object, `H`.

step

System object: `comm.ViterbiDecoder`
Package: `comm`

Decode convolutionally encoded data using Viterbi algorithm

Syntax

```
Y = step(H,X)
Y = step(H,X,ERASURES)
Y = step(H,X,R)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`Y = step(H,X)` decodes encoded data, `X`, using the Viterbi algorithm and returns `Y`. `X`, must be a column vector with data type and values that depend on how you set the `InputFormat` property. If the convolutional code uses an alphabet of 2^N possible symbols, the length of the input vector, `X`, must be $L \times N$ for some positive integer L. Similarly, if the decoded data uses an alphabet of 2^K possible output symbols, the length of the output vector, `Y`, is $L \times K$.

`Y = step(H,X,ERASURES)` uses the binary column input vector, `ERASURES`, to erase the symbols of the input codewords. The elements in `ERASURES` must be of data type double or logical. Values of `1` in the `ERASURES` vector correspond to erased symbols, and values of `0` correspond to non-erased symbols. The lengths of the `X` and `ERASURES` inputs must be the same. This syntax applies when you set the `ErasuresInputPort` property to true.

`Y = step(H,X,R)` resets the internal states of the decoder when you input a non-zero reset signal, `R`. `R` must be a double precision or logical scalar. This syntax applies when you set the `TerminationMethod` property to `Continuous` and the `ResetInputPort` property to `true`.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.WalshCode

Package: comm

Generate Walsh code from orthogonal set of codes

Description

The WalshCode object generates a Walsh code from an orthogonal set of codes.

To generate a Walsh code:

1. Define and set up your Walsh code object. See “Construction” on page 3-1435.
2. Call step to encode the input signal according to the properties of comm.WalshCode. The behavior of step is specific to each object in the toolbox.

Note Starting in R2016b, instead of using the step method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, y = step(obj) and y = obj() perform equivalent operations.

Construction

H = comm.WalshCode creates a Walsh code generator System object, H. This object generates a Walsh code from a set of orthogonal codes.

H = comm.WalshCode(Name,Value) creates a Walsh code generator object, H, with each specified property set to the specified value. You can specify additional name-value pair arguments in any order as (Name1,Value1,...,NameN,ValueN).

Properties

Length

Length of generated code

Specify the length of the generated code as a numeric, integer scalar value that is a power of two. The default is 64.

Index

Index of code of interest

Specify the index of the desired code from the available set of codes as a numeric, integer scalar value in the range [0, 1, ... , *N*-1]. *N* is the value of the Length on page 3-0 property. The default is 60. The number of zero crossings in the generated code equals the value of the specified index.

SamplesPerFrame

Number of output samples per frame

Specify the number of Walsh code samples that the step method outputs as a numeric, positive, integer scalar value . The default is `1`. If you set this property to a value of *M*, then the `step` method outputs *M* samples of a Walsh code of length *N*. *N* is the length of the code that you specify in the `Length` on page 3-0 property.

`OutputDataType`

Data type of output

Specify the output data type as `double` | `int8`. The default is `double`.

Methods

reset	Reset states of Walsh code generator object
step	Generate Walsh code from orthogonal set of codes

Common to All System Objects	
`release`	Allow System object property value changes

Examples

Walsh Code Sequence

Generate 16 samples of a length-64 Walsh code sequence.

```
walsh = comm.WalshCode('SamplesPerFrame',16)
```

```
walsh = 
  comm.WalshCode with properties:

             Length: 64
              Index: 60
    SamplesPerFrame: 16
     OutputDataType: 'double'
```

```
seq = walsh()
```

```
seq = 16×1

     1
    -1
     1
    -1
     1
    -1
     1
    -1
     1
    -1
      ⋮
```

Algorithms

This object implements the algorithm, inputs, and outputs described on the Walsh Code Generator block reference page. The object properties correspond to the block parameters, except:

- The object does not have a property to select frame based outputs.
- The object does not have a property that corresponds to the **Sample time** parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

See “System Objects in MATLAB Code Generation” (MATLAB Coder).

See Also

`comm.HadamardCode` | `comm.OVSFCode`

Introduced in R2012a

reset

System object: `comm.WalshCode`
Package: `comm`

Reset states of Walsh code generator object

Syntax

```
reset(H)
```

Description

`reset(H)` resets the states of the `WalshCode` object, `H`.

step

System object: `comm.WalshCode`
Package: `comm`

Generate Walsh code from orthogonal set of codes

Syntax

```
Y = step(H)
```

Description

Note Starting in R2016b, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj)` and `y = obj()` perform equivalent operations.

`Y = step(H)` outputs a frame of the Walsh code in column vector `Y`. Specify the frame length with the `SamplesPerFrame` property. The Walsh code corresponds to a row of an *N*x*N* Hadamard matrix, where *N* is a nonnegative power of 2 that you specify in the `Length` property. Use the `Index` property to choose the row of the Hadamard matrix. The output code is in a bi-polar format with `0` and `1` mapped to `1` and `-1` respectively.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

comm.WINNER2Channel

Package: `comm`

Filter input signal through WINNER II fading channel

Download Required

To use `comm.WINNER2Channel`, first download the WINNER II Channel Model for Communications Toolbox from the Add-On Explorer. For more information on downloading add-ons, see “Get and Manage Add-Ons”.

Description

The `comm.WINNER2Channel` System object filters an input signal through a WINNER II fading channel. The object utilizes the basic model defined and provided by the WINNER II Channel Models [1].

To filter an input signal using a WINNER II fading channel:

1. Define and set up your WINNER II channel object. See “Construction” on page 3-1440.
2. Call `step` to filter the input signal through a WINNER II fading channel according to the properties of `comm.WINNER2Channel`.

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

Construction

`chan = comm.WINNER2Channel` creates a WINNER II fading channel System object to model single or multiple links. `chan` generates channel coefficients using the WINNER II spatial channel model (SCM). It also filters a real or complex input signal through the fading channel for each link.

`chan = comm.WINNER2Channel(Name,Value)` creates a WINNER II fading channel object, `chan`, that overrides default values using one or more `Name,Value` pair arguments. You can specify additional name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`chan = comm.WINNER2Channel(cfgModel)` creates a WINNER II fading channel object with the `ModelConfig` property set to `cfgModel`.

`chan = comm.WINNER2Channel(cfgModel,cfgLayout)` creates a WINNER II fading channel object with the `ModelConfig` property set to `cfgModel` and the `LayoutConfig` property set to `cfgLayout`.

Properties

`ModelConfig`

WINNER II model parameter configuration

WINNER II model parameter configuration, specified as a structure containing these fields:

`NumTimeSamples`

Number of time samples. The default value is 100.

Note If the number of samples in the input signal (N_S) does not match `NumTimeSamples`, `NumTimeSamples` is updated to match N_S.

`FixedPdpUsed`

Set to `'yes'` to use predefined path delays and powers for specific scenarios. The default value is `'no'`.

`FixedAnglesUsed`

Set to `'yes'` to use predefined angles of departure (AoDs) and angles of arrival (AoAs) for specific scenarios. The default value is `'no'`.

`IntraClusterDsUsed`

Set to `'yes'` to divide each of the two strongest clusters per link into three subclusters. The default value is `'yes'`.

`PolarisedArrays`

Set to `'yes'` to use dual polarized arrays. The default value is `'yes'`.

`UseManualPropCondition`

Set to `'yes'` to use the manually defined propagation conditions (LOS or NLOS) in the `LayoutConfig.PropagConditionVector` field. Set to `'no'` to draw propagation conditions from predefined LOS probabilities. The default value is `'yes'`.

`UniformTimeSampling`

Set to `'yes'` to sample all links at the same time instants. The default value is `'no'`.

`SampleDensity`

Number of time samples per half wavelength. The default value is `2e6`.

`CenterFrequency`

Center frequency of carrier. The default value `5.25e9` Hz.

DelaySamplingInterval

Sampling grid to which the path delays are rounded. The default value of `0` seconds indicates no rounding on path delays.

- `DelaySamplingInterval` specifies the input signal sample time.
- When performing channel filtering, the object uses `DelaySamplingInterval = 0` to obtain the original path delays. Any non-zero value of `DelaySamplingInterval` is ignored, specifically the path delays used are not rounded to be multiples of `DelaySamplingInterval` values that are non-zero.

ShadowingModelUsed

Set to `'yes'` to include shadow fading in the model. The default value is `'no'`.

PathLossModelUsed

Set to `'yes'` to include path loss in the model. The default value is `'no'`.

PathLossModel

Path loss model function name, specified as `'pathloss'`, which uses the internal `pathloss` function from the “WINNER II Channel” Add-On to model the path loss. The `PathLossModel` property is applicable only when `PathLossModelUsed` is `'yes'`. The default value is `'pathloss'`.

PathLossOption

Path loss option indicating the wall material for the NLOS path loss calculation of scenario A1, specified as one of `{'CR_light', 'CR_heavy', 'RR_light', 'RR_heavy'}`. The default value is `'CR_light'`. The `PathLossOption` property is applicable only when `PathLossModelUsed` is `'yes'`.

See `LayoutConfig`.`ScenarioVector` for the scenario number mapping.

RandomSeed

Seed for random number generators. To use the global random stream, set `RandomSeed` to empty, `[]`. The default is `[]`.

LayoutConfig

WINNER II layout parameter configuration

WINNER II layout parameter configuration, specified as a structure containing these fields:

Stations

Row vector of structures to describe antenna arrays for active stations. The row ordering specifies BS sectors first, followed by the MS. The default assigns two structures, one for BS and one for MS.

NofSect

Vector of number of sectors in each BS. The default is `1`.

Pairing

A 2-by-N_L matrix, where N_L specifies the number links to be modeled. The default is `[1;2]`.

ScenarioVector

A 1-by-N_L vector of scenario numbers. The default is `1`, which specifies scenario A1.

The scenarios numbers map as {`1`=A1, `2`=A2, `3`=B1, `4`=B2, `5`=B3, `6`=B4, `10`=C1, `11`=C2, `12`=C3, `13`=C4, `14`=D1, `15`=D2a}.

For more information, see WINNER II Channel Models [1], Section 2.3.

PropagConditionVector

A 1-by-N_L vector of propagation conditions (LOS = 1 and NLOS = 0) for each link. The default is `1`.

StreetWidth

A 1-by-N_L vector of identical values that specify the average width (in meters) of the streets. `StreetWidth` is used for the path loss model of the B1 and B2 scenarios. The default is `20`. See `ScenarioVector` for the scenario number mapping. All elements must have the same value. The `StreetWidth` property is applicable only when the `ModelConfig.PathLossModelUsed` property is `'yes'`.

Dist1

A 1-by-N_L vector of distances from BS to the last LOS point. `Dist1` is used for the path loss model of the B1 and B2 scenarios. The default value is `NaN`, which means the distance is randomly determined in path loss function. See `ScenarioVector` for the scenario number mapping. `Dist1` is applicable only when the `ModelConfig.PathLossModelUsed` property is `'yes'`.

For more information, see WINNER II Channel Models [1], Figure 4-3.

NumFloors

A 1-by-N_L vector indicating the floor number where the indoor BS or MS is located. The default value is `1`. The `NumFloors` property is used for the path loss model of the A2 and B4 scenarios only. See `ScenarioVector` for the scenario number mapping. The `NumFloors` property is applicable only when `ModelConfig.PathLossModelUsed` is `'yes'`.

NumPenetratedFloors

A 1-by-N_L vector indicating the number of penetrated floors between BS and MS. The default value is `0`. The `NumPenetratedFloors` is used for the NLOS path loss model of the A1 scenario. See `ScenarioVector` for the scenario number mapping. The `NumPenetratedFloors` property is applicable only when `PathLossModelUsed` is `'yes'`.

For more information, see WINNER II Channel Models [1], Table 4-4.

NormalizeChannelOutputs

Normalize channel outputs, specified as `true` or `false`. Set this property to `true` to normalize the channel outputs by the number of receive antennas at the mobile station (MS) for each link. The default value is `true`.

For more information, see “Channel Power” on page 3-1447.

Methods

info	Display information about `WINNER2Channel` object
reset	Reset states of `WINNER2Channel` object
step	Filter input signal through WINNER II fading channel

Common to All System Objects	
`release`	Allow System object property value changes

Examples

WINNER II Channel with Two Mobile Stations

Simulate a system that has two MS connected to one BS. One MS is 8 meters away from the BS; the other is 20 meters away from the BS. Impulse signals are sent through the two links. The spectrum of the received signals at MS shows frequency selectivity. It also shows the MS that is closer the BS has a larger average received power than the other MS.

Specify random number generator seed for repeatability.

```
rng(100);
```

Initial frame length and sample rate.

```
frmLen   = 1024;
```

Configure layout parameters.

```
BSAA  = winner2.AntennaArray('UCA', 8, 0.02);  % UCA-8 antenna array for BS
MSAA1 = winner2.AntennaArray('ULA', 2, 0.01);  % ULA-2 antenna array for MS
MSAA2 = winner2.AntennaArray('ULA', 4, 0.005); % ULA-4 antenna array for MS
MSIdx = [2 3]; BSIdx = {1}; NL = 2; maxRange = 100; rndSeed = 101;
cfgLayout = winner2.layoutparset(MSIdx,BSIdx,NL, ...
   [BSAA,MSAA1,MSAA2],maxRange,rndSeed);
```

Adjust BS and MS positions.

```
cfgLayout.Stations(1).Pos(1:2) = [10, 10];
cfgLayout.Stations(2).Pos(1:2) = [18, 10];  % 8 meters away from BS
cfgLayout.Stations(3).Pos(1:2) = [22, 26];  % 20 meters away from BS
```

NLOS for both links

```
cfgLayout.Pairing = [1 1; 2 3];
cfgLayout.PropagConditionVector = [0 0];
```

Configure model parameters

```
cfgModel = winner2.wimparset;
cfgModel.NumTimeSamples     = frmLen; % Frame length
cfgModel.IntraClusterDsUsed = 'no';   % No cluster splitting
cfgModel.SampleDensity      = 2e5;    % For lower sample rate
cfgModel.PathLossModelUsed  = 'yes';  % Turn on path loss
cfgModel.ShadowingModelUsed = 'yes';  % Turn on shadowing
```

Create a WINNER II channel System object.

```
wimChan = comm.WINNER2Channel(cfgModel, cfgLayout);
```

Call the info method of the object to get some system information

```
chanInfo = info(wimChan)
```

```
chanInfo = struct with fields:
               NumLinks: 2
          NumBSElements: [8 8]
          NumMSElements: [2 4]
               NumPaths: [16 16]
             SampleRate: [1.0000e+07 1.0000e+07]
     ChannelFilterDelay: [7 7]
    NumSamplesProcessed: 0
```

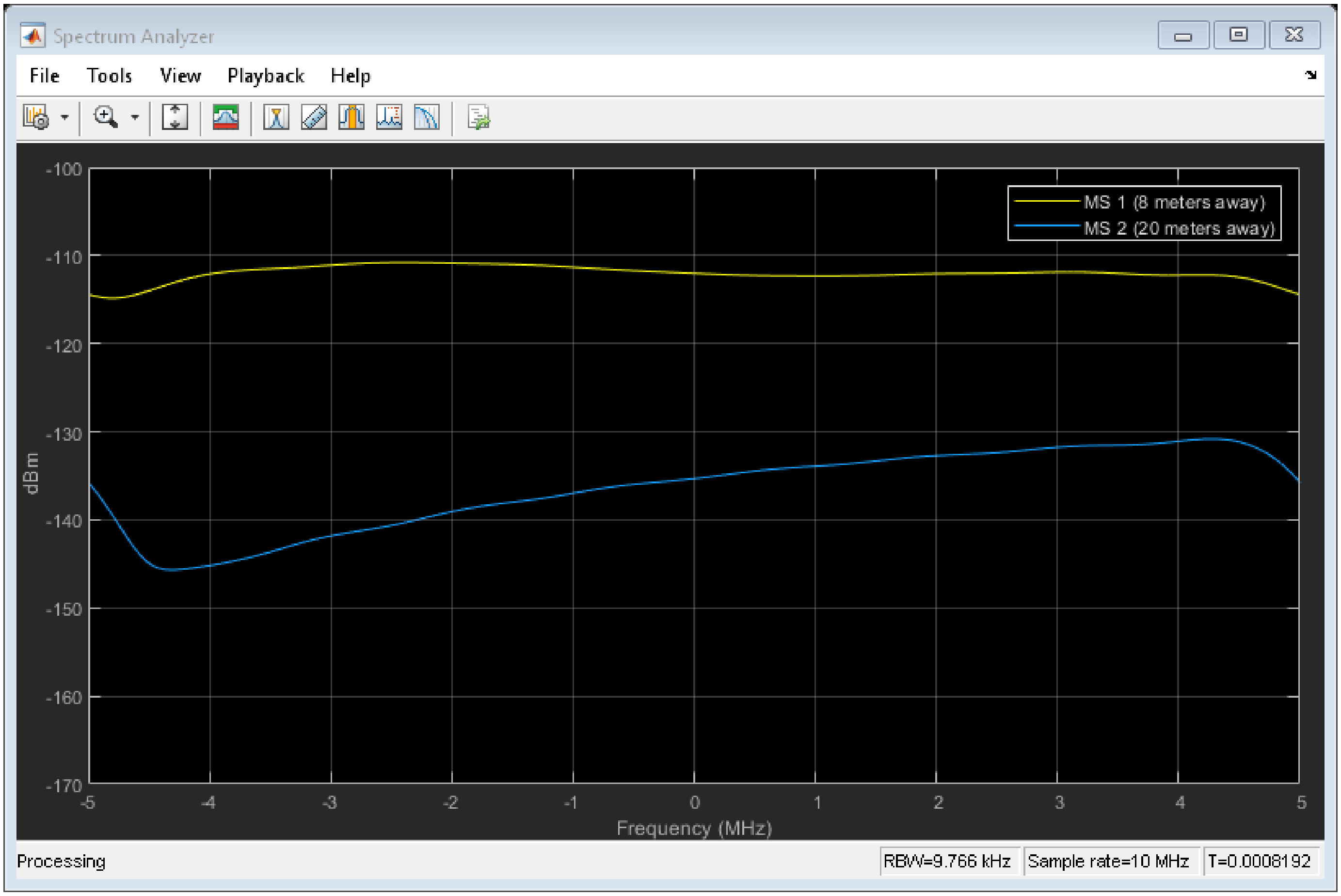
```
numTx    = chanInfo.NumBSElements(1);
Rs       = chanInfo.SampleRate(1);
```

Create a Spectrum Analyzer System object.

```
SA = dsp.SpectrumAnalyzer('SampleRate', Rs, ...
    'YLimits', [-170, -100], 'ShowLegend', true, ...
    'ChannelNames', {'MS 1 (8 meters away)','MS 2 (20 meters away)'});
```

Pass impulse signals through the two links and show spectra of the received signals at the two MS.

```
for i = 1:10
    x = [ones(1,numTx); zeros(frmLen-1, numTx)];
    y = wimChan(x);
    SA([y{1}(:,1), y{2}(:,1)]);
end
```



More About

WINNER II Sampling Rate

The signal sample rate (R_S) for generating channel coefficients and performing channel filtering is calculated per link using the mobile station speed (V_{MS}), half wavelength distance, and sample density. The sample rate for each link is available as a field in the info method return.

$R_S = V_{MS} / (C / F_{center} / 2 / N_{SD})$,

- For the MS speed,V_{MS},
  - When `ModelConfig.UniformTimeSampling` is set to `'no'`, V_{MS} is the speed of the MS for the corresponding link, derived from the `LayoutConfig.Stations(i).Velocity` field.
  - When `ModelConfig.UniformTimeSampling` is set to `'yes'`, V_{MS} is the maximum speed of the MS for all links.
- C is the speed of light (`2.99792458e8` m/s).
- F_{center} is `ModelConfig.CenterFrequency`.
- N_{SD} is `ModelConfig.SampleDensity`.

Channel Power

These conditions apply to the channel power of the `comm.WINNER2Channel` object:

- When path loss and shadowing are off, path gains are normalized. Specifically, path gains are normalized when the ModelConfig.`ShadowingModelUsed` and ModelConfig.`PathLossModelUsed` parameters are set to `'no'`.
- When the NormalizeChannelOutputs property is `true`, the average gain of the channel is 0 dB.

References

[1] Kyosti, Pekka, Juha Meinila, et al. *WINNER II Channel Models*. D1.1.2 V1.2. IST-4–027756 WINNER II, September 2007.

See Also

Objects
`comm.AWGNChannel` | `comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Functions
`winner2.AntennaArray` | `winner2.layoutparset` | `winner2.wim` | `winner2.wimparset`

Introduced in R2016b

info

System object: `comm.WINNER2Channel`
Package: `comm`

Display information about `WINNER2Channel` object

Syntax

```
s = info(obj)
```

Description

`s = info(obj)` returns a structure containing information about the `Winner2Channel` System object characteristics. The information structure contains:

- `Numlinks` - Number of links in the system
- `NumBSElements` - Number of transmit antennas at the BS for each link
- `NumMSElements` - Number of receive antennas at the MS for each link
- `NumPaths` - Number of delay paths for each link
- `SampleRate` - Sample rate for each link
- `ChannelFilterDelay` - Channel filter delay per link, measured in samples
- `NumSamplesProcessed` - Number of samples the channel has processed since the last reset

Introduced in R2016b

reset

System object: `comm.WINNER2Channel`
Package: `comm`

Reset states of `WINNER2Channel` object

Syntax

```
reset(obj)
```

Description

`reset(obj)` resets the states of the `Winner2Channel` System object.

If the `ModelConfig.RandomSeed` property of `obj` is empty, the `reset` method resets the filters only. Otherwise, the `reset` method resets the filters and also reinitializes the random number stream to the value of the `ModelConfig.RandomSeed` property.

Introduced in R2016b

step

System object: `comm.WINNER2Channel`
Package: `comm`

Filter input signal through WINNER II fading channel

Syntax

```
y = step(obj,x)
[y,pathGains] = step(obj,x)
```

Description

Note Alternatively, instead of using the `step` method to perform the operation defined by the System object, you can call the object with arguments, as if it were a function. For example, `y = step(obj,x)` and `y = obj(x)` perform equivalent operations.

`y = step(obj,x)` filters input signal `x` through a WINNER II fading channel and returns the result in `y`. Both `x` and `y` are N_L-by-1 cell arrays, where N_L represents the number of links, as determined by the `LayoutConfig` property of `obj`. The `i`th element of `x` must be an N_S-by-N_T`(i)` matrix of doubles.

- N_S represents the number of samples to be generated and must be the same for all elements of `x`.
- N_T`(i)` is the number of transmit antennas at the base station (BS) for the `i`th link, determined by the `LayoutConfig` property of `obj`.

If the channel has only one link or if all links have the same number of transmit antennas, `x` can also be an N_S-by-N_T matrix of doubles. In this case, the same input signal is filtered through all the links. The `i`th element of `y` is an N_S-by-N_R`(i)` matrix of doubles. N_R`(i)` is the number of receive antennas at the mobile station (MS) for the `i`th link, as determined by the `LayoutConfig` property of `obj`.

`[y,pathGains] = step(obj,x)` also returns the channel coefficients of the underlying WINNER II fading process. `pathGains` is an N_L-by-1 cell array. The `i`th element of `pathGains` is an N_R`(i)`-by-N_T`(i)`-by-N_P`(i)`-by-N_S array of complex doubles. N_P`(i)` is the number of paths for the `i`th link, as determined by the `LayoutConfig` property of `obj`.

N_R, N_T, and N_P are link specific. N_S is the same for all the links.

Note `obj` specifies the System object on which to run this `step` method.

The object performs an initialization the first time the `step` method is executed. This initialization locks nontunable properties and input specifications, such as dimensions, complexity, and data type of the input data. If you change a nontunable property or an input specification, the System object issues an error. To change nontunable properties or inputs, you must first call the `release` method to unlock the object.

Introduced in R2016b

rxsite

Create radio frequency receiver site

Description

Use the `rxsite` object to create a radio frequency receiver site.

Creation

Syntax

```
rx = rxsite
rx = rxsite(coordsys)
rx = rxsite(Name,Value)
```

Description

`rx = rxsite` creates a radio frequency receiver site.

`rx = rxsite(coordsys)` creates a receiver site with coordinate system set to `'geographic'` or `'cartesian'`.

`rx = rxsite(Name,Value)` sets properties using one or more name-value pairs. For example, `rx = rxsite('Name','RX Site')` creates a receiver site with name `RX Site`. Enclose each property name in quotes.

Create a 1-by-*N* array of receiver sites by specifying a property value as an array of *N* columns. Other property values must be specified with either 1 or *N* columns. The `Name`, `Latitude`, and `Longitude` properties may be specified as either a row vector or column vector with *N* elements. The `CoordinateSystem` property must be a string scalar or a character vector.

Properties

Name — Site name
character vector | string | row or column vector

Site name, specified as a character vector or as a row or column vector or as a string.

Example: `'Name','Site 3'`

Example: `rx.Name = 'Site 3'`

Example: If you want to assign multiple values then - `names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"]; rx = rxsite('Name',names)`

Data Types: `char` | `string`

`CoordinateSystem` — Coordinate system of site location
`'geographic'` (default) | `'cartesian'`

Coordinate system of the site location, specified as `'geographic'` or `'cartesian'`. If this property is `'geographic'`, the site location is defined using the properties `Latitude`, `Longitude`, and `AntennaHeight`. If this property is `'cartesian'`, the site location is defined using `AntennaPosition`.

Example: `'CoordinateSystem'`,`'cartesian'`

Example: `tx.CoordinateSystem = 'cartesian'`

Latitude — Site latitude coordinates
`42.3021` (default) | numeric scalar | row or column vector

Site latitude coordinates, specified as a numeric scalar or a row or column vector in the range of range `-90` to `90`. Coordinates are defined using Earth ellipsoid model WGS-84. Latitude is the north/south angle.

Example: `'Latitude'`,`45.098`

Example: `rx.Latitude = 45.098`

Example: If you want to assign multiple values then - `latitude = [42.3467,42.3598,42.3763]; rx = rxsite('Latitude',latitude)`

Dependencies

To use this property, `CoordinateSystem` must be set to `'geographic'`.

Longitude — Site longitude coordinates
`-71.3764` (default) | numeric scalar | row or column vector

Site longitude coordinates, specified as a numeric scalar or a row or column vector. Coordinates are defined using Earth ellipsoid model WGS-84. Longitude is the east/west angle.

Example: `'Longitude'`,`-68.890`

Example: `rx.Longitude = -68.890`

Example: If you want to assign multiple values then - `longitude = [-71.0972,-71.0545,-71.0611]; rx = rxsite('Longitude',longitude)`

Dependencies

To use this property, `CoordinateSystem` must be set to `'geographic'`.

Antenna — Antenna element or array
`'isotropic'` (default) | object | row vector

Antenna element or array specified as one of these:

- `'isotropic'` to model an antenna that radiates uniformly in all directions.
- An `arrayConfig` object.
- If you have Antenna Toolbox™, an antenna element from the “Antenna Catalog” (Antenna Toolbox).
- If you have Phased Array System Toolbox™, any antenna object in “Antennas, Microphones, and Sonar Transducers” (Phased Array System Toolbox) or any array object in “Array Geometries and Analysis” (Phased Array System Toolbox).

Example: `'Antenna'`,`cfgArray`, where `cfgArray` is an `arrayConfig` object

Example: `rx.Antenna = arrayConfig('Size',[8 1]);` specifies an 8-element ULA along z-axis

`AntennaAngle` — Angle of antenna local X-axis

`0` (default) | numeric scalar | 2-by-1 vector | 2-by-*N* matrix

Angle of antenna local Cartesian coordinate system X-axis, specified as a numeric scalar representing azimuth angle in degrees or a 2-by-1 vector representing both azimuth and elevation angles with each element unit in degrees.

The azimuth angle is measured counterclockwise to the antenna X-axis, either from the east (for geographical sites) or from the global X-axis around the global Z-axis (for Cartesian sites).

The elevation angle is measured from the horizontal plane or X-Y plane to the antenna X-axis in the range `-90` to `90` degrees.

Example: `'AntennaAngle'`,25

Example: `tx.AntennaAngle = [25,-80]`

`AntennaHeight` — Antenna height above surface

`1` (default) | non-negative numeric scalar | row vector

Antenna height from the ground or building surface, specified as a non-negative numeric scalar in meters. Maximum value for this property is 6,371,000 m.

If the site coincides with the building, the height is measured from the top of the building to the center of the antenna. Otherwise,the height is measured from ground elevation to the center of the antenna.

Example: `'AntennaHeight'`,25

Example: `rx.AntennaHeight = 15`

Dependencies

To use this property, `CoordinateSystem` must be set to `'geographic'`.

Data Types:

`AntennaPosition` — Position of antenna center

`[0;0;0]` (default) | 3-by-1 vector

Position of the antenna center, specified as a 3-by-1 vector representing [*x;y;z*] Cartesian coordinates with each element in meters.

Example: `'AntennaPosition'`,`[0;2;4]`

Example: `tx.AntennaPosition = [0;2;4]`

Dependencies

To use this property, choose `CoordinateSystem` must be set to `'cartesian'`.

Data Types:

`SystemLoss` — System loss

`0` (default) | nonnegative numeric scalar | row vector

System loss, specified as a non-negative numeric scalar or a row vector in dB.

System loss includes transmission line loss and any other miscellaneous system losses.

Example: `'SystemLoss'`,`10`

Example: `rx.SystemLoss = 10`

Data Types:

`ReceiverSensitivity` — Minimum received power to detect signal

`-100` (default) | numeric scalar | row vector

Minimum received power to detect the signal, specified as a numeric scalar or a row vector in dBm.

Example: `'ReceiverSensitivity'`,`-80`

Example: `rx.ReceiverSensitivity = -80`

Data Types: `double`

Object Functions

show	Show site location on map
hide	Hide site location on map
distance	Distance between sites
angle	Angle between sites
elevation	Elevation of site
location	Location coordinates at a given distance and angle from site
sigstrength	Signal strength due to transmitter
los	Plot or compute the line-of-sight (LOS) visibility between sites on a map
link	Display communication link on map
pattern	Plot antenna radiation pattern on map

Examples

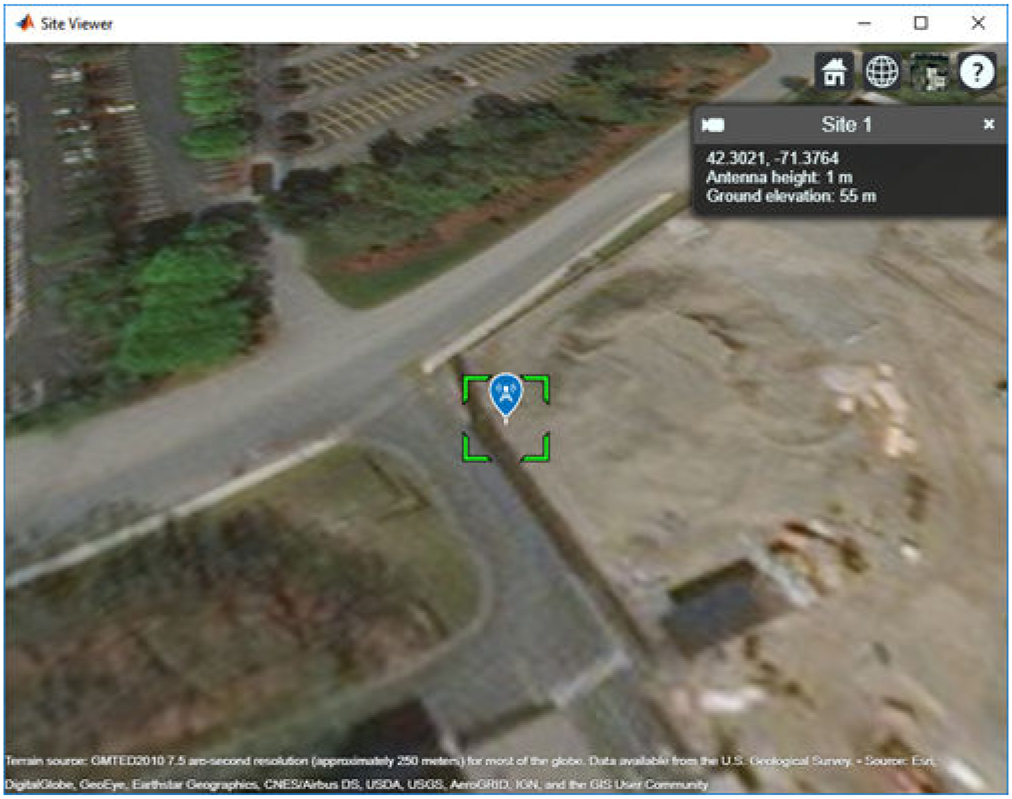
Default Receiver Site

Create and show the default receiver site.

```
rx = rxsite
```

```
rx = 
  rxsite with properties:

                   Name: 'Site 2'
               Latitude: 42.3021
              Longitude: -71.3764
                Antenna: 'isotropic'
           AntennaAngle: 0
          AntennaHeight: 1
             SystemLoss: 0
    ReceiverSensitivity: -100
```

```
show(rx)
```



Site 1

42.3021, -71.3764
Antenna height: 1 m
Ground elevation: 55 m

Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

See Also

`arrayConfig` | `siteviewer` | `txsite`

Introduced in R2019b

siteviewer

Create Site Viewer map display for visualizing sites

Description

Use the `siteviewer` object to create a map viewer for visualizing transmitter and receiver sites.

Note

- Site Viewer is a 3-D map display and requires hardware graphics support for WebGL™.
- This object only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

Creation

Syntax

```
viewer = siteviewer
viewer = siteviewer(Name,Value)
```

Description

`viewer = siteviewer` creates a “Site Viewer” map display for visualizing transmitter or receiver sites.

`viewer = siteviewer(Name,Value)` creates a Site Viewer map display with properties specified by one or more name-value pairs. Properties you do not specify retain their default values.

Properties

`Name` — Caption to display on map viewer window
`'Site Viewer'` (default) | character vector | string scalar

Caption to display on map viewer window, specified as a character vector or a string scalar.

Data Types: `char` | `string`

`Position` — Size and location of map viewer window in pixels
four-element integer-valued vector

Size and location of map viewer window in pixels, specified as a four-element integer-valued vector in the form `[left bottom width height]`. The default value depends on the screen resolution such that the window lies in the center of the screen with a width of 800 pixels and a height of 600 pixels.

Data Types: `double`

Basemap — Map imagery used to visualize sites
'satellite' (default) | 'streets' | 'openstreetmap' | 'darkwater' | 'grayland' | 'bluegreen' | 'colorterrain' | 'grayterrain' | 'landcover'

Map imagery used to visualize sites, specified as a one of the following:

- 'satellite' - Satellite imagery provided by ESRI
- 'streets' - Street maps provided by ESRI.
- 'openstreetmap' - Street maps provided by OpenStreetMap.
- 'darkwater' - Two-tone map with light gray for land and dark gray for water.
- 'grayland' - Two-tone map with gray for land and white for water.
- 'bluegreen' - Two-tone map with green for land and blue for water.
- 'colorterrain' - Shaded relief map derived from elevation and climate.
- 'grayterrain' - Shaded relief map in shades of gray.
- 'landcover' - Shaded relief map derived from satellite data.

Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks®.

Data Types: char | string

Terrain — Data on which to visualize sites and perform terrain calculations
'gmted2010' (default) | 'none' | character vector | scalar

Data on which to visualize sites and perform terrain calculations, specified as a character vector or a scalar previously added using addCustomTerrain or one of the following options:

- 'none' - Terrain elevation is 0 everywhere.
- 'gmted2010' - USGS GMTED2010 terrain data. This option requires an internet connection.

This property is read-only once the Site Viewer is created.

For limitations, see “Limitations” (Antenna Toolbox).

Data Types: char | string

Buildings — Name of OpenStreetMap (.osm) file to use as buildings data
string scalar | character vector

Name of the OpenStreetMap (.osm) file to use as buildings data, specified as a string scalar or a character vector. The file must be in the current directory, in a directory on the MATLAB path. You can also use a full or relative path to the file to specify the data. By default, this value is empty.

This property is read-only once the Site Viewer is created.

For limitations, see “Limitations” (Antenna Toolbox).

Data Types: char | string

Object Functions

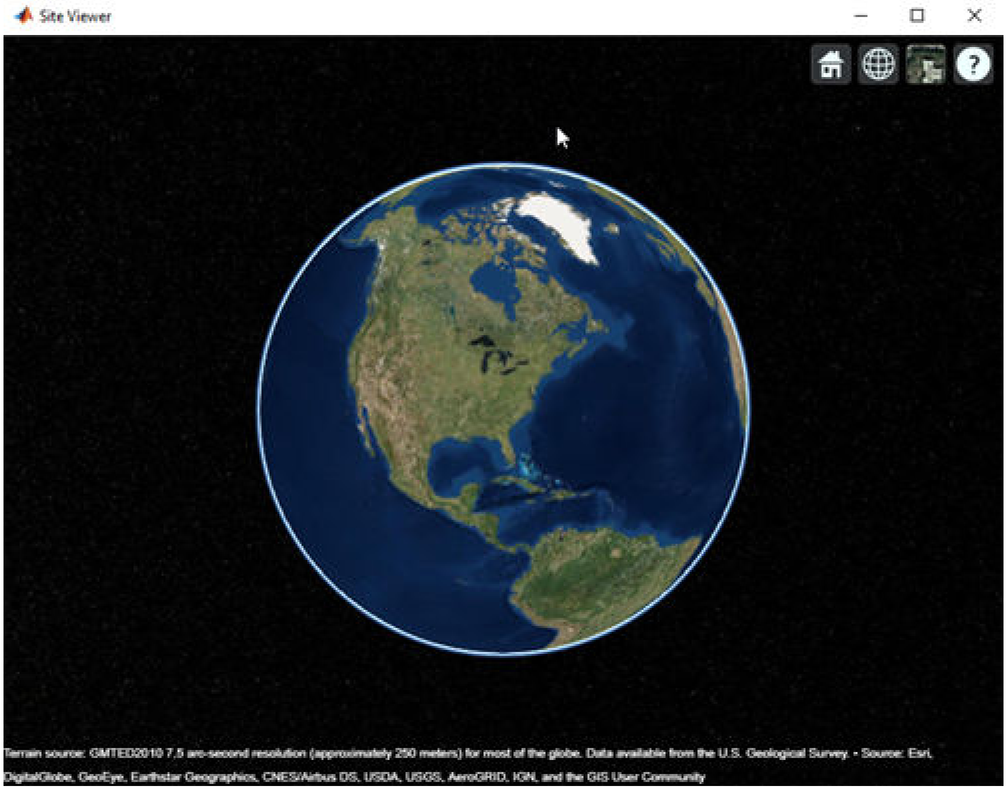
clearMap	Clear map visualizations
close	Close map viewer window

Examples

Default Site Viewer Map Display

Create a default Site Viewer map display.

```
viewer = siteviewer;
```

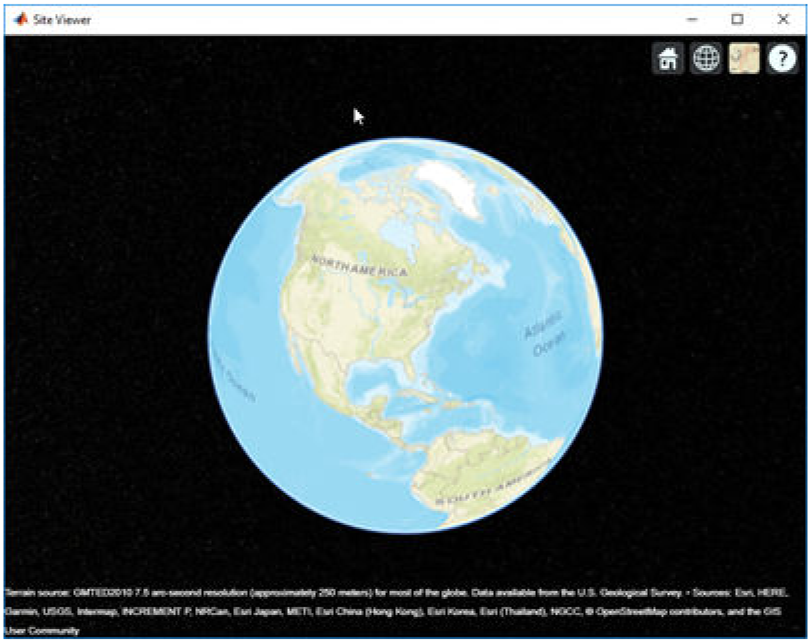


Terrain source: GMTED2010 7,5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

View Transmitter Site On Site Viewer

Launch a Site Viewer with `streets` basemap.

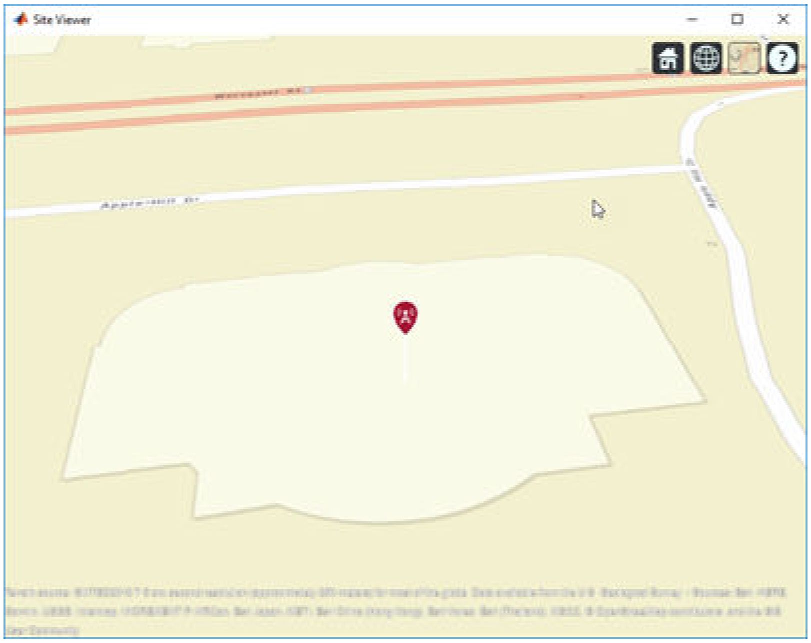
```
viewer = siteviewer("Basemap","streets");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Sources: Esri, HERE, Garmin, USGS, Intermap, INCREMENT P, NRCan, Esri Japan, METI, Esri China (Hong Kong), Esri Korea, Esri (Thailand), NGCC, © OpenStreetMap contributors, and the GIS User Community

View a transmitter site on this map.

```
tx = txsite;
show(tx)
```

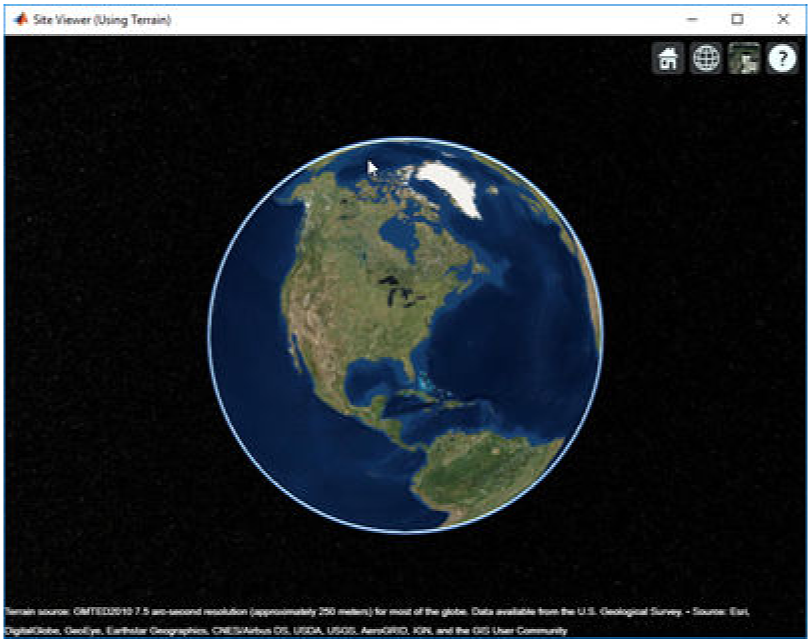


Compare Coverage Maps

Launch two Site Viewer windows.

One Site Viewer window uses the terrain model.

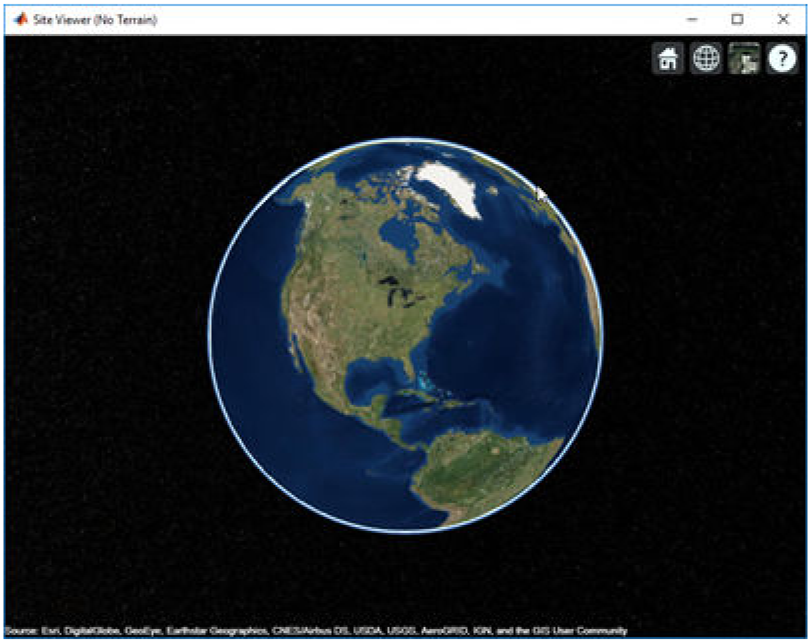
```
viewer1 = siteviewer("Terrain","gmted2010","Name","Site Viewer (Using Terrain)");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

The second Site Viewer window does not use the terrain model.

```
viewer2 = siteviewer("Terrain","none","Name","Site Viewer (No Terrain)");
```



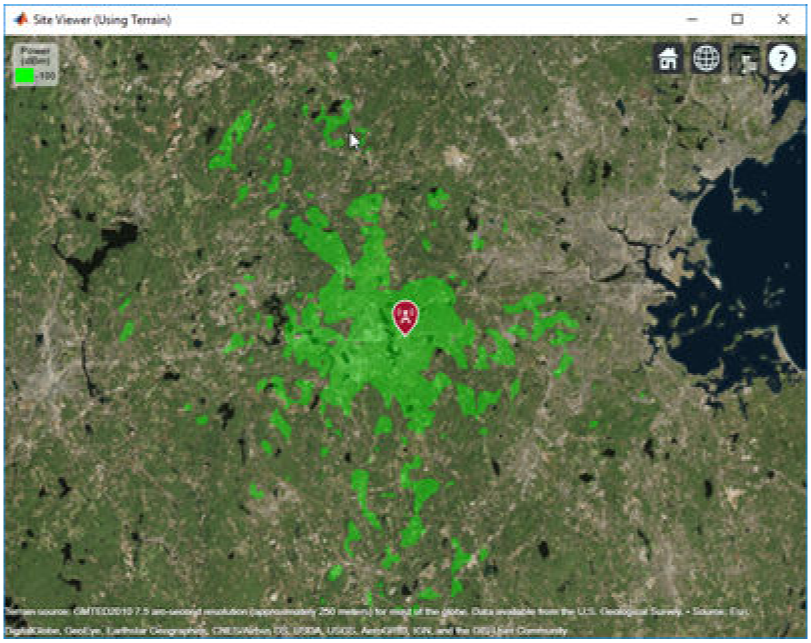
Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a transmitter site.

```
tx = txsite;
```

Generate a coverage map on each window. The map with terrain uses the Longley-Rice propagation model by default.

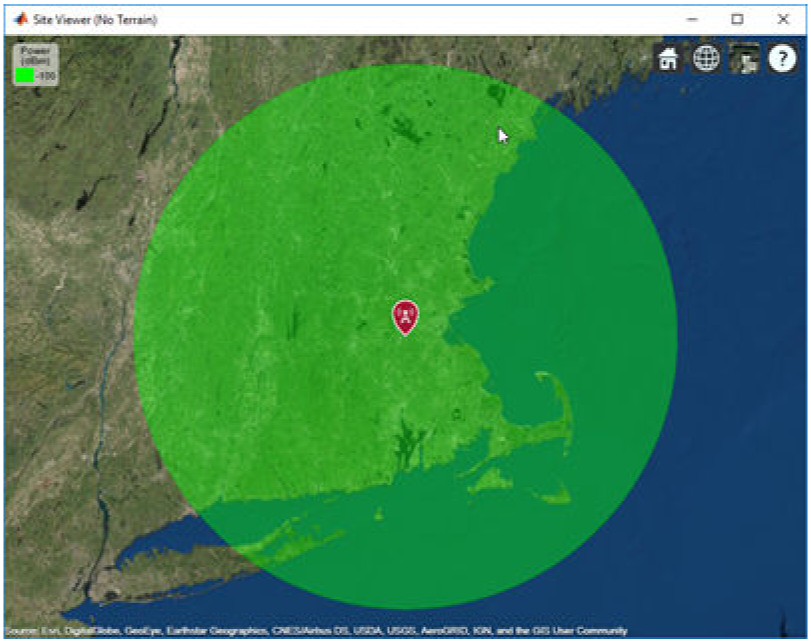
```
coverage(tx,"Map",viewer1)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

The map without terrain uses the free-space model by default.

```
coverage(tx,"Map",viewer2)
```



Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

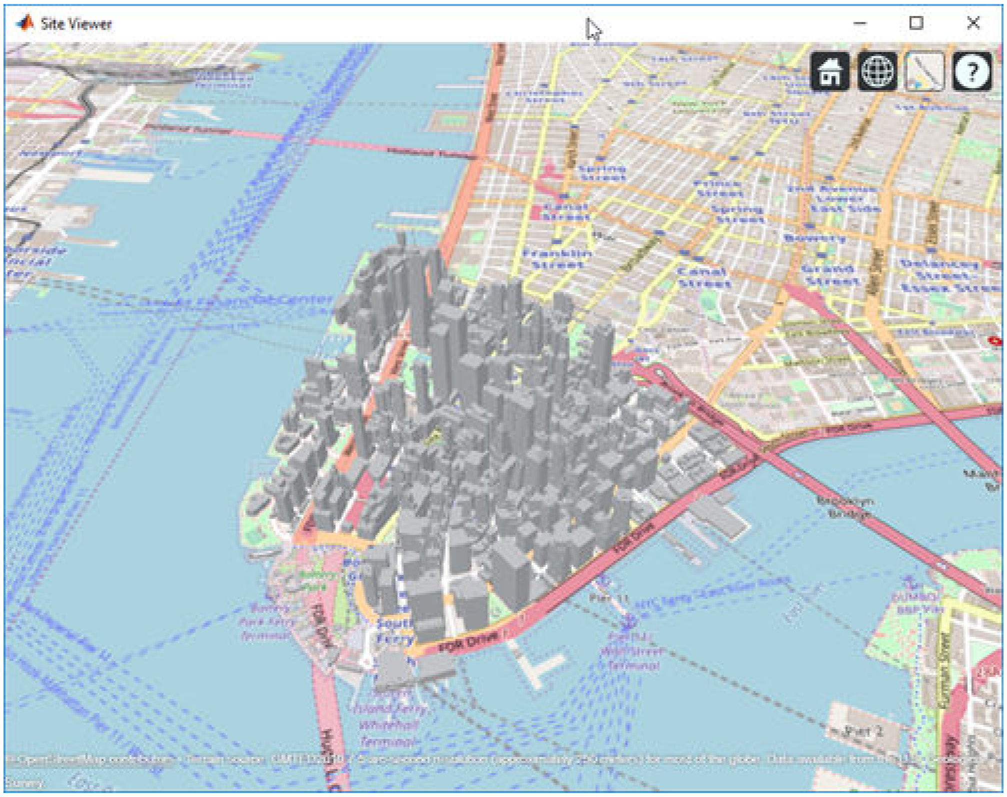
Close the maps.

```
close(viewer1)
close(viewer2)
```

Site Viewer with Buildings

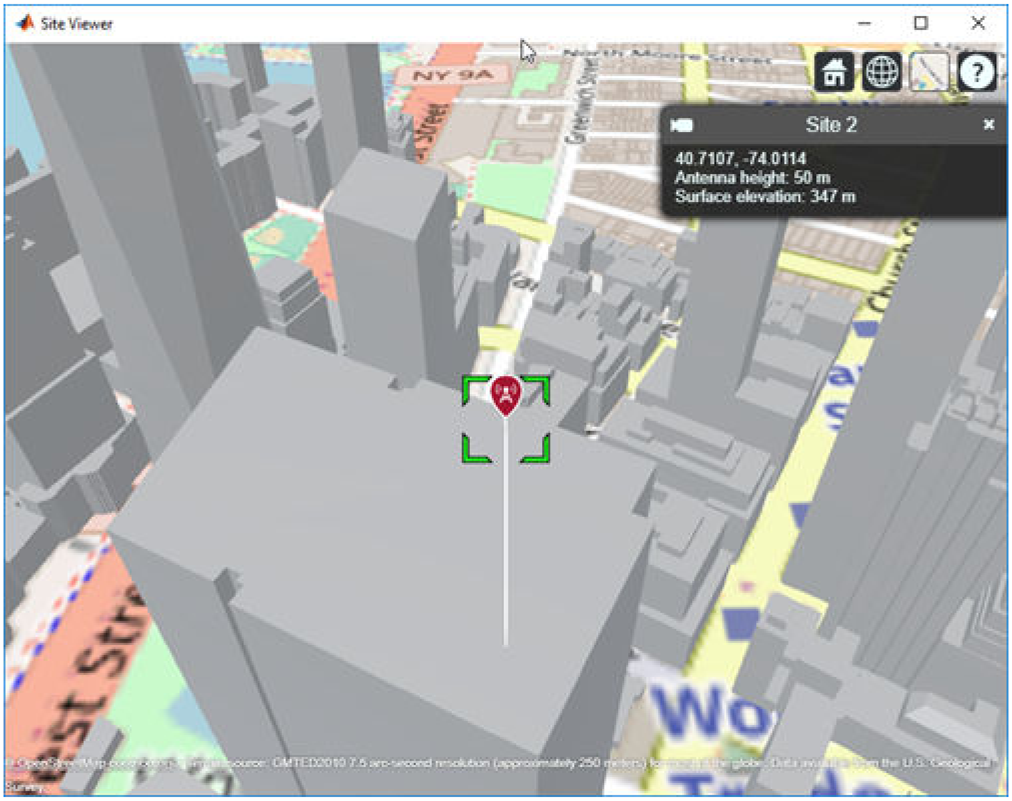
Launch siteviewer map window with basemap and buildings file for Manhattan. For more information about the osm file, see [1] on page 3-0 .

```
viewer = siteviewer("Basemap","openstreetmap",...
       "Buildings","manhattan.osm");
```



Show a transmitter site on a building.

```
tx = txsite("Latitude",40.7107,...
       "Longitude",-74.0114,...
       "AntennaHeight",50);
show(tx)
```



Site 2

40.7107, -74.0114
Antenna height: 50 m
Surface elevation: 347 m

© OpenStreetMap contributors, [illegible] source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) [illegible] the globe. Data available from the U.S. Geological Survey.

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Add and Remove a Custom Basemap

Add a custom basemap to view locations on an OpenTopoMap® basemap, then remove the custom basemap from `siteviewer`.

Initialize simulation variables to:

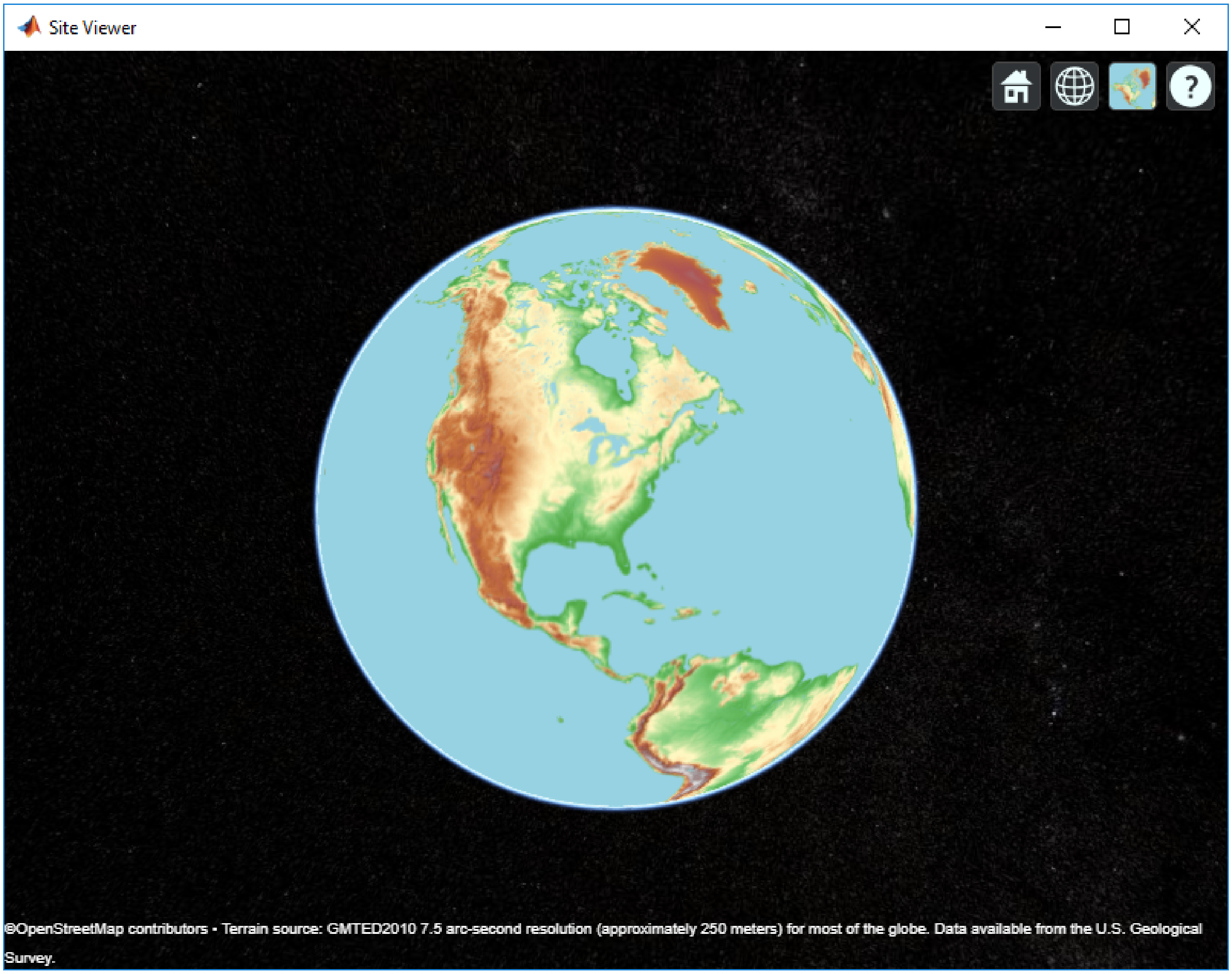
- Define the name that you will use to specify your custom basemap.
- Specify the website that provides the map data. The first character of the URL indicates which server to use to get the data. For load balancing, the provider has three servers that you can use: a, b, or c.
- Create an attribution to display on the map that gives credit to the provider of the map data. Web map providers might define specific requirements for the attribution.
- Define a display name for the custom map.

```
name = 'opentopomap';
url = 'a.tile.opentopomap.org';
copyright = char(uint8(169));
```

```
attribution = copyright + "OpenStreetMap contributors";
displayName = 'Open Topo Map';
```

Use `addCustomBasemap` to load the custom basemap, and then create a `siteviewer` object that loads the custom basemap.

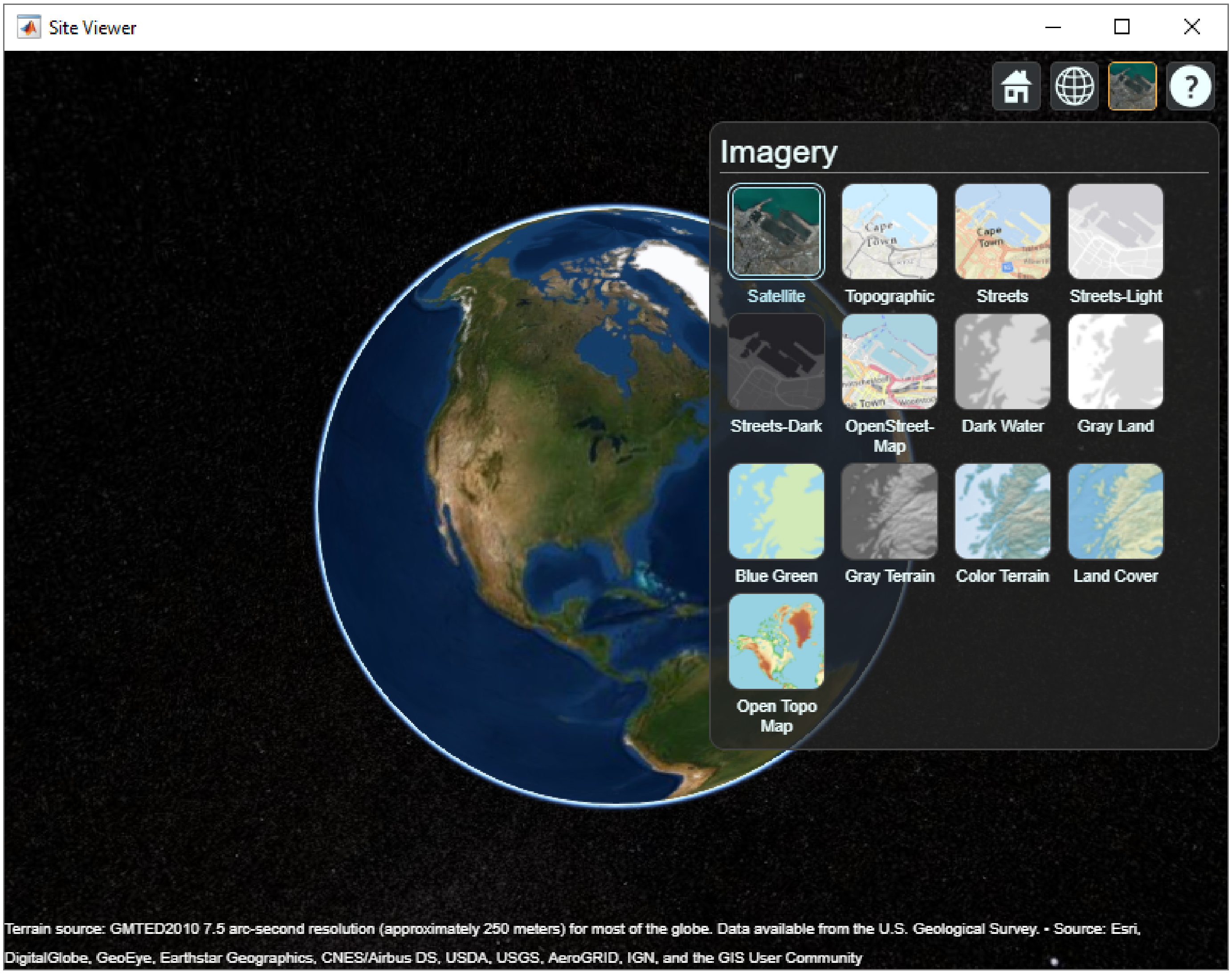
```
addCustomBasemap(name,url,'Attribution',attribution','DisplayName',displayName)
viewer = siteviewer('Basemap',name);
```



©OpenStreetMap contributors • Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey.

After a custom basemap is added to `siteviewer`, the custom map is available for future calls to `siteviewer`. Note the `'Open Topo Map'` icon in the `Imagery` tab.

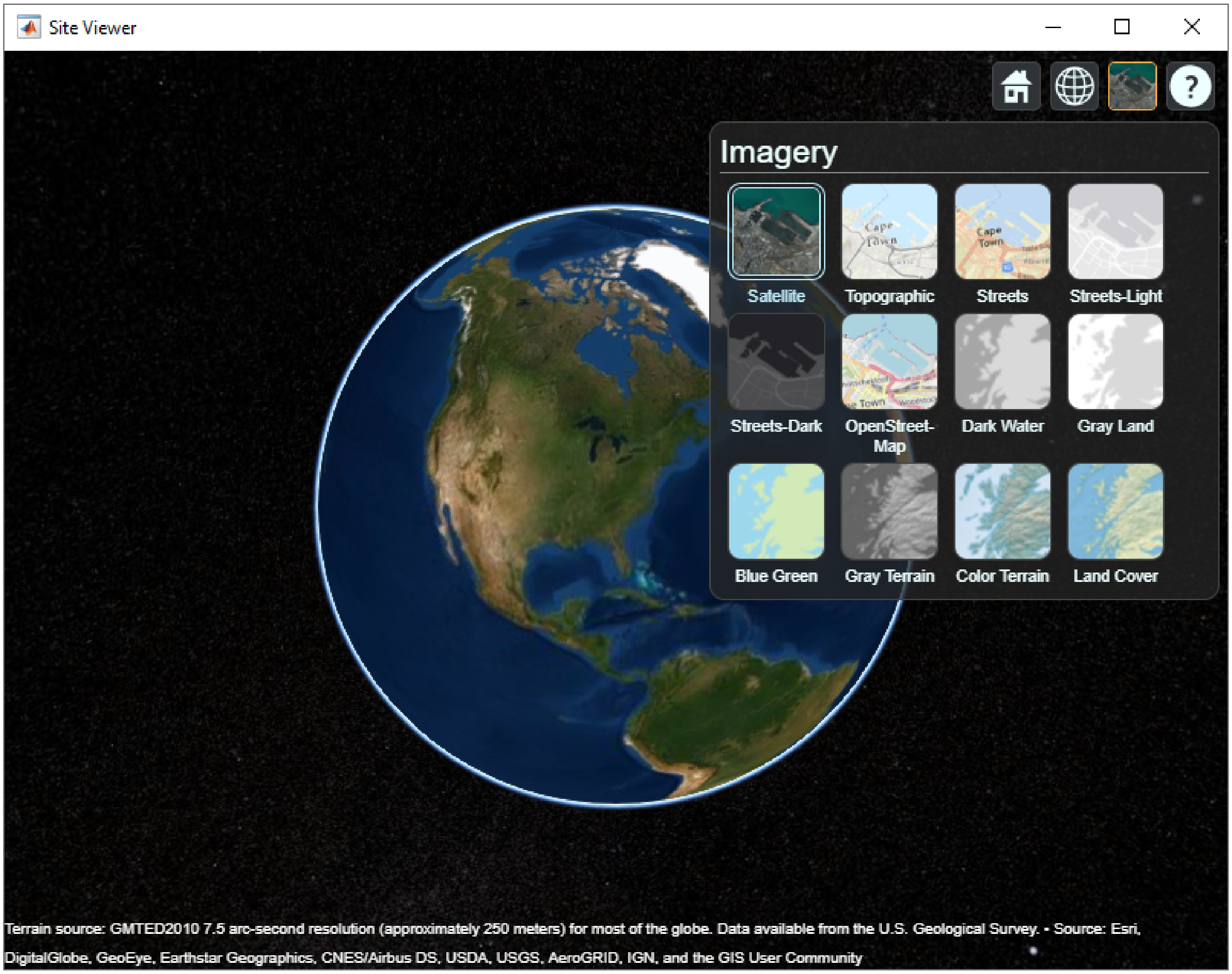
```
siteviewer;
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Use `removeCustomBasemap` to remove the custom basemap from future calls to `siteviewer`. Note the `'Open Topo Map'` icon is no longer available in the `Imagery` tab.

```
removeCustomBasemap(name)
siteviewer;
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Limitations

Terrain

- Default terrain access requires Internet connection. If no internet connection exists, then Site Viewer automatically uses `'none'` in the property `Terrain`.
- Custom DTED terrain files for use with `addCustomTerrain` must be acquired outside of MATLAB for example by using USGS EarthExplorer.
- When using custom terrain, analysis is restricted to the terrain region. For example, an error occurs if trying to show a txsite or rxsite outside of the region.

Buildings

- OpenStreetMap files obtained from https://www.openstreetmap.org represent crowd-sourced map data, and the completeness and accuracy of the buildings data may vary depending on the map location.

- When downloading data from https://www.openstreetmap.org, select an export area larger than the desired area to ensure that all expected building features are fully captured. Building features at the edge of the selected export area may be missing.
- Building geometry and features are interpreted from the file according to the recommendations of OpenStreetMap for 3D buildings.

See Also

`addCustomBasemap` | `addCustomTerrain` | `removeCustomBasemap` | `removeCustomTerrain` | `rxsite` | `txsite`

Topics

"Site Viewer"

Introduced in R2019b

txsite

Create radio frequency transmitter site

Description

Use the `txsite` object to create a radio frequency transmitter site.

Creation

Syntax

```
tx = txsite
tx = txsite(coordsys)
tx = txsite( ___ ,Name,Value)
```

Description

`tx = txsite` creates a radio frequency transmitter site.

`tx = txsite(coordsys)` creates a transmitter site with the specified coordinate system. You can specify either `'geographic'` or `'cartesian'` coordinate system.

`tx = txsite( ___ ,Name,Value)` sets "Properties" (Antenna Toolbox) using one or more name-value pairs. For example, `tx = txsite('Name','TX Site')` creates a transmitter site with the name `TX Site`. Enclose each property name in quotes.

You can create multiple transmitter sites by using `Name`, `Latitude`, and `Longitude` properties. For example: `names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"]; lats = [42.3467,42.3598,42.3763]; lons = [-71.0972,-71.0545,-71.0611];`. The `CoordinateSystem` property must be a string scalar or a character vector.

Properties

`Name` — Site name
character vector | string | row or column vector

Site name, specified as a character vector or string or as a row or column vector of *N* elements. Specifying name as a row or column vector creates multiple sites.

Example: `'Name','Site 2'`

Example: `tx.Name = 'Fenway Park'`

Example: `names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"]; tx = txsite('Name',names)`

Data Types: `char` | `string`

`CoordinateSystem` — Coordinate system used to site location
`'geographic'` (default) | `'cartesian'`

Coordinate system used to the site location, specified as `'geographic'` or `'cartesian'`. If you specify `'geographic'`, the site location is defined using the `Latitude`, `Longitude`, and `AntennaHeight` properties. If you specify `'cartesian'`, the site location is defined using the `AntennaPosition` property.

Example: `'CoordinateSystem'`,`'cartesian'`

Example: `tx.CoordinateSystem = 'cartesian'`

Latitude — Site latitude coordinates

`42.3001` (default) | numeric scalar in the range `[-90 90]` | row or column vector

Site latitude coordinates, specified as a numeric scalar in the range of `-90` to `90`, or as a row or column vector of *N* elements in the range `[-90 90]`. Specifying latitude as a row or column vector creates multiple sites. The coordinates are defined using the world geodesic system of 1984 (WGS-84) reference ellipsoid. Latitude specifies north-south position.

Example: `'Latitude'`,`45.098`

Example: `tx.Latitude = 45.098`

Example: `latitude = [42.3467,42.3598,42.3763]; tx = txsite('Latitude',latitude)`

Dependencies

To enable this property, set `CoordinateSystem` to `'geographic'`.

Longitude — Site longitude coordinates

`-71.3504` (default) | numeric scalar in the range `[-180 180]` | row or column vector

Site longitude coordinates, specified as a numeric scalar in the range `[-180 180]`or as a row or column vector of *N* elements in the range `[-180 180]`. Specifying longitude as a row or column vector creates multiple sites. The coordinates are defined using the world geodesic system of 1984 (WGS-84) reference ellipsoid. Longitude specifies the east-west the position.

Example: `'Longitude'`,`-68.890`

Example: `tx.Longitude = -71.0972`

Example: `longitude = [-71.0972,-71.0545,-71.0611]; tx = txsite('Longitude',longitude)`

Dependencies

To enable this property, set the `CoordinateSystem` to `'geographic'`.

Antenna — Antenna element or array

`'isotropic'` (default) | object | row vector

Antenna element or array specified as one of these:

- `'isotropic'` to model an antenna that radiates uniformly in all directions
- An `arrayConfig` object
- An antenna element object from the “Antenna Catalog” (Antenna Toolbox)

Note When using the antenna element object, use the `design` function to design the antenna at the required transmit frequency. Then add this antenna element to the transmitter site.

- Any antenna object in “Antennas, Microphones, and Sonar Transducers” (Phased Array System Toolbox), if you have Phased Array System Toolbox
- Any array object in “Array Geometries and Analysis” (Phased Array System Toolbox), if you have Phased Array System Toolbox

Example: `'Antenna'`,`cfgArray`, where `cfgArray` is an `arrayConfig` object

Example: `tx.Antenna = arrayConfig('Size',[8 1]);` specifies an 8-element ULA along z-axis

`AntennaAngle` — Antenna X-axis angle

`0` (default) | numeric scalar | 2-by-1 vector | 2-by-*N* matrix

Antenna X-axis angle defined with reference to a local Cartesian coordinate system, specified as a numeric scalar representing an azimuth angle in degrees or as a 2-by-1 vector or a 2-by-*N* matrix representing both azimuth and elevation angles with each element unit in degrees.

The azimuth angle is measured counterclockwise from the east along the antenna X-axis (for geographical sites) or from the global X-axis around the global Z-axis (for Cartesian sites). Specify the azimuth angle between `-180` to `180`. degrees.

The elevation angle is measured from antenna X-axis along the horizontal or XY plane. Specify the elevation angle between`-90` to `90` degrees.

Example: `'AntennaAngle'`,25

Example: `tx.AntennaAngle = [25,-80]`

`AntennaHeight` — Antenna height above surface

`10` (default) | nonnegative numeric scalar | row vector

Antenna height from the ground or building surface, specified as a nonnegative numeric scalar in meters. Maximum value for this property is 6,371,000 m.

If the site location coincides with the building location, the antenna height is measured from the top of the building to the center of the antenna. Otherwise,the height is measured from ground elevation to the center of the antenna.

Example: `'AntennaHeight'`,25

Example: `tx.AntennaHeight = 15`

Dependencies

To enable this property, set `CoordinateSystem` to `'geographic'`.

Data Types:

`AntennaPosition` — Position of antenna center

`[0;0;0]` (default) | 3-by-1 vector

Position of the antenna center, specified as a 3-by-1 vector representing X-, Y-, and Z-axis Cartesian coordinates with each element in meters.

Example: `'AntennaPosition'`,`[0;2;4]`

Example: `tx.AntennaPosition = [0;2;4]`

Dependencies

To enable this property, set `CoordinateSystem` to `'cartesian'`.

Data Types:

`SystemLoss` — System loss

`0` (default) | nonnegative scalar | row vector

System loss, specified as a nonnegative scalar in dB.

System loss includes transmission line loss and any other miscellaneous system losses.

Example: `'SystemLoss',10`

Example: `txsite.SystemLoss = 10`

Data Types:

`TransmitterFrequency` — Transmitter operating frequency

`1.9000e+09` (default) | positive scalar | row vector

Transmitter operating frequency, specified as a positive scalar in Hz. in the range `[1e3 200e9`.

Example: `'TransmitterFrequency',30e9`

Example: `txsite.TransmitterFrequency = 30e9`

Data Types: `double`

`TransmitterPower` — Signal power at transmitter output

`10` (default) | positive scalar

Signal power at transmitter output, specified as a positive scalar in watts. The transmitter out is connected to the antenna.

Example: `'TransmitterPower',30`

Example: `txsite.TransmitterPower = 30`

Data Types: `double`

Object Functions

show	Show site location on map
hide	Hide site location on map
distance	Distance between sites
angle	Angle between sites
elevation	Elevation of site
location	Location coordinates at a given distance and angle from site
los	Plot or compute the line-of-sight (LOS) visibility between sites on a map
coverage	Display coverage map
sinr	Display signal-to-interference-plus-noise ratio (SINR) map
pattern	Plot antenna radiation pattern on map

Examples

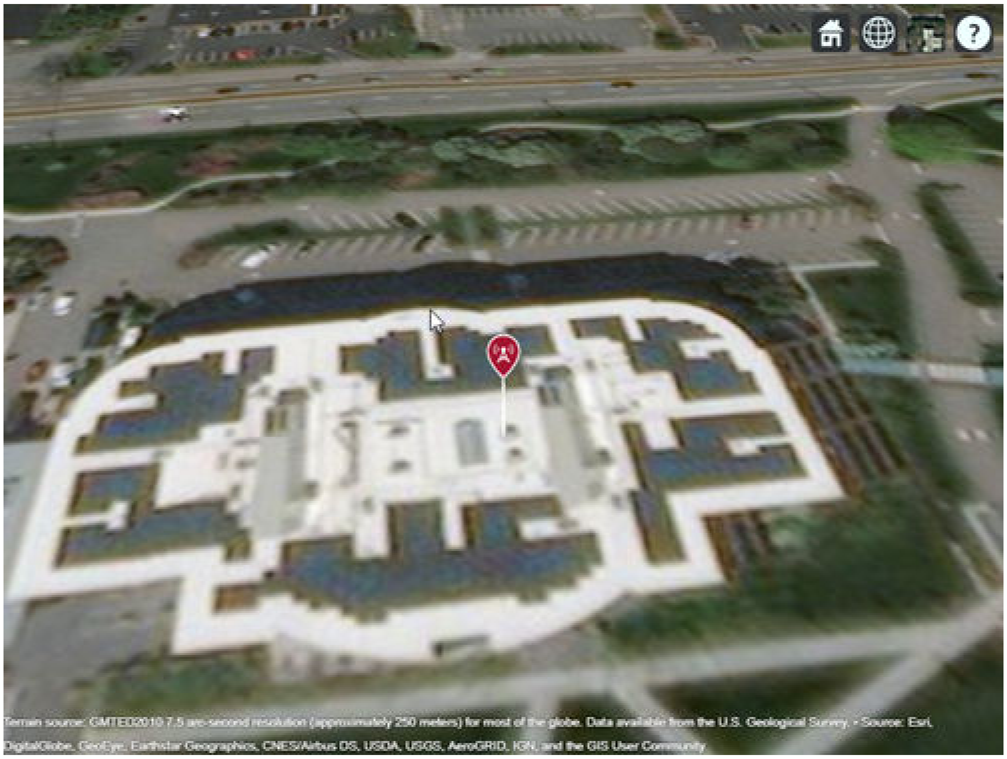
Default Transmitter Site

Create and view a transmitter site at a latitude of 42.3001 and a longitude of -71.3504.

```
tx = txsite('Name','MathWorks Apple Hill','Latitude',42.3001,...
    'Longitude',-71.3504)
```

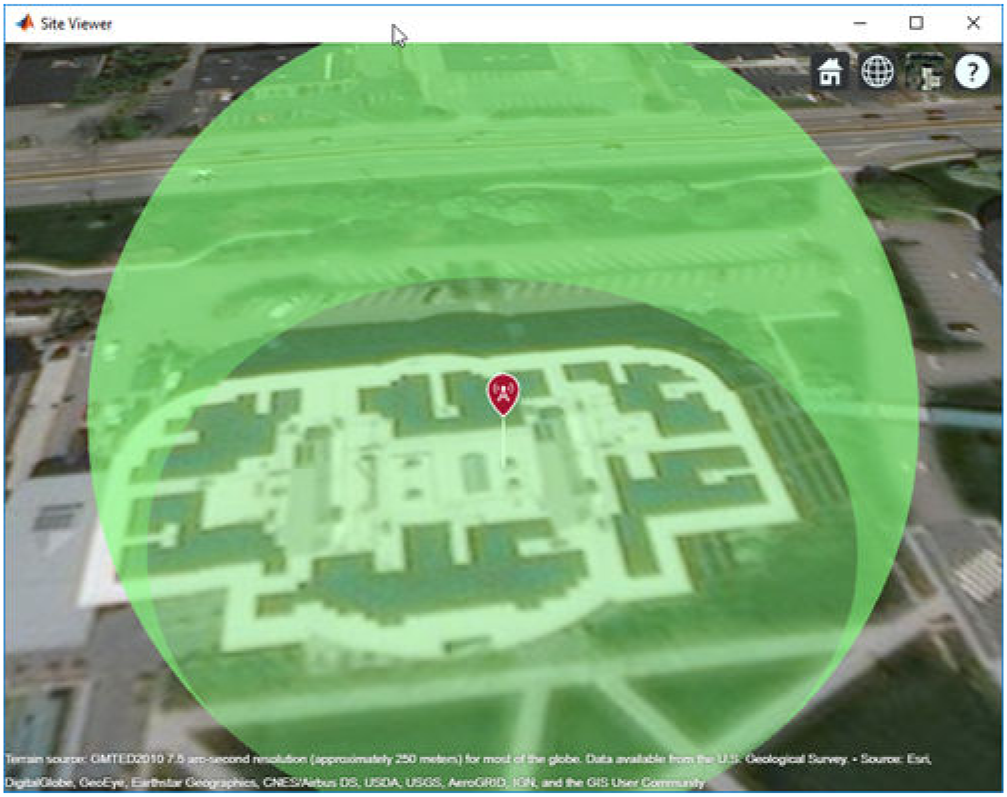
```
tx = 
  txsite with properties:

                   Name: 'MathWorks Apple Hill'
               Latitude: 42.3001
              Longitude: -71.3504
                Antenna: 'isotropic'
           AntennaAngle: 0
          AntennaHeight: 10
             SystemLoss: 0
    TransmitterFrequency: 1.9000e+09
       TransmitterPower: 10
```

```
show(tx)
```


Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

View the coverage of the antenna.

```
pattern(tx)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

See Also

`arrayConfig` | `rxsite` | `siteviewer`

Introduced in R2019b

propagationData

Create RF propagation data container

Description

Use the `propagationData` object to import and visualize geolocated propagation data. The measurement data can be path loss data, signal strength measurements, signal-to-noise-ratio (SNR) data, or cellular information.

Creation

Syntax

```
pd = propagationData(filename)
pd = propagationData(table)

pd = propagationData(latitude,longitude,varname,varvalue)
pd = propagationData( ___ ,Name,Value)
```

Description

`pd = propagationData(filename)` creates a propagation data container object by reading data from a file specified by `filename`.

`pd = propagationData(table)` creates a propagation data container object from a table object specified by `table`.

`pd = propagationData(latitude,longitude,varname,varvalue)` creates a propagation data container object using `latitude` and `longitude` coordinates with data specified using `varname` and `varvalue`.

`pd = propagationData( ___ ,Name,Value)` sets properties using one or more name-value pairs. Enclose each property name in quotes.

Input Arguments

`filename` — Name of file containing propagation data
character vector | string scalar

Name of the file containing propagation data, specified as a character vector or a string scalar. The file must be in the current directory, in a directory on the MATLAB path, or be specified using a full or relative path. The file must be compatible with the `readtable` function. Call the `readtable` function if customized parameters are required to import the file and then pass the `table` object to the `propagationData` object.

Propagation data in the file must have one variable corresponding to the latitude values, one variable corresponding to longitude values, and at least one variable containing numeric data.

Data Types: `string` | `char`

`table` — Table containing propagation data
table object

Table containing propagation data, specified as a table object.

Propagation data in the file must have one variable corresponding to the latitude values, one variable corresponding to longitude values, and at least one variable containing numeric data.

Data Types: `table`

`latitude` — Latitude coordinate values
vector

Latitude coordinate values, specified as a vector in decimal degrees with reference to Earth's ellipsoid model WGS-84. The latitude coordinates must be in the range `[-90 90]`.

Data Types: `double`

`longitude` — Longitude coordinate values
vector

Longitude coordinate values, specified as a vector in decimal degrees with reference to earth's ellipsoid. model WGS-84.

Data Types: `double`

`varname` — Variable name
character vector | string scalar

Variable name, specified as a character vector or a string scalar. This variable name must correspond to the variable with numeric data other than latitude or longitude. The variable name and the corresponding values are stored as a column in the Data property table object.

Data Types: `string` | `char`

`varvalue` — Variable values
numeric vector

Variable values, specified as a numeric vector. The numeric vectors must be the same size as latitude and longitude. The variable name and corresponding values are stored as a column in the Data property table object.

Data Types: `double`

Output Arguments

pd — Propagation data
`propagationData` object

Propagation data, returned as a `propagationData` object.

Properties

Name — Propagation data name
`'Propagation Data'` (default) | character vector | string scalar

Propagation data name, specified as a character vector or string scalar.

Example: 'Name','propdata'

Example: pd.Name = 'propdata'

Data Types: char | string

Data — Propagation data table

scalar table object

This property is read-only.

Propagation data table, specified as a scalar table object containing a column corresponding to latitude coordinates, a column corresponding to longitude coordinates, and one or more columns corresponding to associated propagation data.

Data Types: table

DataVariableName — Name of data variable to plot

character vector | string scalar

Name of the data variable to plot, specified as a character vector or string scalar corresponding to a variable name in the Data table used to create propagation data container object. The variable name must correspond to a variable with numeric data and cannot correspond to the latitude or longitude variables. The default value for this property is the name of the first numeric data variable name in the Data table that is not a latitude or longitude variable.

Data Types: char | string

Object Functions

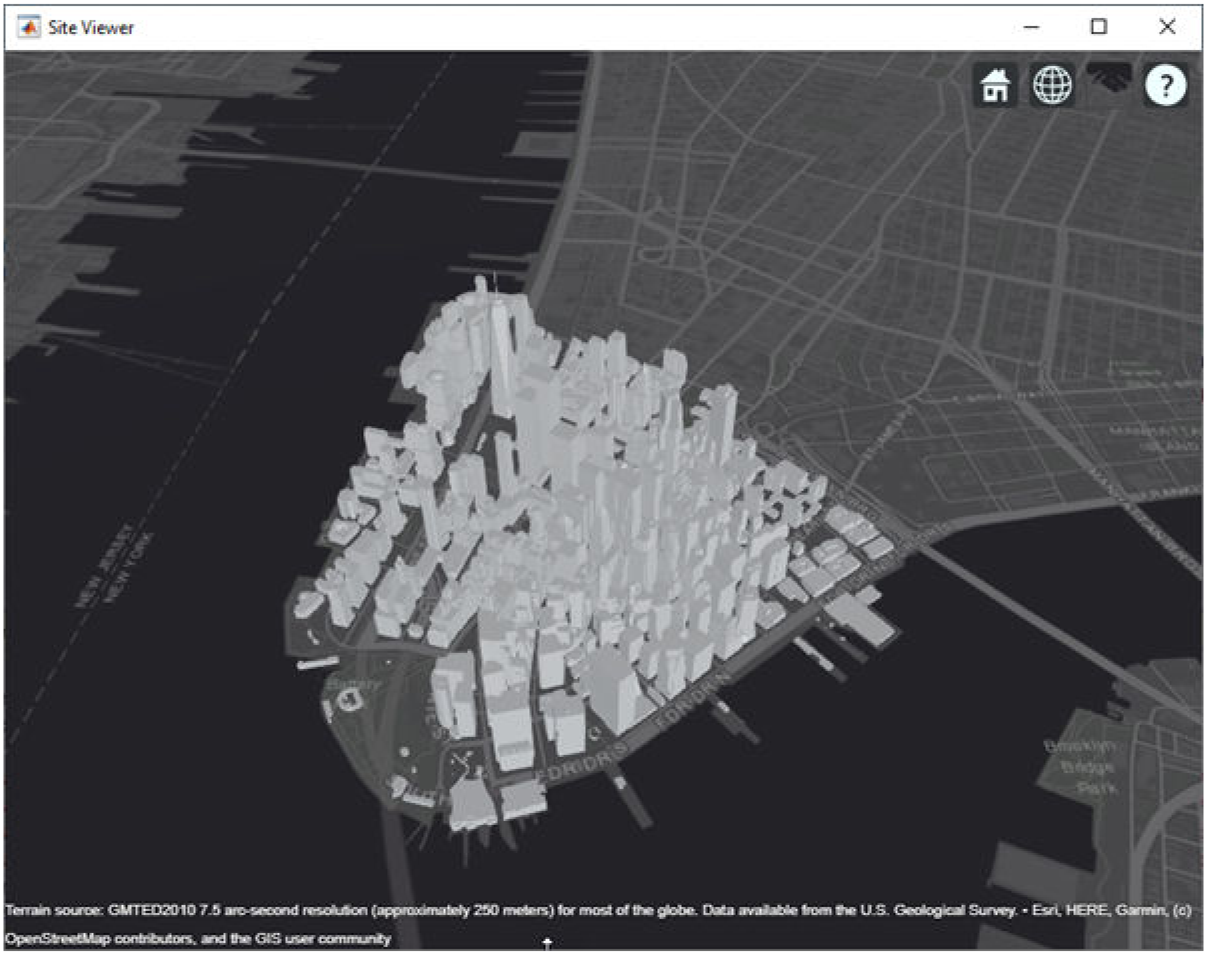
plot	Plot propagation data on map
contour	Display contour map
location	Data location coordinates
getDataVariable	Get data variable values of data points in propagation data object
interp	Geographic data interpolation

Examples

Compute Signal Strength Data in Urban Environment

Launch Site Viewer with basemaps and building files for Manhattan. For more information about the osm file, see [1] on page 3-0 .

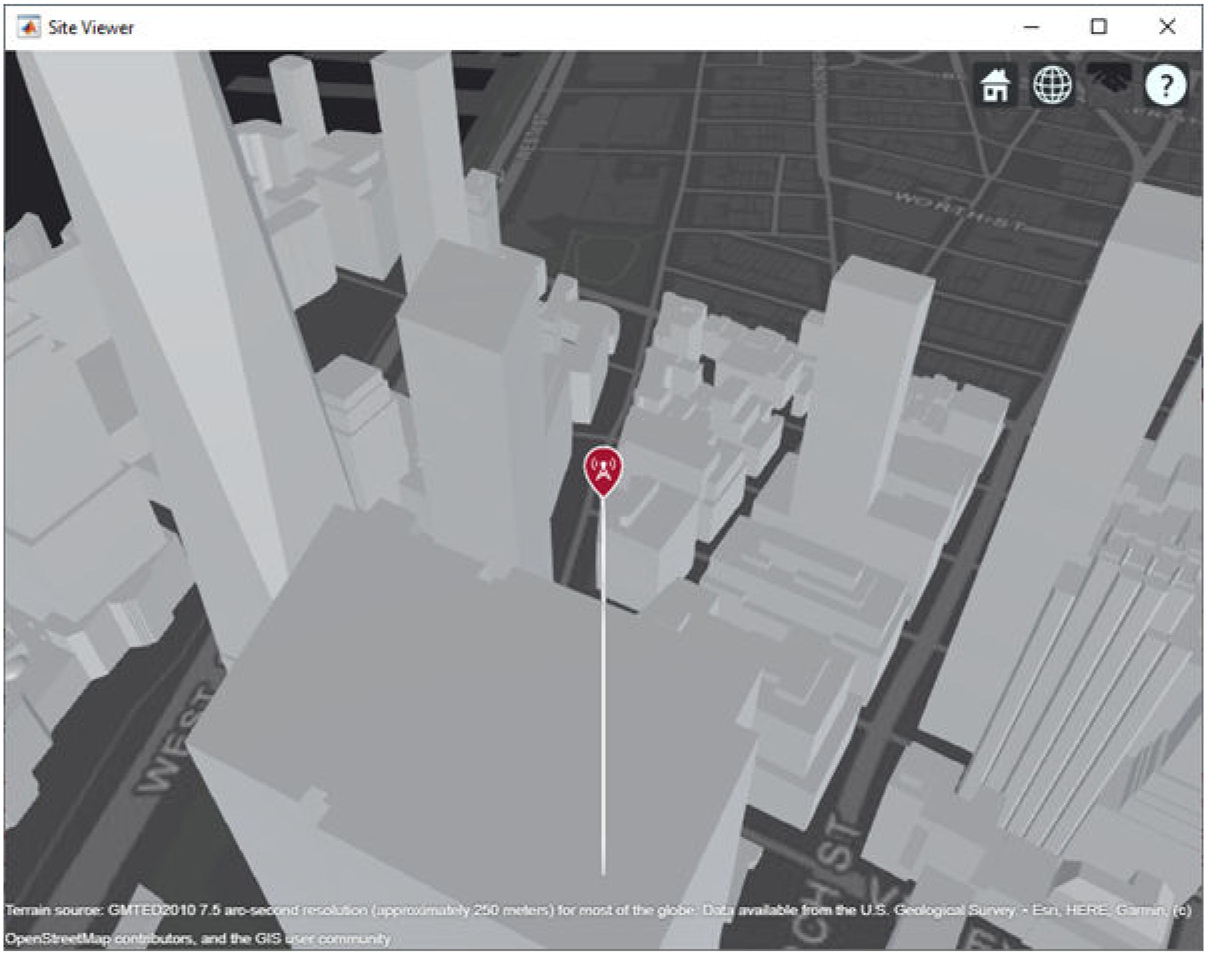
```
viewer = siteviewer("Basemap","streets_dark",...
        "Buildings","manhattan.osm");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Show a transmitter site on a building.

```
tx = txsite("Latitude",40.7107,...
        "Longitude",-74.0114,...
        "AntennaHeight",80);
show(tx)
```


Site Viewer
WORTH ST
WEST
CHURCH ST
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Create receiver sites along nearby streets.

```
latitude = [linspace(40.7088, 40.71416, 50), ...
        linspace(40.71416, 40.715505, 25), ...
        linspace(40.715505, 40.7133, 25), ...
        linspace(40.7133, 40.7143, 25)]';
longitude = [linspace(-74.0108, -74.00627, 50), ...
        linspace(-74.00627 ,-74.0092, 25), ...
        linspace(-74.0092, -74.0110, 25), ...
        linspace(-74.0110, -74.0132, 25)]';
rxs = rxsite("Latitude", latitude, "Longitude", longitude);
```

Compute signal strength at each receiver location.

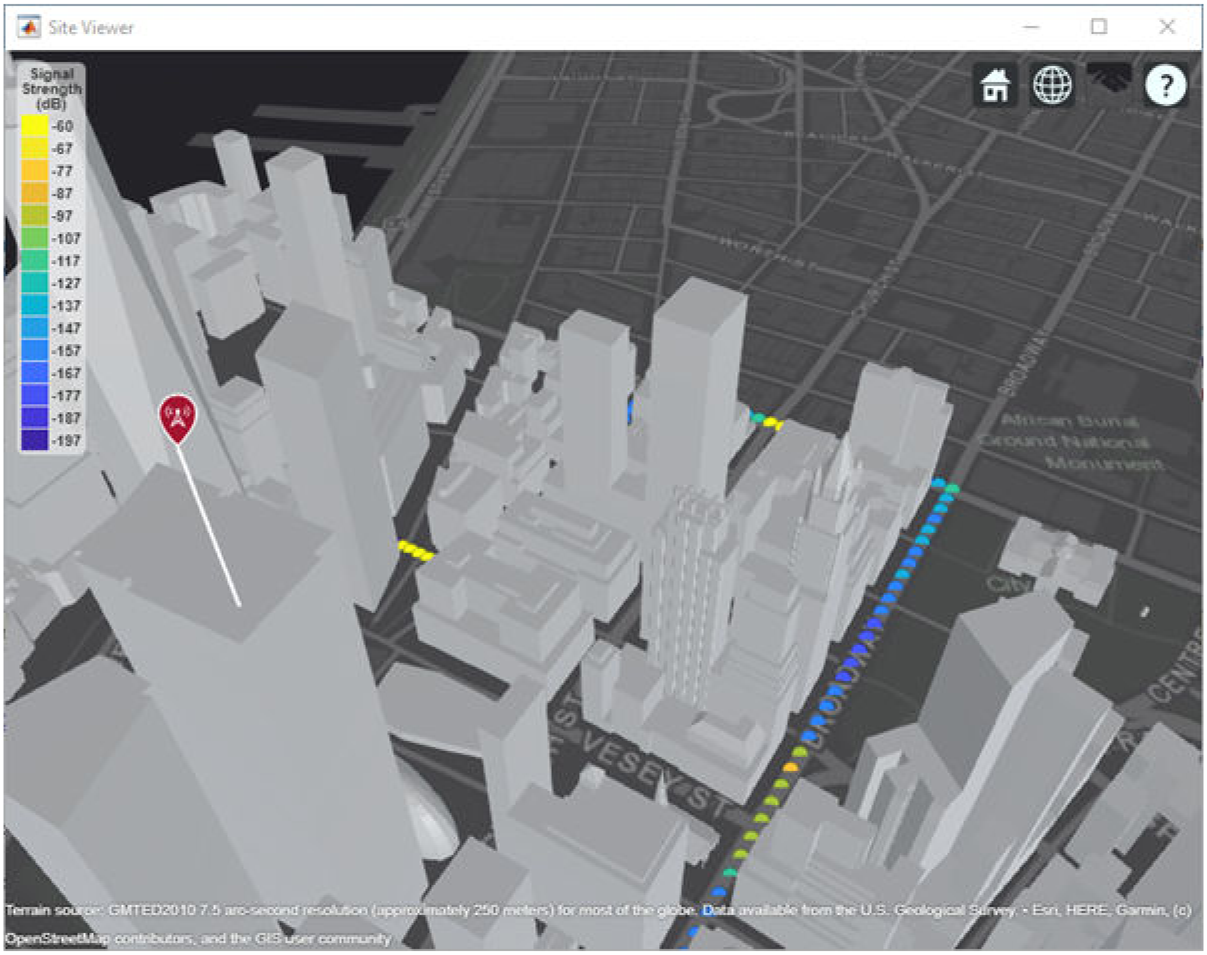
```
signalStrength = sigstrength(rxs, tx)';
```

Create a `propagationData` object to hold computed signal strength data.

```
tbl = table(latitude, longitude, signalStrength);
pd = propagationData(tbl);
```

Plot the signal strength data on a map as colored points.

```
legendTitle = "Signal" + newline + "Strength" + newline + "(dB)";
plot(pd, "LegendTitle", legendTitle, "Colormap", parula);
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

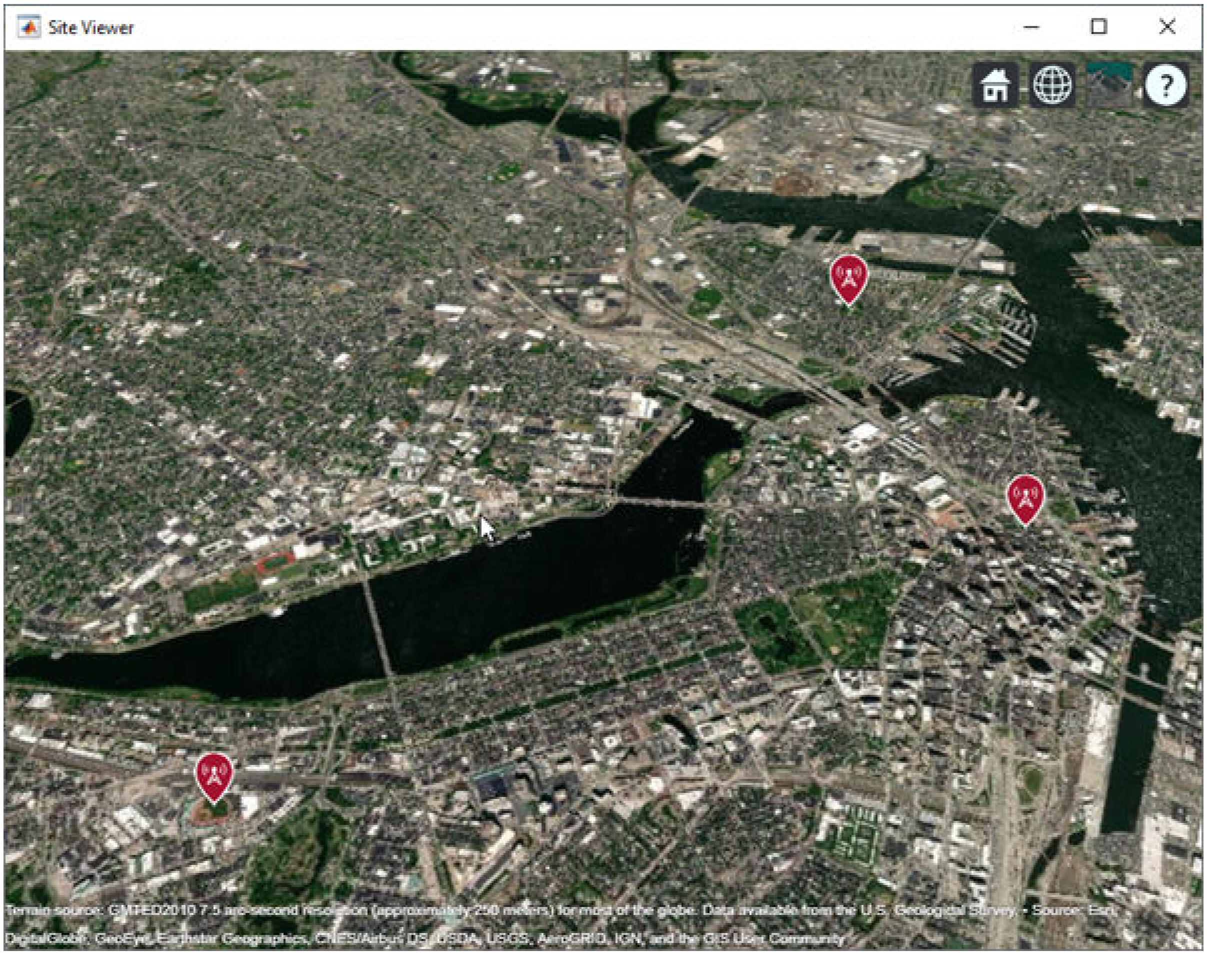
Capacity Map Using SINR Data

Define names and locations of sites around Boston.

```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

Create an array of transmitter sites.

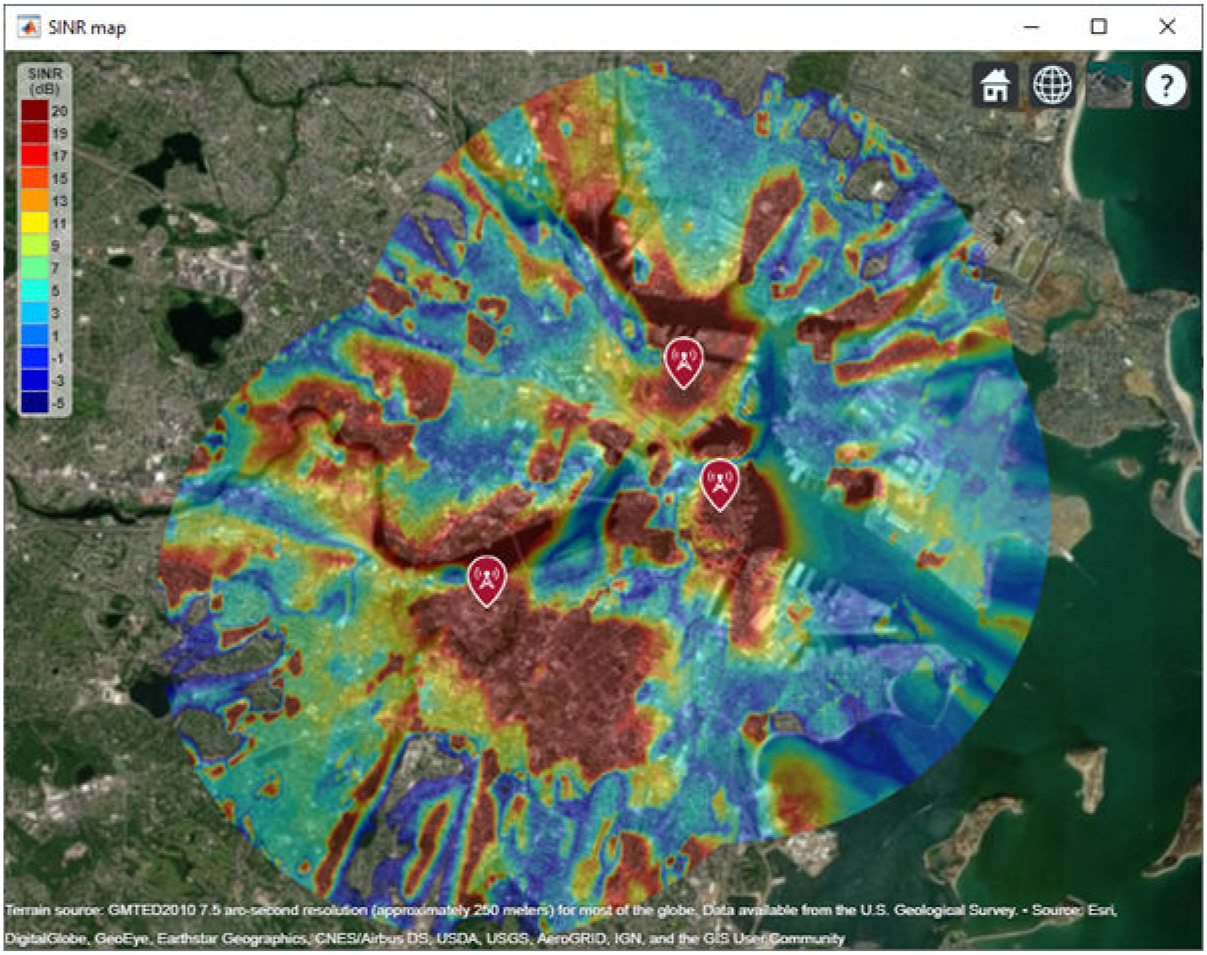
```
txs = txsite("Name",names,...
       "Latitude",lats,...
       "Longitude",lons, ...
       "TransmitterFrequency",2.5e9);
show(txs)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a signal-to-interference-plus-noise-ratio (SINR) map, where signal source for each location is selected as the transmitter site with the strongest signal.

```
sv1 = siteviewer("Name","SINR map");
sinr(txs,"MaxRange",5000)
```


SINR map
SINR (dB)
20
19
17
15
13
11
9
7
5
3
1
-1
-3
-5
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Return SINR propagation data.

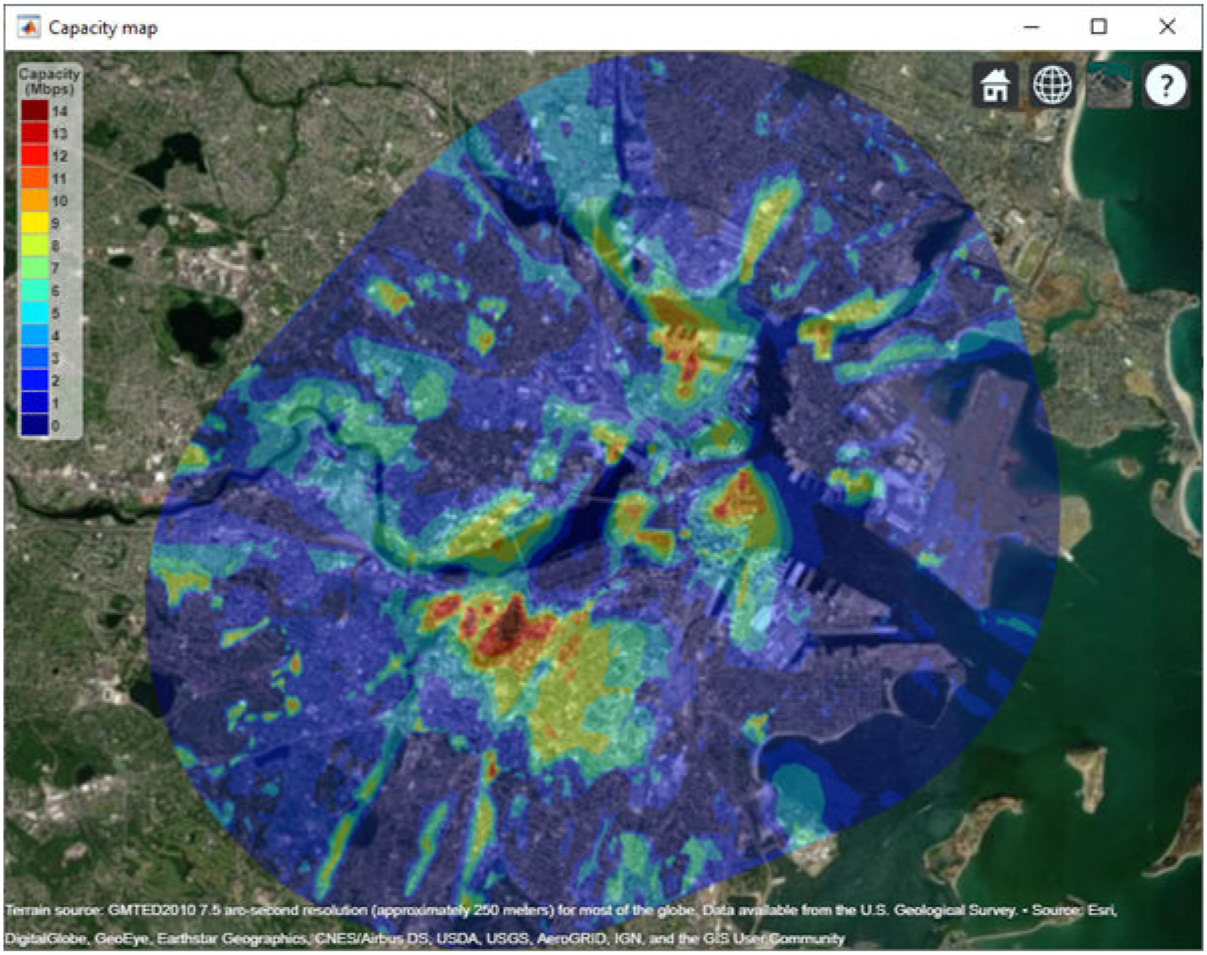
```
pd = sinr(txs,"MaxRange",5000);
[sinrDb,lats,lons] = getDataVariable(pd,"SINR");
```

Compute capacity using the Shannon-Hartley theorem.

```
bw = 1e6; % Bandwidth is 1 MHz
sinrRatio = 10.^(sinrDb./10); % Convert from dB to power ratio
capacity = bw*log2(1+sinrRatio)/1e6; % Unit: Mbps
```

Create new propagation data for the capacity map and display the contour plot.

```
pdCapacity = propagationData(lats,lons,"Capacity",capacity);
sv2 = siteviewer("Name","Capacity map");
legendTitle = "Capacity" + newline + "(Mbps)";
contour(pdCapacity,"LegendTitle",legendTitle);
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

See Also

`readtable` | `rxsite` | `siteviewer` | `txsite`

Introduced in R2020a

comm.Ray

Propagation ray container object

Description

The comm.Ray object is a container object for the properties of a propagation ray. The object contains the geometric and electromagnetic information of a radio wave propagating from one point to another point in the space.

Creation

Typically you create comm.Ray objects by using the raytrace function.

Syntax

```
ray = comm.Ray
ray = comm.Ray(Name,Value)
```

Description

ray = comm.Ray creates a container object that initializes properties for a propagation ray.

ray = comm.Ray(Name,Value) sets properties using one or more name-value pair arguments. Enclose each property name in quotes. For example, comm.Ray('CoordinateSystem','Geographic','TransmitterLocation',[40.730610,-73.935242,0]) specifies the geographic coordinate system and a transmitter located in New York City.

Properties

PathSpecification — Propagation path specification method
'Locations' (default) | 'Delay and angles'

Propagation path specification method, specified as one of these values.

- 'Locations' — The ray object path between waypoints are conveyed as (*x*, *y*, *z*) coordinate points by the TransmitterLocation, ReceiverLocation, and, if applicable, ReflectorLocations properties .
- 'Delay and angles' — The ray object path between waypoints are conveyed by the PropagationDelay, AngleOfDeparture, and AngleOfArrival properties.

Data Types: char | string

CoordinateSystem — Coordinate system
'Cartesian' (default) | 'Geographic'

Coordinate system, specified as 'Cartesian' or 'Geographic'. When you set the CoordinateSystem property to 'Geographic', the coordinate system is defined relative to the

WGS-84 Earth ellipsoid model and the object defines angles relative to the local East-North-Up (ENU) coordinate system at the transmitter and receiver.

Dependencies

To enable this property, set the `PathSpecification` property to `'Locations'`.

Data Types: `char` | `string`

`SystemScale` — Cartesian coordinate system scale

`1` (default) | positive scalar

Cartesian coordinate system scale in meters, specified as a positive scalar.

Dependencies

To enable this property, set the `PathSpecification` property to `'Locations'` and the `CoordinateSystem` property to `'Cartesian'`.

Data Types: `double`

`TransmitterLocation` — Transmitter location

`[0;0;0]` (default) | three-element numeric column vector

Transmitter location, specified as a three-element numeric column vector of coordinates in one of these forms.

- [*x*; *y*; *z*] — This form applies when you set the `CoordinateSystem` property to `'Cartesian'`. The object does not perform range validation for *x*, *y*, and *z*.
- [*latitude*; *longitude*; *height*] — This form applies when you set the `CoordinateSystem` property to `'Geographic'`. *latitude* must be in the range [–90, 90], and *height* must be nonnegative. The object does not perform range validation for *longitude*.

Dependencies

To enable this property, set the `PathSpecification` property to `'Locations'`.

Data Types: `double`

`ReceiverLocation` — Receiver location

`[10;10;10]` (default) | three-element numeric column vector

Receiver location, specified as a three-element numeric column vector of coordinates in one of these forms.

- [*x*; *y*; *z*] — This form applies when you set the `CoordinateSystem` property to `'Cartesian'`. The object does not perform range validation for *x*, *y*, and *z*.
- [*latitude*; *longitude*; *height*] — This form applies when you set the `CoordinateSystem` property to `'Geographic'`. *latitude* must be in the range [–90, 90], and *height* must be nonnegative. The object does not perform range validation for *longitude*.

Dependencies

To enable this property, set the `PathSpecification` property to `'Locations'`.

Data Types: `double`

LineOfSight — Line of sight

true or 1 (default) | false or 0

Line of sight, specified as a logical value of 1 (true) or 0 (false) to indicate whether the ray is a line-of-sight ray.

Dependencies

To enable this property, set the PathSpecification property to 'Locations'.

Data Types: logical

ReflectorLocations — Reflector locations

[10;10;0] (default) | 3-by-*N* numeric matrix

Reflector locations, specified as a 3-by-*N* numeric matrix containing the coordinates of the reflection points for the ray. *N* is the number of reflection points for the ray and is set by the NumReflections property. Each column represents the coordinate location of one reflection point along the propagation path from transmitter to receiver. The order of the columns is the same as the order of the points along the path. Columns (reflection point coordinates) are of one of these forms.

- [*x*; *y*; *z*] — when the CoordinateSystem property is set to 'Cartesian'. The object does not perform range validation for *x*, *y*, and *z*.
- [*latitude*; *longitude*; *height*] — when the CoordinateSystem property is set to 'Geographic'. *latitude* must be in the range [–90, 90], and *height* must be nonnegative. The object does not perform range validation for *longitude*.

Dependencies

To enable this property, set the PathSpecification property to 'Locations' and the LineOfSight property to 0 (false).

.

Data Types: double

PropagationDelay — Propagation delay

5.7775e-08 | nonnegative scalar

Propagation delay in seconds, specified as a nonnegative scalar. The default value is computed using the default values of the TransmitterLocation and ReceiverLocation properties for a line-of-sight ray.

- When you set the PathSpecification property to 'Locations', this property is read-only and the value is derived from TransmitterLocation, ReceiverLocation and, if applicable, the ReflectionLocations.
- When you set the PathSpecification property to 'Delay and angles', this property is configurable.

Data Types: double

PropagationDistance — Propagation distance

17.3205 | nonnegative scalar

This property is read-only.

Propagation distance in meters, specified as a nonnegative scalar. The default value is computed using the default values of the TransmitterLocation and ReceiverLocation properties for a line-of-sight ray.

- When you set the PathSpecification property to 'Locations', the value is derived from TransmitterLocation, ReceiverLocation and, if applicable, the ReflectionLocations.
- When you set the PathSpecification property to 'Delay and angles', the value is derived from PropagationDelay.

Data Types: double

AngleOfDeparture — Angle of departure
[45; 35.2644] | numeric vector of the form [*az*; *el*]

Angle of departure in degrees of the ray at the transmitter, specified as a numeric vector of the form [*az*; *el*]. The azimuth angle, *az*, is measured from the positive x-axis counterclockwise and must be in the range (–180, 180]. The elevation angle, *el*, is measured from the x-y plane and must be in the range [–90, 90]. The default value is computed using the default values of the TransmitterLocation and ReceiverLocation properties for a line-of-sight ray.

- When you set the PathSpecification property to 'Delay and angles', this property is configurable.
- When you set the PathSpecification property to 'Locations', this property is read-only and the value is derived from TransmitterLocation, ReceiverLocation and, if applicable, the ReflectionLocations.
- When CoordinateSystem is set to 'Geographic', the angles are defined with reference to the local East-North-Up (ENU) coordinate system at transmitter.

Data Types: double

AngleOfArrival — Angle of arrival
[-135; -35.2644] | numeric vector of the form [*az*; *el*]

Angle of arrival in degrees of the ray at the receiver, specified as a numeric vector of the form [*az*; *el*]. The azimuth angle, *az*, is measured from the positive x-axis counterclockwise and must be in the range (–180, 180]. The elevation angle, *el*, is measured from the x-y plane and must be in the range [–90, 90]. The default value is computed using the default values of the TransmitterLocation and ReceiverLocation properties for a line-of-sight ray.

- When you set the PathSpecification property to 'Delay and angles', this property is configurable.
- When you set the PathSpecification property to 'Locations', this property is read-only and the value is derived from TransmitterLocation, ReceiverLocation and, if applicable, the ReflectionLocations.
- When CoordinateSystem is set to 'Geographic', the angles are defined with reference to the local East-North-Up (ENU) coordinate system at receiver.

Data Types: double

NumReflections — Number of reflection points
0 (default) | nonnegative integer

This property is read-only.

Number of reflection points for the ray object from the transmitter to the receiver, specified as a nonnegative integer. The value is derived from `LineOfSight` and, if applicable, the `ReflectionLocations`.

Dependencies

To enable this property, set the `PathSpecification` property to `'Locations'`.

Data Types: `double`

`Frequency` — Signal frequency

`1.9e+09` (default) | positive scalar

Signal frequency in Hz, specified as a positive scalar.

Data Types: `double`

`PathLossSource` — Path loss source

`'Free space model'` (default) | `'Custom'`

Path loss source, specified as `'Free space model'` or `'Custom'`.

Data Types: `char` | `string`

`PathLoss` — Path loss

`62.7941` | nonnegative scalar

Path loss in dB, specified as a nonnegative scalar. The default value is computed using the default values of the `TransmitterLocation` and `ReceiverLocation` properties for a line-of-sight ray.

- When you set the `PathLossSource` property to `'Free space model'`, the `PathLoss` property is read-only and derived from the `PropagationDistance` and `Frequency` properties by using the free space propagation model.
- When you set the `PathLossSource` property to `'Custom'`, you can set the `PathLoss` property, independent of the geometric properties.

Data Types: `double`

`PhaseShift` — Phase shift

`4.8537` | numeric scalar

Phase shift in radians, specified as a numeric scalar. The default value is computed using the default values of the `TransmitterLocation` and `ReceiverLocation` properties for a line-of-sight ray.

- When you set the `PathLossSource` property to `'Free space model'`, the `PhaseShift` property is read-only and derived from the `PropagationDistance` and `Frequency` properties by using the free space propagation model.
- When you set the `PathLossSource` property to `'Custom'`, you can set the `PhaseShift` property, independent of the geometric properties.

Data Types: `double`

Object Functions

plot (rays)	Plot rays in Site Viewer map

Examples

Perform Ray Tracing Between Two Sites in Hong Kong

Perform ray tracing between two sites in Hong Kong, generating a cell array containing `comm.Ray` objects. The `comm.Ray` objects contain the geometric and electromagnetic information for the radio wave propagation paths from the transmitter site to the receiver site.

Create a Site Viewer map, loading building data for Hong Kong. For more information about the osm file, see [1] on page 3-0 .

```
viewer = siteviewer('Buildings','hongkong.osm');
```

Specify transmitter and receiver sites.

```
tx = txsite('Latitude',22.2789,'Longitude',114.1625, ...
    'AntennaHeight',10,'TransmitterPower',5, ...
    'TransmitterFrequency',28e9);
rx = rxsite('Latitude',22.2799,'Longitude',114.1617, ...
    'AntennaHeight',1);
```

Perform ray tracing between the sites, generating `comm.Ray` objects in a cell array. For the specified transmitter and receiver sites, performing ray tracing results in a 1-by-1 cell array containing three ray objects in a row vector.

```
rays = raytrace(tx,rx,'Type','pathloss','ColorLimits',[100 250])
```

```
rays = 1×1 cell array
    {1×3 comm.Ray}
```

Display the properties of the first `comm.Ray` object. The `LineOfSight` property value is `1`, and the `NumReflections` property value is `0`. This combination indicates that the ray defines a line-of-sight path.

```
rays{1}(1)
```

```
ans = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 1
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 104.2656
             PhaseShift: 4.6390

   Read-only properties:
       PropagationDelay: 4.6442e-07
    PropagationDistance: 139.2294
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 0
```

Display the properties of the second and third `comm.Ray` objects. The `LineOfSight` property values are `0`, and the `NumReflections` property values are greater than `0`. This combination indicates that the rays define reflected paths.

```
rays{1}(2)
```

```
ans = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×1 double]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 106.2545
             PhaseShift: 0.3951

   Read-only properties:
       PropagationDelay: 4.6490e-07
    PropagationDistance: 139.3720
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 1
```

```
rays{1}(3)
```

```
ans = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×1 double]
              Frequency: 2.8000e+10
         PathLossSource: 'Custom'
               PathLoss: 120.0733
             PhaseShift: 0.3965

   Read-only properties:
       PropagationDelay: 1.1327e-06
    PropagationDistance: 339.5692
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 1
```

Visualize ray tracing results.

```
plot(rays{1});
```

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Plot Propagation Rays Between Sites in Chicago

Return ray tracing results in `comm.Ray` objects and plot the ray propagation path after relaunching the Site Viewer map.

Create a Site Viewer map, loading building data for Chicago. For more information about the osm file, see [1] on page 3-0 .

```
viewer = siteviewer('Buildings','chicago.osm');
```

Create and show a transmitter site on one building and a receiver site on another building.

```
tx = txsite('Latitude',41.8800,'Longitude',-87.6295, ...
    'TransmitterFrequency',2.5e9);
show(tx);
rx = rxsite('Latitude',41.881352,'Longitude',-87.629771, ...
    'AntennaHeight',30);
show(rx);
```

Perform ray tracing, returning the ray object results. For the configuration defined, ray tracing returns a cell array containing one ray object. Display the ray object properties. Then, close the Site Viewer map.

```
rays = raytrace(tx,rx)
```

```
rays = 1×1 cell array
    {1×1 comm.Ray}
```

```
rays{1}
```

```
ans = 
  Ray with properties:

      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×1 double]
              Frequency: 2.5000e+09
         PathLossSource: 'Custom'
               PathLoss: 94.0915
             PhaseShift: 1.2939

   Read-only properties:
       PropagationDelay: 5.7088e-07
    PropagationDistance: 171.1462
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 1
```

```
close(viewer);
```

You can plot the rays without performing ray tracing again. Create another Site Viewer map with the same buildings. Show the transmitter and receiver sites. Using the previously returned cell array of ray objects, plot the reflected rays between the transmitter site and the receiver site. The plot function can plot the path for one ray object at a time.

```
siteviewer('Buildings','chicago.osm');
show(tx);
show(rx);
plot(rays{1},'Type','power', ...
    'TransmitterSite',tx,'ReceiverSite',rx);
```

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

buildingMaterialPermittivity | earthSurfacePermittivity | propagationModel | raypl | raytrace

Objects

arrayConfig | comm.RayTracingChannel | siteviewer

Introduced in R2020a

4

Object Functions

getElementPosition

Positions of the antenna array elements

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
pos = getElementPosition(cfg)
```

Description

`pos = getElementPosition(cfg)` returns the positions of the antenna array elements for the input BLE angle estimation configuration object. The object function uses the origin of the local coordinate system as the position of the first antenna of the antenna array.

Examples

Return Positions of Antenna Array Elements

Create a default BLE angle estimation configuration object.

```
cfg = bleAngleEstimateConfig
```

```
cfg = 
  bleAngleEstimateConfig with properties:

            ArraySize: 4
       ElementSpacing: 0.5000
         SlotDuration: 2
     SwitchingPattern: [1 2 3 4]
```

Return the positions of antenna array elements.

```
pos = getElementPosition(cfg)
```

```
pos = 3×4

         0         0         0         0
         0    0.5000    1.0000    1.5000
         0         0         0         0
```

Input Arguments

`cfg` — BLE angle estimation configuration object
`bleAngleEstimateConfig` object

BLE angle estimation configuration object, specified as a `bleAngleEstimateConfig` object.

Output Arguments

`pos` — Positions of the antenna array elements
3-by-*N* matrix

Positions of the antenna array elements, returned as a 3-by-*N* matrix, where *N* is the number of antenna array elements specified by the `cfg` input. Each column in `pos` defines the position of the antenna array element in the form [*x*; *y*; *z*] in the local coordinate system.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Wooley, Martin. *Bluetooth Direction Finding: A Technical Overview*. Bluetooth Special Interest Group (SIG), Accessed April 6, 2020, https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleAngleEstimate` | `bleIdealReceiver` | `bleWaveformGenerator` | `getNumElements`

Objects
`bleAngleEstimateConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”
“Bluetooth Location and Direction Finding”

Introduced in R2020b

getNumElements

Number of elements in antenna array

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
numElements = getNumElements(cfg)
```

Description

`numElements = getNumElements(cfg)` returns the number of elements in the antenna array for the input BLE angle estimation configuration object.

Examples

Return Number of Elements in Antenna Array

Create a BLE angle estimation configuration object, specifying a 2-by-3 URA.

```
cfg = bleAngleEstimateConfig('ArraySize',[2 3])
```

```
cfg = 
  bleAngleEstimateConfig with properties:

           ArraySize: [2 3]
      ElementSpacing: 0.5000
        SlotDuration: 2
    SwitchingPattern: [1 2 3 4]
```

Return the number of elements in the antenna array.

```
N = getNumElements(cfg)
```

```
N = 6
```

Input Arguments

`cfg` — BLE angle estimation configuration object
`bleAngleEstimateConfig` object

BLE angle estimation configuration object, specified as a `bleAngleEstimateConfig` object.

Output Arguments

`numElements` — Number of elements in antenna array
positive integer

Number of elements in the antenna array, returned as a positive integer. If the `SlotDuration` property of the `cfg` input is 2 µs, the value of this output is in the range [2, 38]. If the `SlotDuration` property is 1 µs, the value of this output is in the range [2, 75].

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

[3] Wooley, Martin. *Bluetooth Direction Finding: A Technical Overview*. Bluetooth Special Interest Group (SIG), Accessed April 6, 2020, https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`bleAngleEstimate` | `bleIdealReceiver` | `bleWaveformGenerator` | `getElementPosition`

Objects
`bleAngleEstimateConfig`

Topics
“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”
“Bluetooth Location and Direction Finding”

Introduced in R2020b

nextHop

Select Bluetooth BR/EDR channel index to hop for next frequency

Note Download Required: To use , first download Communications Toolbox™ Library for the Bluetooth® Protocol. For more information, see “Get and Manage Add-Ons”. Alternatively, see Communications Toolbox Library for the Bluetooth Protocol File Exchange.

Syntax

```
[channelIndex,X] = nextHop(FH,Clock)
```

Description

`[channelIndex,X] = nextHop(FH,Clock)` selects a Bluetooth basic rate/enhanced data rate (BR/EDR) channel index, `channelIndex` to hop for next frequency. This selection is based on the Bluetooth BR/EDR frequency hopping object, `FH`, the clock, `Clock`, and the “SequenceType” on page 3-0 property of the “FH” on page 4-0 . The function also returns `X`, which is required for implementing whitening process in the physical layer (PHY).

Examples

Select Bluetooth BR/EDR Channel Index Using Default Values

Create a default Bluetooth BR/EDR channel index object for frequency hopping.

```
fh = bluetoothFrequencyHop
```

```
fh = 
  bluetoothFrequencyHop with properties:

      DeviceAddress: '9E8B33'
       SequenceType: 'Inquiry'
    InterlaceOffset: 16
             KNudge: 0
            KOffset: 24
            Counter: 0
       UsedChannels: [1x79 double]
```

Specify a clock value.

```
inputClock = '12C'; % 28-bit
```

Select a Bluetooth BR/EDR channel index to hop for the next frequency.

```
[channelIndex, X] = nextHop(fh,inputClock)
```

```
channelIndex = 41
```

```
X = 30
```

Input Arguments

FH — Bluetooth BR/EDR channel index for frequency hopping

bluetoothFrequencyHop object

Bluetooth BR/EDR channel index for frequency hopping, specified as a `bluetoothFrequencyHop` object.

`Clock` — Clock

character vector in hexadecimal format | string scalar in hexadecimal format | numeric scalar in the range [0, 2^{28}–1]

Clock, specified as one of these values:

- Character vector — This vector represents the `Clock` in hexadecimal format
- String scalar — This scalar represents the `Clock` in hexadecimal format
- Numeric scalar — This scalar represent the `Clock` in the range [0, 2^{28}–1]

This argument is a 28-bit value that computes the inputs to the hop selection kernel. This table shows the dependency of the `Clock` argument on the value of the `SequenceType` property of the `FH` input.

Value of SequenceType Property	`Clock` Input
`'Connection basic'` or `'Connection adaptive'`	Indicates native clock of the Master
`'Page'` or `'Inquiry'`	Indicates native clock of the Slave
`'Page scan'` or `'Inquiry scan'`	Indicates the estimated value of the clock for the Slave
`'Slave page response'`	Indicates the value when the access code of the recipient is detected
`'Master page response'`	Indicates the value that triggered a response from the paged device

Data Types: `char` | `string` | `double`

Output Arguments

`channelIndex` — Channel index

integer in the range [0, 78]

Channel index, returned as an integer in the range [0, 78].

Data Types: `double`

X — Control signal to be used in whitening process

nonnegative integer

Control signal to be used in whitening process, returned as a nonnegative integer.

Data Types: `double`

References

[1] Bluetooth Technology Website. “Bluetooth Technology Website | The Official Website of Bluetooth Technology.” Accessed November 22, 2019. https://www.bluetooth.com/.

[2] Bluetooth Special Interest Group (SIG). "Bluetooth Core Specification." Version 5.1. https://www.bluetooth.com/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Objects

`bluetoothFrequencyHop` | `bleChannelSelection`

Topics

“Bluetooth Protocol Stack”
“Bluetooth Packet Structure”

Introduced in R2020b

write

Write BLE LL protocol packet data to PCAP or PCAPNG file

Syntax

```
write(pcapObj,packet,timestamp)
write(pcapObj,packet,timestamp,Name,Value)
```

Description

`write(pcapObj,packet,timestamp)` writes Bluetooth low energy (BLE) link layer (link layer) protocol packet data to the PCAP or PCAPNG file specified in the BLE PCAP file writer object, `pcapObj`. Input `packet` specifies the BLE LL protocol packet, and input `timestamp` specifies the packet arrival time.

`write(pcapObj,packet,timestamp,Name,Value)` specifies options using one or more name-value pair arguments. For example, `'PacketFormat','bits'` sets the format of the BLE LL protocol packets to bits.

Examples

Write BLE LL Packet to PCAP File

Create a default BLE PCAP file writer object.

```
pcapObj = blePCAPWriter;
```

Generate a BLE LL packet.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', ...
    'Data (start fragment/complete)');
payload = '0E00050014010A001F004000170017000000';
llDataPDU = bleLLDataChannelPDU(cfgLLData, payload);
connAccessAddress = de2bi(hex2dec('E213BC42'), 32)';
llpacket = [connAccessAddress; llDataPDU];
```

Write the BLE LL packet to the PCAP file.

```
timestamp = 0; % Number of microseconds
write(pcapObj,llpacket,timestamp,'PacketFormat','bits');
```

Use BLE PCAP Writer to Write BLE LL Packet to PCAPNG File

Create a BLE PCAPNG file writer object, specifying the name and extension of the PCAPNG file.

```
pcapObj = blePCAPWriter('FileName','sampleBLELL', ...
    'FileExtension','pcapng');
```

Generate a BLE LL packet.

```
cfgLLData = bleLLDataChannelPDUConfig('LLID', ...
    'Data (start fragment/complete)');
payload = '0E00050014010A001F004000170017000000';
llDataPDU = bleLLDataChannelPDU(cfgLLData,payload);
connAccessAddress = de2bi(hex2dec('E213BC42'),32)';
llpacket = [connAccessAddress; llDataPDU];
```

Write the BLE LL packet to the PCAPNG file.

```
timestamp = 12800000; % Number of microseconds
write(pcapObj,llpacket,timestamp,'PacketFormat','bits');
```

Input Arguments

Note The `blePCAPWriter` object does not overwrite the existing PCAP or PCAPNG file. During each call of this object, specify a unique PCAP or PCAPNG file name.

`pcapObj` — BLE PCAP file writer object

`blePCAPWriter` object

BLE PCAP file writer object, specified as a `blePCAPWriter` object.

`packet` — BLE LL protocol packet

binary-valued vector | character vector | string scalar | numeric vector | *n*-by-2 character array

BLE LL protocol packet, specified as one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.
- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format.

Data Types: `char` | `string` | `double`

`timestamp` — Packet arrival time

nonnegative integer

Packet arrival time since 1/1/1970, specified as a nonnegative integer. This value must be expressed in microseconds.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'PacketFormat','bits' sets the format of the BLE LL protocol packets to bits.

PhyHeader — BLE LL protocol packet metadata
binary-valued vector | character vector | string scalar | numeric vector | *n*-by-2 character array

BLE LL protocol packet metadata, specified as the comma-separated pair consisting of PhyHeader and one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.
- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format.

Data Types: char | string | double

PacketComment — Comment for the BLE LL protocol packet
'' (default) | character vector | string scalar

Comment for the BLE LL protocol packet, specified as the comma-separated pair consisting of PacketComment and a character vector or a string scalar.

Data Types: char | string

PacketFormat — Format of the BLE LL protocol packet
'octets' (default) | 'bits'

Format of the BLE LL protocol packet, specified as the comma-separated pair consisting of PacketFormat and 'octets' or 'bits'. If this value is specified as 'octets', packet is specified as one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.
- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format.

Data Types: char | string | double

PhyHeaderFormat — Format of the PHY header
'octets' (default) | 'bits'

Format of the physical layer (PHY) header, specified as the comma-separated pair consisting of PhyHeaderFormat and 'octets' or 'bits'. If this value is specified as 'octets', PhyHeader can be specified as one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.

- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format

Data Types: `char` | `string` | `double`

References

[1] Tuexen, M. “PCAP Next Generation (Pcapng) Capture File Format.” 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Objects
`blePCAPWriter`

Introduced in R2020b

write

Write protocol packet data to PCAP or PCAPNG file

Syntax

```
write(pcapObj,packet,timestamp)
write(pcapngObj,packet,timestamp,interfaceID)
write( ___ ,Name,Value)
```

Description

`write(pcapObj,packet,timestamp)` writes the protocol packet data to the PCAP file specified in the PCAP file writer object, `pcapObj`. Input `packet` specifies the protocol packet and input `timestamp` specifies the packet arrival time.

`write(pcapngObj,packet,timestamp,interfaceID)` writes protocol packet data to a PCAPNG file specified in the PCAPNG file writer object, `pcapngObj`. Input `packet`, `timestamp`, and `interfaceID` specifies the protocol packet, packet arrival time, and interface identifier, respectively.

`write( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input argument combinations from any of the previous syntaxes. For example, `'PacketFormat','bits'` sets the format of the protocol packets to bits.

Examples

Write BLE Packet Data to PCAP File

Create a PCAP file writer object, specifying the name of the PCAP file. Specify the Bluetooth low energy (BLE) link type.

```
pcapObj = pcapWriter('FileName','writeBLEpacket');
bleLinkType = 251;
```

Write a global header to the PCAP file.

```
writeGlobalHeader(pcapObj,bleLinkType);
```

Specify the BLE link layer (LL) packet.

```
llpacket = '42BC13E206120E00050014010A001F004000170017000007D47C0';
```

Write BLE LL packet to the PCAP file.

```
timestamp = 129100; % Number of microseconds
write(pcapObj,llpacket,timestamp);
```

Write BLE Packet Data to PCAPNG File

Create a PCAPNG file writer object, specifying the name of the PCAPNG file.

```
pcapngObj = pcapngWriter('FileName','writeBLEpacket');
```

Write an interface description block for BLE.

```
interfaceName = 'BLE interface';
bleLinkType = 251;
interfaceId = writeInterfaceDescriptionBlock(pcapngObj,bleLinkType, ...
    interfaceName);
```

Specify the BLE LL packet.

```
llpacket = '42BC13E206120E00050014010A001F004000170017000007D47C0';
```

Write BLE LL packet to the PCAPNG format file.

```
timestamp = 0; % Number of microseconds
packetComment = 'This is BLE packet';
write(pcapngObj, llpacket,timestamp,interfaceId,'PacketComment', ...
    packetComment);
```

Input Arguments

Note The `pcapWriter` and `pcapngWriter` objects do not overwrite the existing PCAP or PCAPNG files, respectively. During each call of these objects, specify a unique PCAP or PCAPNG file name.

`pcapObj` — PCAP file writer object

`pcapWriter` object

PCAP file writer object, specified as a `pcapWriter` object.

`packet` — Protocol packet

binary-valued vector | character vector | string scalar | numeric vector | *n*-by-2 character array

Protocol packet, specified as one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.
- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format .

Data Types: `char` | `string` | `double`

`timestamp` — Packet arrival time

nonnegative integer

Packet arrival time since 1/1/1970, specified as a nonnegative integer. This value must be expressed in microseconds.

Data Types: `double`

`pcapngObj` — PCAPNG file writer object
`pcapngWriter` object

PCAPNG file writer object, specified as a `pcapngWriter` object.

`interfaceID` — Unique identifier for an interface
nonnegative scalar

Unique identifier for an interface, specified as a nonnegative scalar.

Data Types: `double`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'PacketFormat','bits'` sets the format of the protocol packets to bits.

`PacketFormat` — Format of the protocol packet
`'octets'` (default) | `'bits'`

Format of the protocol packet, specified as the comma-separated pair consisting of `PacketFormat` and `'octets'` or `'bits'`. If this value is specified as `'octets'`, `packet` is specified as one of these values.

- Binary-valued vector – This value represents bits.
- Character vector – This value represents octets in hexadecimal format.
- String scalar – This value represents octets in hexadecimal format.
- Numeric vector with each element in the range [0, 255] – This value represents octets in decimal format.
- *n*-by-2 character array – In this value, each row represents an octet in hexadecimal format .

Data Types: `char` | `string` | `double`

`PacketComment` — Comment for protocol packet
`''` (default) | character vector | string scalar

Comment for the protocol packet, specified as the comma-separated pair consisting of `PacketComment` and a character vector or a string scalar.

Dependencies

To enable this name-value pair argument, specify the `pcapngObj` input argument.

Data Types: `char` | `string`

References

[1] Tuexen, M. "PCAP Next Generation (Pcapng) Capture File Format." 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`writeCustomBlock` | `writeGlobalHeader` | `writeInterfaceDescriptionBlock`

Objects

`pcapWriter` | `pcapngWriter`

Introduced in R2020b

writeCustomBlock

Write custom block to PCAPNG file

Syntax

```
writeCustomBlock(pcapngObj,customData)
```

Description

`writeCustomBlock(pcapngObj,customData)` writes the custom block data, `customData`, to a PCAPNG file specified in the PCAPNG file writer object.

Examples

Write BLE User-Defined Custom Block to PCAPNG File

Create a PCAPNG file writer object, specifying the name of the PCAPNG file.

```
pcapngObj = pcapngWriter('FileName','writeBLEcustomdata');
```

Write the interface description block for Bluetooth low energy (BLE).

```
interfaceName = 'BLE interface';
bleLinkType = 251;
interfaceId = writeInterfaceDescriptionBlock(pcapngObj,bleLinkType, ...
    interfaceName);
```

Write the custom block to specify user-defined data.

```
writeCustomBlock(pcapngObj,"This block writes user-defined data");
```

Input Arguments

Note The `pcapngWriter` object does not overwrite the existing PCAPNG file. During each call of this object, specify a unique PCAPNG file name.

pcapngObj — PCAPNG file writer object

`pcapngWriter` object

PCAPNG file writer object, specified as a `pcapngWriter` object.

customData — User-defined data

character vector | string scalar

User-defined data, specified as a character vector or a string scalar.

Data Types: `char` | `string`

References

[1] Tuexen, M. “PCAP Next Generation (Pcapng) Capture File Format.” 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`write` | `writeInterfaceDescriptionBlock`

Objects

`pcapWriter` | `pcapngWriter`

Introduced in R2020b

writeGlobalHeader

Write global header to PCAP file

Syntax

```
writeGlobalHeader(pcapObj,linkType)
```

Description

`writeGlobalHeader(pcapObj,linkType)` writes a global header to the PCAP file specified in the PCAP file writer object, `pcapObj`. Input `linkType` specifies the unique identifier for a protocol.

Examples

Write BLE Packet Global Header to PCAP File

Create a PCAP file writer object, specifying the name of the PCAP file.

```
pcapObj = pcapWriter('FileName','writeBLEheader');
```

Specify the Bluetooth low energy (BLE) link type.

```
bleLinkType = 251;
```

Write a global header to the PCAP file.

```
writeGlobalHeader(pcapObj,bleLinkType);
```

Input Arguments

Note The `pcapWriter` object does not overwrite the existing PCAP file. During each call of this object, specify a unique PCAP file name.

`pcapObj` — PCAP file writer object

`pcapWriter` object

PCAP file writer object, specified as a `pcapWriter` object.

`linkType` — Unique identifier for a protocol

integer in the range [0, 65,535].

Unique identifier for a protocol, specified as an integer in the range [0, 65535].

Data Types: `double`

References

[1] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[2] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`write`

Objects

`pcapWriter` | `pcapngWriter`

Introduced in R2020b

writeInterfaceDescriptionBlock

Write interface description block to PCAPNG file

Syntax

```
interfaceID = writeInterfaceDescriptionBlock(pcapngObj,linkType,interface)
```

Description

`interfaceID = writeInterfaceDescriptionBlock(pcapngObj,linkType,interface)` writes an interface description block to the PCAPNG file specified in the PCAPNG file writer object, `pcapngObj`. Input `linkType` specifies the unique identifier for the protocol and input `interface` specifies the interface on which the protocol packets are captured. This object function returns the unique identifier for the interface.

Examples

Write BLE Interface Description Block to PCAPNG File

Create a PCAPNG file writer object, specifying the name of the PCAPNG file.

```
pcapngObj = pcapngWriter('FileName','writeBLEinterface');
```

Write the interface description block for BLE.

```
interfaceName = 'BLE interface';
bleLinkType = 251;
interfaceId = writeInterfaceDescriptionBlock(pcapngObj,bleLinkType, ...
    interfaceName);
```

Input Arguments

Note The `pcapngWriter` object does not overwrite the existing PCAPNG file. During each call of this object, specify a unique PCAPNG file name.

`pcapngObj` — PCAPNG file writer object

`pcapngWriter` object

PCAPNG file writer object, specified as a `pcapngWriter` object.

`linkType` — Unique identifier for protocol

integer in the range [0, 65,535].

Unique identifier for a protocol, specified as an integer in the range [0, 65,535].

Data Types: `double`

`interface` — Name of the interface on which protocol packets are captured
character vector | string scalar

Name of the interface on which protocol packets are captured, specified as a character vector or a string scalar in 8-bit unicode transformation format (UTF-8) format.

Data Types: `char` | `string`

Output Arguments

`interfaceID` — Unique identifier for an interface
nonnegative scalar

Unique identifier for an interface, specified as a nonnegative scalar.

Data Types: `double`

References

[1] Tuexen, M. “PCAP Next Generation (Pcapng) Capture File Format.” 2020. https://www.ietf.org/.

[2] Group, The Tcpdump. “Tcpdump/Libpcap Public Repository.” Accessed May 20, 2020. https://www.tcpdump.org.

[3] “Development/LibpcapFileFormat - The Wireshark Wiki.” Accessed May 20, 2020. https://www.wireshark.org/.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`write` | `writeCustomBlock`

Objects
`pcapWriter` | `pcapngWriter`

Introduced in R2020b

coeffs

Package: `comm`

Coefficients for filters

Syntax

```
coefInfo = coeffs(rcfilter)
coefInfo = coeffs(rcfilter,'Arithmetic',arithType)
```

Description

`coefInfo = coeffs(rcfilter)` obtain the coefficients for the specified filter System object.

`coefInfo = coeffs(rcfilter,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`.

Examples

Obtain Coefficients of Raised Cosine Filters

Create a receive raised cosine filter and obtain its numerator coefficients.

```
rxfilter = comm.RaisedCosineReceiveFilter('FilterSpanInSymbols',25);
srx = coeffs(rxfilter)
```

```
srx = struct with fields:
    Numerator: [1x201 double]
```

Calculate the expected number of numerator coefficients and confirm the value equals the length of `srx.Numerator`.

```
numcoefs = rxfilter.FilterSpanInSymbols * rxfilter.InputSamplesPerSymbol + 1
```

```
numcoefs = 201
```

```
isequal (numcoefs,length(srx.Numerator))
```

```
ans = logical
   1
```

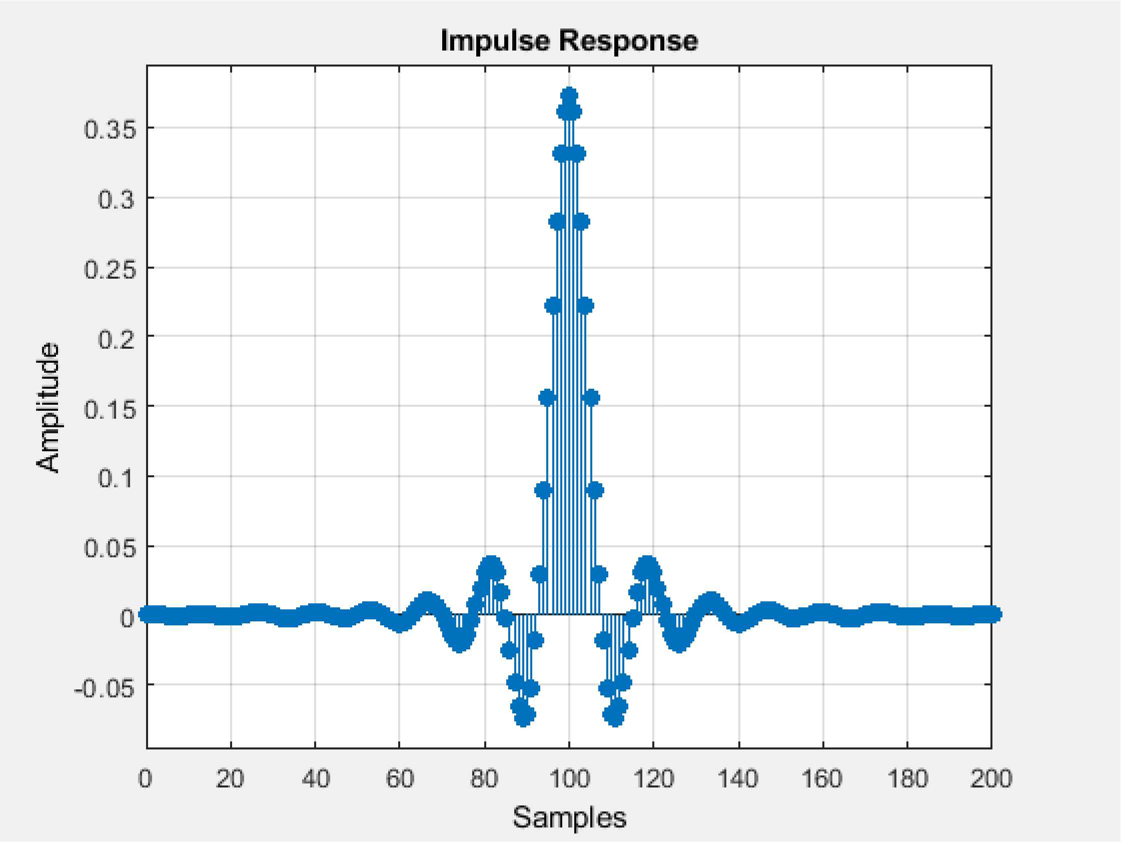
Display the first ten coefficients.

```
srx.Numerator(1:10)
```

```
ans = 1×10
10^-3 ×

    0.9094    0.8559    0.6136    0.2320   -0.2074   -0.6071   -0.8759   -0.9486   -0.8021   -0.4
```

Display the impulse response of the receive raised cosine filter.

```
fvtool(rxfilter,'impulse')
```



Create a transmit raised cosine filter and obtain its numerator coefficients.

```
txfilter = comm.RaisedCosineTransmitFilter('RolloffFactor',0.5);
stx = coeffs(txfilter);
```

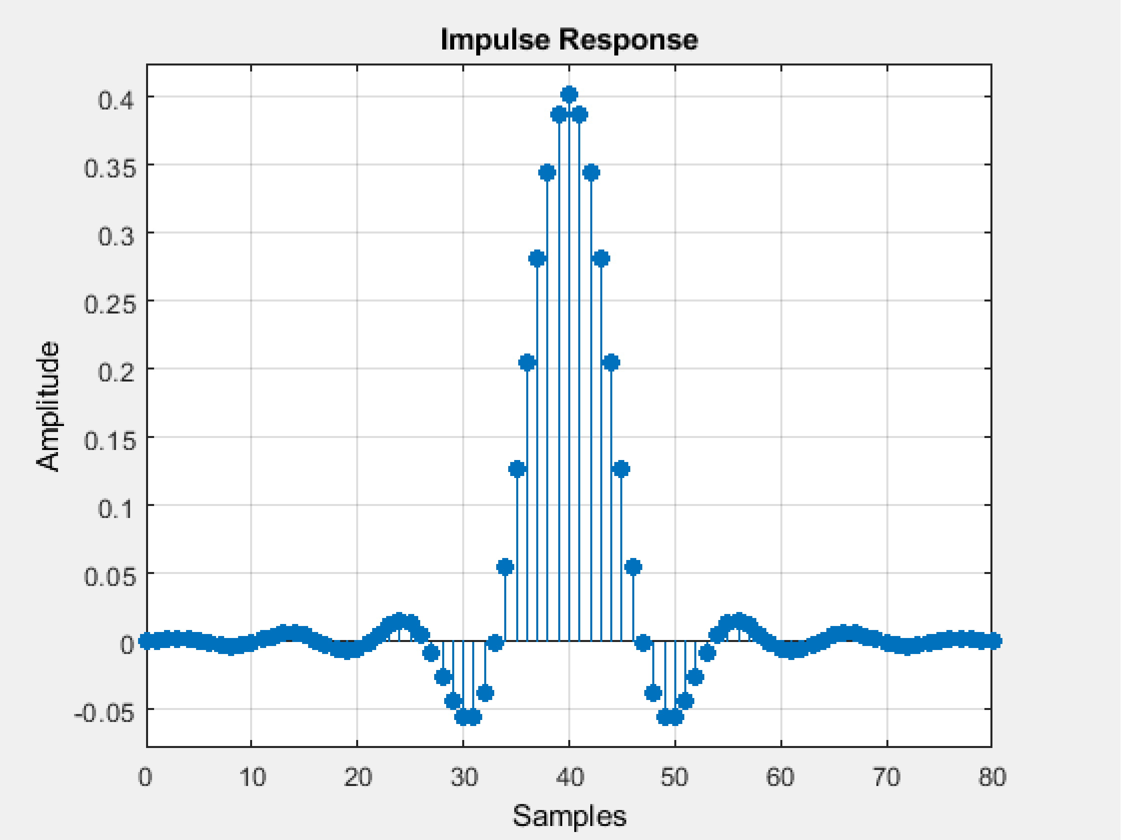
Display the first ten filter coefficients.

```
stx.Numerator(1:10)
```

```
ans = 1×10

   -0.0002    0.0011    0.0021    0.0024    0.0018    0.0004   -0.0014   -0.0029   -0.0036   -0.0
```

Display the impulse response of the transmit raised cosine filter.

```
fvtool(txfilter,'impulse')
```



Input Arguments

`rcfilter` — Input filter

`comm.RaisedCosineReceiveFilter` System object | `comm.RaisedCosineTransmitFilter` System object

Input filter, specified as one of these of filter System objects.

- `comm.RaisedCosineReceiveFilter`
- `comm.RaisedCosineTransmitFilter`

`arithType` — Arithmetic type

`'double'` (default) | `'single'` | `'Fixed'`

Specify the arithmetic used in the analysis. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The `'Arithmetic'` input argument specified as `'Fixed'` applies only to filter System objects with fixed-point properties.

When the `'Arithmetic'` input argument is specified as `'Fixed'` and the filter object has coefficients data type set to `'Same word length as input'`, the arithmetic analysis depends on whether the System object is:

- unlocked –– The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- locked –– When the input data type is 'double' or 'single', the analysis object function cannot determine the coefficients data type. The function assumes that the coefficient data type is signed, has 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the isLocked function.

When the arithmetic input is specified as 'Fixed' and the filter object has coefficients data type set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

Output Arguments

coefInfo — Filter coefficient info
structure

Filter coefficient info, returned as a structure containing the filter coefficients in the Numerator field. When the filter uses fixed-point arithmetic, the function returns additional information about the filter. This information includes the arithmetic setting and details about the filter internals.

See Also

Functions
info | order

Objects
comm.RaisedCosineReceiveFilter | comm.RaisedCosineTransmitFilter

Introduced in R2013b

info

Package: comm

Information about filter System object

Syntax

```
filtInfo = info(rcfilter)
filtInfo = info(rcfilter,infoType)
filtInfo = info(___,'Arithmetic',arithType)
```

Description

`filtInfo = info(rcfilter)` obtains information about the specified filter System object. The type of information returned by the function depends on the filter type and configuration.

`filtInfo = info(rcfilter,infoType)` obtains the amount of filter information as specified by `infoType`.

`filtInfo = info(___,'Arithmetic',arithType)` analyzes the filter System object based on the arithmetic specified in `arithType`. Specify this option with any of the input combinations from previous syntaxes.

For more input options, see `info` in Signal Processing Toolbox™.

Examples

Obtain Raised Cosine Filter Information

Obtain short-format and long-format information about a raised cosine filter.

```
txfilter = comm.RaisedCosineTransmitFilter;
info(txfilter)
```

```
ans = 10x62 char array
    'Discrete-Time FIR Multirate Filter (real)                     '
    '-----------------------------------------                     '
    'Filter Structure      : Direct-Form FIR Polyphase Interpolator'
    'Interpolation Factor  : 8                                     '
    'Polyphase Length      : 11                                    '
    'Filter Length         : 81                                    '
    'Stable                : Yes                                   '
    'Linear Phase          : Yes (Type 1)                          '
    '                                                              '
    'Arithmetic            : double                                '
```

```
info(txfilter,'long')
```

```
ans = 17x62 char array
    'Discrete-Time FIR Multirate Filter (real)                     '
```

```
'----------------------------------------                      '
'Filter Structure      : Direct-Form FIR Polyphase Interpolator'
'Interpolation Factor  : 8                                     '
'Polyphase Length      : 11                                    '
'Filter Length         : 81                                    '
'Stable                : Yes                                   '
'Linear Phase          : Yes (Type 1)                          '
'                                                              '
'Arithmetic            : double                                '
'                                                              '
'Implementation Cost                                           '
'Number of Multipliers            : 81                         '
'Number of Adders                 : 73                         '
'Number of States                 : 10                         '
'Multiplications per Input Sample : 81                         '
'Additions per Input Sample       : 73                         '
```

Input Arguments

`rcfilter` — Input filter
`comm.RaisedCosineReceiveFilter` System object | `comm.RaisedCosineTransmitFilter` System object

Input filter, specified as one of these of filter System objects.

- `comm.RaisedCosineReceiveFilter`
- `comm.RaisedCosineTransmitFilter`

`infoType` — Amount of information to display
`'short'` (default) | `'long'`

Amount of filter information to display, specified as one of these values.

- `'short'` –– The function displays the basic filter information. This information is the same as the information output by `info(rcfilter)`.
- `'long'` –– The function returns this information about the filter.
  - Specifications such as the filter structure and filter order.
  - Information about the design method and options.
  - Performance measurements, such as the passband cutoff or stopband attenuation, for the filter response.
  - Cost of implementing the filter in terms of operations required to apply the filter to data.

  When the filter uses fixed-point arithmetic, the function returns additional information about the filter. This information includes the arithmetic setting and details about the filter internals.

Data Types: `char` | `string`

`arithType` — Arithmetic type
`'double'` (default) | `'single'` | `'Fixed'`

Specify the arithmetic used in the analysis. When the arithmetic input is not specified and the filter System object is unlocked, the analysis tool assumes a double-precision filter. When the arithmetic input is not specified and the System object is locked, the function performs the analysis based on the data type of the locked input.

The `'Arithmetic'` input argument specified as `'Fixed'` applies only to filter System objects with fixed-point properties.

When the `'Arithmetic'` input argument is specified as `'Fixed'` and the filter object has coefficients data type set to `'Same word length as input'`, the arithmetic analysis depends on whether the System object is:

- `unlocked` –– The analysis object function cannot determine the coefficients data type. The function assumes that the coefficients data type is signed, has 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.
- `locked` –– When the input data type is `'double'` or `'single'`, the analysis object function cannot determine the coefficients data type. The function assumes that the coefficient data type is signed, has 16-bit word length, and is auto scaled. The function performs fixed-point analysis based on this assumption.

To check if the System object is locked or unlocked, use the `isLocked` function.

When the arithmetic input is specified as `'Fixed'` and the filter object has coefficients data type set to a custom numeric type, the object function performs fixed-point analysis based on the custom numeric data type.

Output Arguments

`filtInfo` — Filter information
character array

Filter information, returned as a character array.

- When you set the `infoType` input to `'short'`, the function displays basic filter information.
- When you set the `infoType` input to `'long'`, the function displays this.
  - Specifications such as the filter structure and filter order.
  - Information about the design method and options.
  - Performance measurements, such as the passband cutoff or stopband attenuation, for the filter response.
  - Cost of implementing the filter in terms of operations required to apply the filter to data.

When the filter uses fixed-point arithmetic, the function returns additional information about the filter. The information includes the arithmetic setting and details about the filter internals.

See Also

Functions
`coeffs` | `info` | `order`

Objects
`comm.RaisedCosineReceiveFilter` | `comm.RaisedCosineTransmitFilter`

Topics

"Analysis Methods for Filter System Objects"

Introduced in R2013b

hide

Package: `comm`

Hide scope window

Syntax

```
hide(scope)
```

Description

`hide(scope)` hides the window of the System object scope.

Examples

Hide and Show Scope

Create a `comm.ConstellationDiagram` object.

```
scope = comm.ConstellationDiagram;
```

Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```

Hide the constellation diagram scope window again.

```
if(isVisible(scope))
    hide(scope)
end
```

Hide and Show Constellation Diagram

Generate a 16-QAM reference constellation and a signal to display.

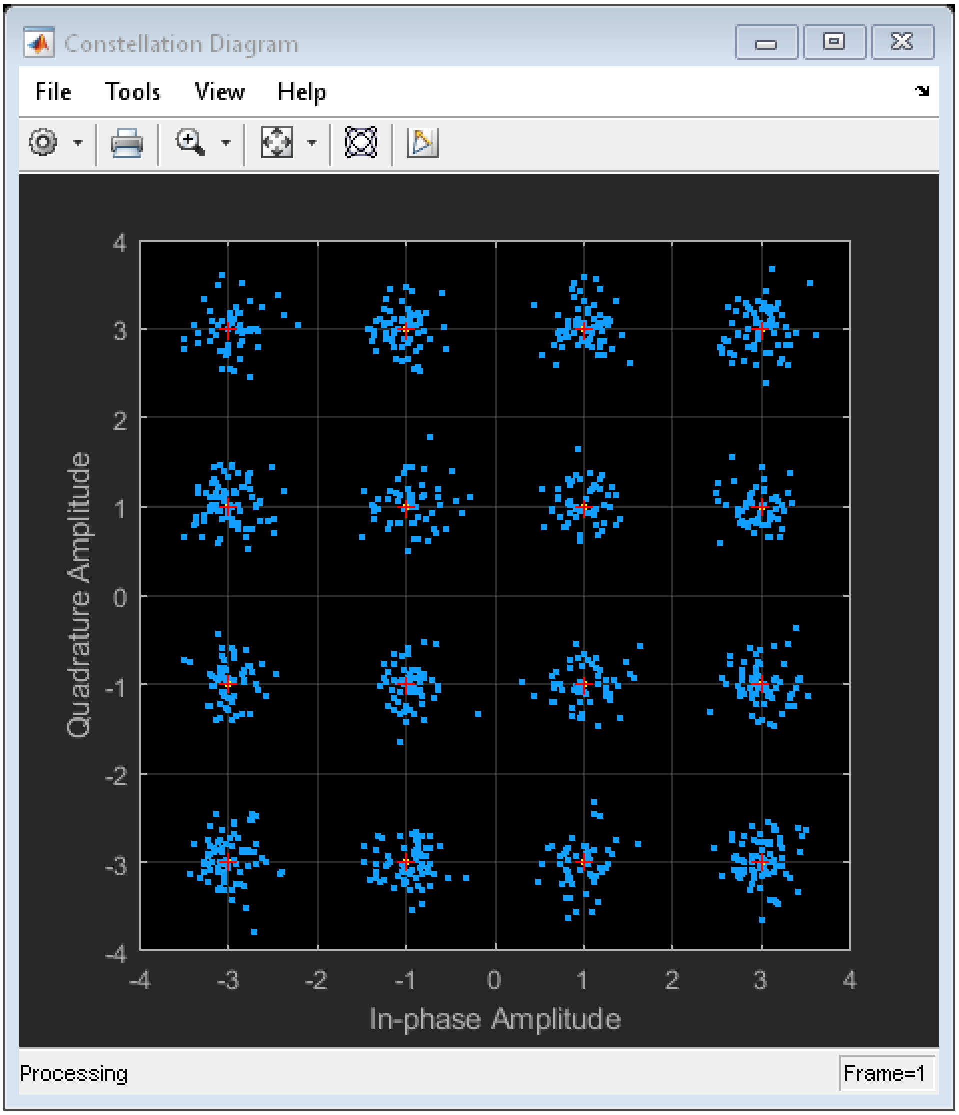
```
M = 16;
xRef = (0:M-1)';
refConst = qammod(xRef,M);
signal = randi([0 M-1],1000,1);
```

Create a constellation diagram System object™, specifying the constellation reference points and axes limits using name-value pairs.

```
scope = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Modulate the random data signal using QAM. Add Gaussian white noise to the QAM symbols. Display the QAM symbols and noisy symbols with the constellation diagram object.

```
sym = qammod(signal,M);
rcv = awgn(sym,20,'measured');
scope([sym rcv]);
```

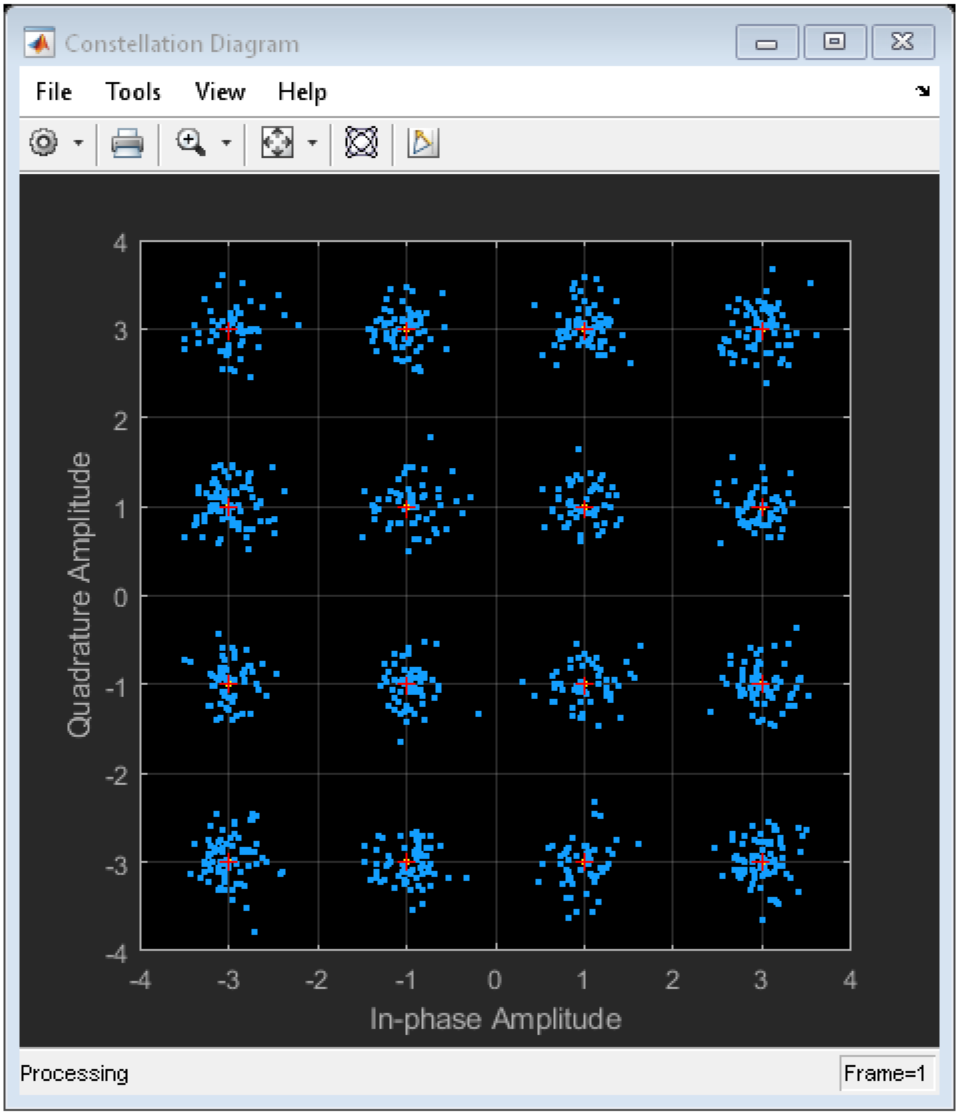


Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Clear the workspace variables.

```
clear scope sym rcv M refConst signal xRef
```

Input Arguments

scope — Scope System object

scope System object

Scope System object, specified as a `comm.ConstellationDiagram` System object.

Example: `scope = comm.ConstellationDiagram;`

See Also

Functions

`isVisible` | `show`

Objects

`comm.ConstellationDiagram`

Introduced in R2013a

isVisible

Package: comm

Determine visibility of scope window

Syntax

```
visibility = isVisible(scope)
```

Description

visibility = isVisible(scope) returns a logical to verify in the System object scope is open. visibility is 1 if the scope window is open and 0 otherwise.

Examples

Hide and Show Scope

Create a comm.ConstellationDiagram object.

```
scope = comm.ConstellationDiagram;
```

Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```

Hide the constellation diagram scope window again.

```
if(isVisible(scope))
    hide(scope)
end
```

Hide and Show Constellation Diagram

Generate a 16-QAM reference constellation and a signal to display.

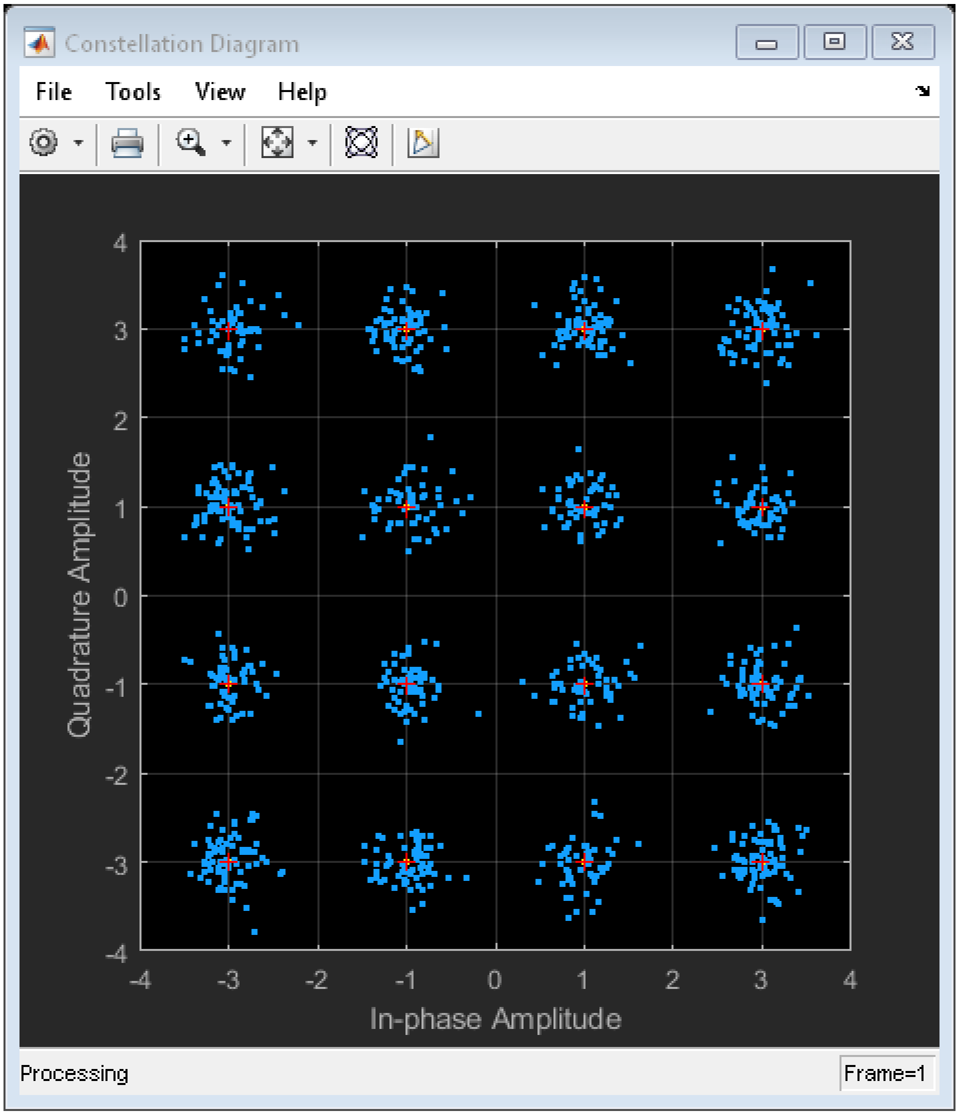
```
M = 16;
xRef = (0:M-1)';
refConst = qammod(xRef,M);
signal = randi([0 M-1],1000,1);
```

Create a constellation diagram System object™, specifying the constellation reference points and axes limits using name-value pairs.

```
scope = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Modulate the random data signal using QAM. Add Gaussian white noise to the QAM symbols. Display the QAM symbols and noisy symbols with the constellation diagram object.

```
sym = qammod(signal,M);
rcv = awgn(sym,20,'measured');
scope([sym rcv]);
```

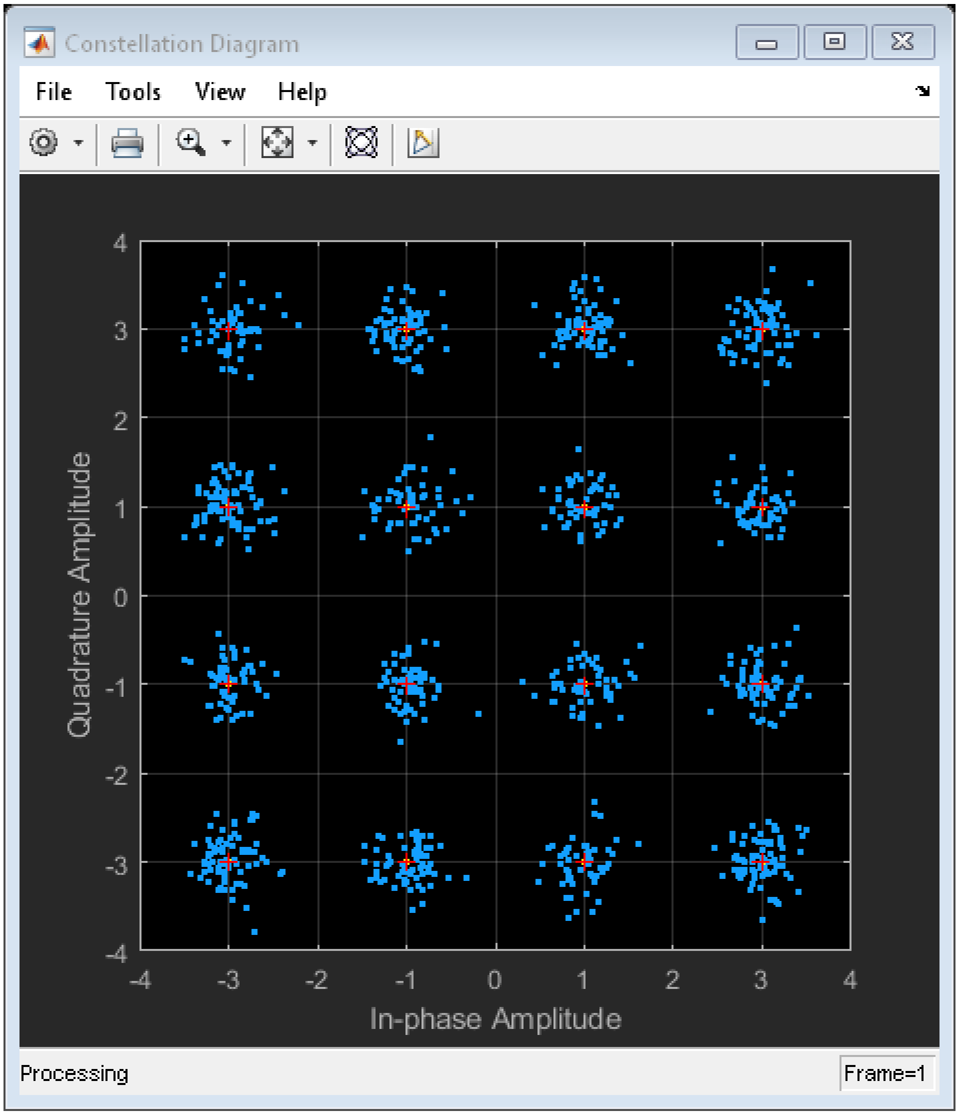


Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Clear the workspace variables.

```
clear scope sym rcv M refConst signal xRef
```

Input Arguments

scope — Scope System object

scope System object

Scope System object, specified as a `comm.ConstellationDiagram` or `comm.EyeDiagram` System object.

Example: `scope = comm.ConstellationDiagram;`

See Also

Functions
`hide` | `show`

Objects
`comm.ConstellationDiagram` | `comm.EyeDiagram`

Introduced in R2013a

order

Package: `comm`

Order of discrete-time filter System object

Syntax

```
filtOrder = order(rcfilter)
```

Description

`filtOrder = order(rcfilter)` returns the order of the filter System object. The order depends on the filter structure and the reference double-precision floating-point coefficients.

Examples

Determine Group Delay for RRC Filter Pair

Create a square root raised cosine (RRC) filter pair using the `comm.RaisedCosineTransmitFilter` and `comm.RaisedCosineReceiveFilter` System objects. Determine the group delay of the overall filter pair.

```
txrcfilt = comm.RaisedCosineTransmitFilter
```

```
txrcfilt = 
  comm.RaisedCosineTransmitFilter with properties:

                     Shape: 'Square root'
             RolloffFactor: 0.2000
       FilterSpanInSymbols: 10
    OutputSamplesPerSymbol: 8
                      Gain: 1
```

```
rxrcfilt = comm.RaisedCosineReceiveFilter
```

```
rxrcfilt = 
  comm.RaisedCosineReceiveFilter with properties:

                    Shape: 'Square root'
            RolloffFactor: 0.2000
      FilterSpanInSymbols: 10
    InputSamplesPerSymbol: 8
         DecimationFactor: 8
         DecimationOffset: 0
                     Gain: 1
```

```
groupDelay = order(txrcfilt)/2 + order(rxrcfilt)/2
```

```
groupDelay = 80
```

Input Arguments

`rcfilter` — Input filter

`comm.RaisedCosineReceiveFilter` System object | `comm.RaisedCosineTransmitFilter` System object

Input filter, specified as one of these of filter System objects.

- `comm.RaisedCosineReceiveFilter`
- `comm.RaisedCosineTransmitFilter`

Output Arguments

`filtOrder` — Filter order

scalar

Filter order, returned as a scalar. The order depends on the filter structure and the reference double-precision floating-point coefficients.

Data Types: `double`

See Also

Functions
`coeffs` | `info`

Objects
`comm.RaisedCosineReceiveFilter` | `comm.RaisedCosineTransmitFilter`

Introduced in R2013b

show

Package: `comm`

Show scope window

Syntax

```
show(scope)
```

Description

`show(scope)` shows the window of the System object scope.

Examples

Hide and Show Scope

Create a `comm.ConstellationDiagram` object.

```
scope = comm.ConstellationDiagram;
```

Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```

Hide the constellation diagram scope window again.

```
if(isVisible(scope))
    hide(scope)
end
```

Hide and Show Constellation Diagram

Generate a 16-QAM reference constellation and a signal to display.

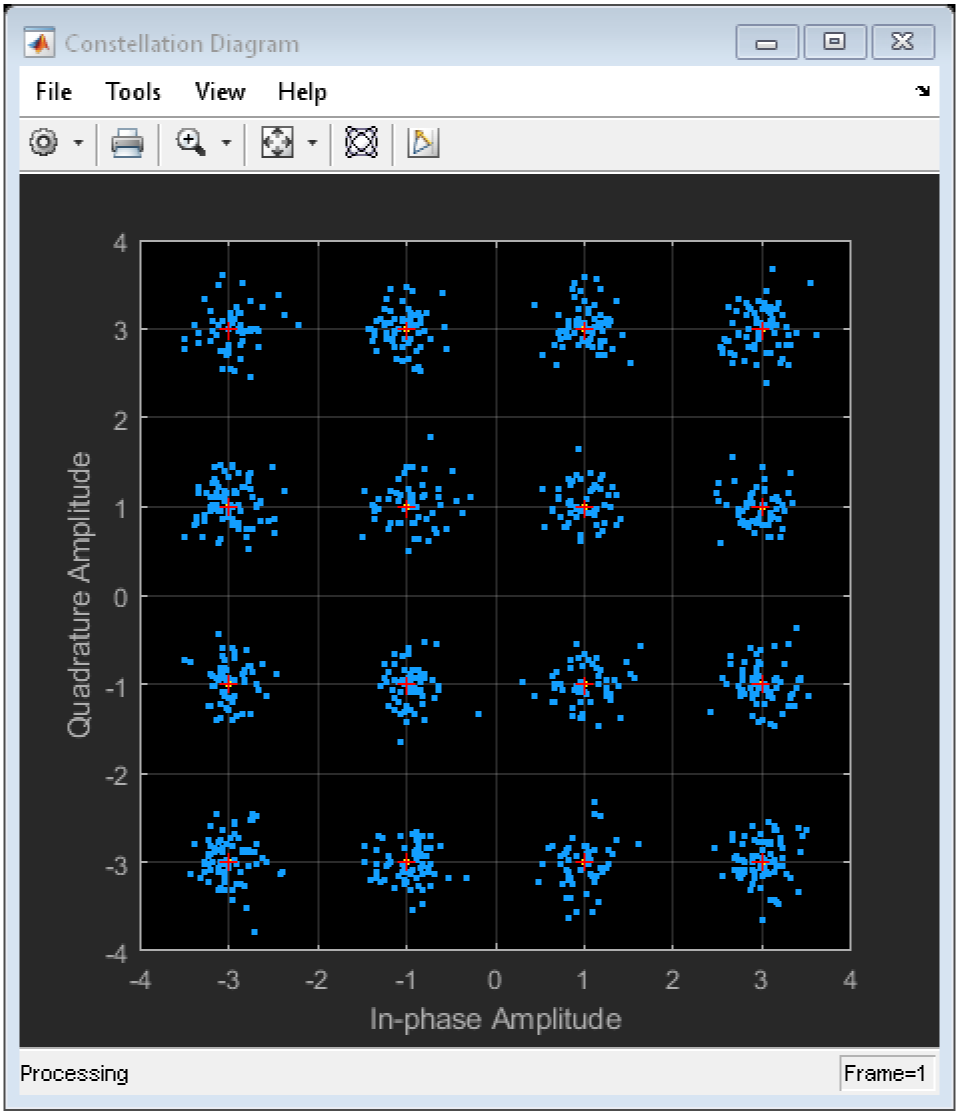
```
M = 16;
xRef = (0:M-1)';
refConst = qammod(xRef,M);
signal = randi([0 M-1],1000,1);
```

Create a constellation diagram System object™, specifying the constellation reference points and axes limits using name-value pairs.

```
scope = comm.ConstellationDiagram('ReferenceConstellation',refConst, ...
    'XLimits',[-4 4],'YLimits',[-4 4]);
```

Modulate the random data signal using QAM. Add Gaussian white noise to the QAM symbols. Display the QAM symbols and noisy symbols with the constellation diagram object.

```
sym = qammod(signal,M);
rcv = awgn(sym,20,'measured');
scope([sym rcv]);
```

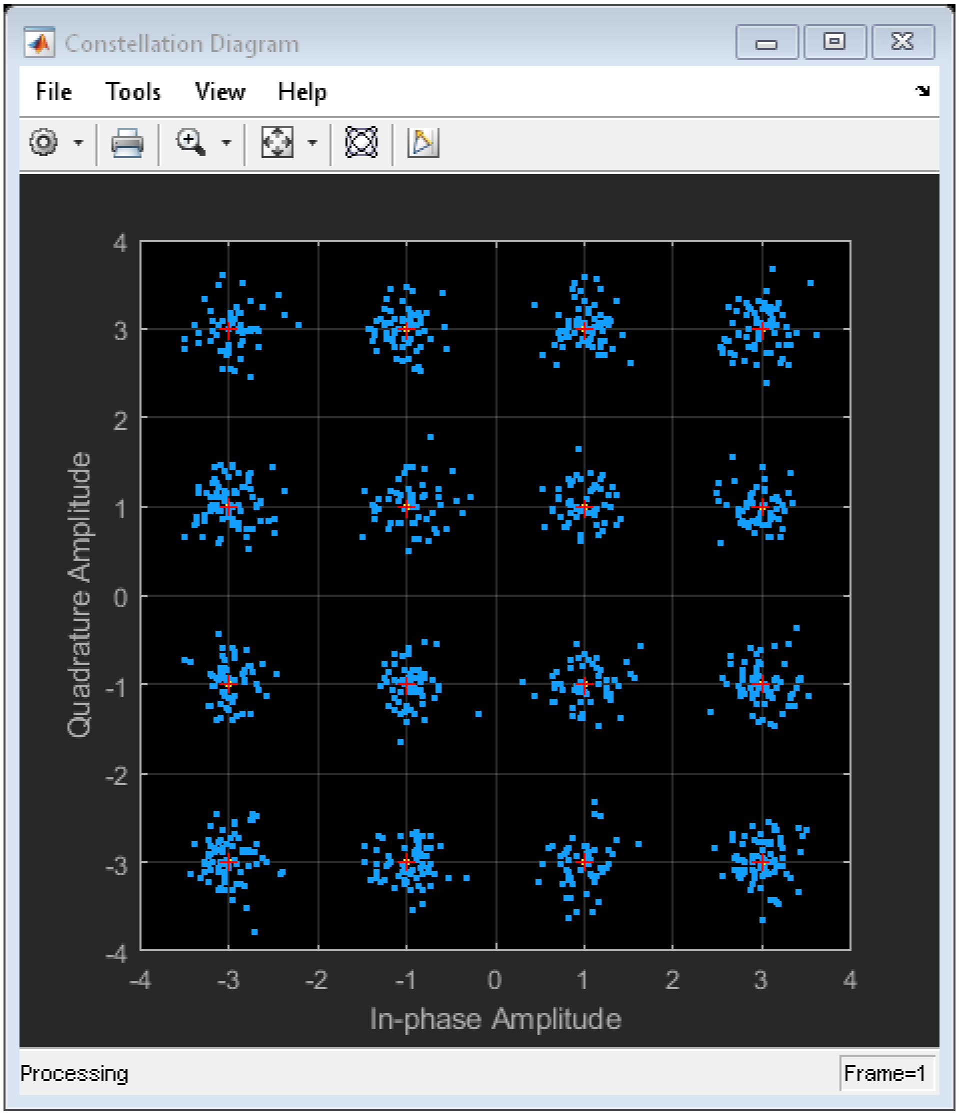


Hide the constellation diagram scope window.

```
if(isVisible(scope))
    hide(scope)
end
```

Show the constellation diagram scope window.

```
if(~isVisible(scope))
    show(scope)
end
```



Clear the workspace variables.

```
clear scope sym rcv M refConst signal xRef
```

Input Arguments

scope — Scope System object

scope System object

Scope System object, specified as a `comm.ConstellationDiagram` System object.

Example: `scope = comm.ConstellationDiagram;`

See Also

Functions
`hide` | `isVisible`

Objects
`comm.ConstellationDiagram`

Introduced in R2013a

info

Package: `comm`

Characteristic information about fading channel object

Syntax

```
infostruct = info(obj)
```

Description

`infostruct = info(obj)` returns a structure containing characteristic information about the fading channel System object.

Examples

Get comm.RayleighChannel Info

Use the `info` object function to get information from a `comm.RayleighChannel` object.

Create a Rayleigh channel object and some data to pass through the channel.

```
rayleighchan = comm.RayleighChannel('SampleRate',1000,'PathDelays',[0 0],'AveragePathGains',[0 0
```

```
rayleighchan = 
  comm.RayleighChannel with properties:

             SampleRate: 1000
             PathDelays: [0 0]
       AveragePathGains: [0 0]
     NormalizePathGains: true
    MaximumDopplerShift: 1.0000e-03
        DopplerSpectrum: [1x1 struct]

  Show all properties
```

```
data = randi([0 1],600,1);
```

Check the Rayleigh channel object information

```
info(rayleighchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: [2x1 double]
          NumSamplesProcessed: 0
```

Pass data through the channel and check the object information again.

```
rayleighchan(data);
info(rayleighchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: [2x1 double]
          NumSamplesProcessed: 600
```

Release the object so you can update attributes. Add a 1.5e-3 second path delay to the second delay path.

```
release(rayleighchan)
rayleighchan.PathDelays = [0 1.5e-3]
```

```
rayleighchan = 
  comm.RayleighChannel with properties:

             SampleRate: 1000
             PathDelays: [0 0.0015]
       AveragePathGains: [0 0]
     NormalizePathGains: true
    MaximumDopplerShift: 1.0000e-03
        DopplerSpectrum: [1x1 struct]

  Show all properties
```

Pass data through the channel and check the object information again.

```
rayleighchan(data);
info(rayleighchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 6
    ChannelFilterCoefficients: [2x16 double]
          NumSamplesProcessed: 600
```

Get comm.RicianChannel Info

Use the `info` object function to get information from a `comm.RicianChannel` object.

Create a Rician channel object and some data to pass through the channel.

```
ricianchan = comm.RicianChannel('SampleRate',500)
```

```
ricianchan = 
  comm.RicianChannel with properties:

                SampleRate: 500
                PathDelays: 0
          AveragePathGains: 0
        NormalizePathGains: true
                   KFactor: 3
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 1.0000e-03
           DopplerSpectrum: [1x1 struct]

  Show all properties
```

```
data = randi([0 1],600,1);
```

Check the Rician channel object information

```
info(ricianchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 0
```

Pass data through the channel and check the object information again.

```
ricianchan(data);
info(ricianchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 600
```

Release the object so you can update attributes. Add a second path delay with a delay of 3.1e-3 second and an average path gain of -3 dB.

```
release(ricianchan)
ricianchan.PathDelays = [0 3.1e-3];
ricianchan.AveragePathGains = [0 -3]
```

```
ricianchan = 
  comm.RicianChannel with properties:

                SampleRate: 500
                PathDelays: [0 0.0031]
          AveragePathGains: [0 -3]
        NormalizePathGains: true
                   KFactor: 3
    DirectPathDopplerShift: 0
    DirectPathInitialPhase: 0
       MaximumDopplerShift: 1.0000e-03
           DopplerSpectrum: [1x1 struct]

  Show all properties
```

Pass data through the channel and check the object information again.

```
ricianchan(data);
info(ricianchan)
```

```
ans = struct with fields:
           ChannelFilterDelay: 6
```

```
    ChannelFilterCoefficients: [2x16 double]
          NumSamplesProcessed: 600
```

Get comm.MIMOChannel Info

Use the `info` object function to get information from a `comm.MIMOChannel` object.

Create a MIMO channel object and some data to pass through the channel.

```
mimo = comm.MIMOChannel('SampleRate',1000);
data = randi([0 1],600,2);
```

Check the MIMO channel object information

```
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 0
```

Pass data through the channel and check the object information again.

```
mimo(data);
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 0
    ChannelFilterCoefficients: 1
          NumSamplesProcessed: 600
```

Release the object so you can update attributes. Add a `2.5e-3` second path delay. Recheck the object information.

```
release(mimo)
mimo.PathDelays = 2.5e-3;
info(mimo)
```

```
ans = struct with fields:
           ChannelFilterDelay: 5
    ChannelFilterCoefficients: [1x16 double]
          NumSamplesProcessed: 0
```

Model MIMO Channel Using Sum-of-Sinusoids Technique

Create a MIMO channel object and pass data through it using the sum-of-sinusoids technique. The example demonstrates how the channel state is maintained in cases in which data is discontinuously transmitted.

Define the overall simulation time and three time segments for which data will be transmitted. In this case, the channel is simulated for 1 s with a 1000 Hz sampling rate. One 1000-sample, continuous data sequence is transmitted at time 0. Three 100-sample data packets are transmitted at time 0.1 s, 0.4 s, and 0.7 s.

```
t0 = 0:0.001:0.999;   % Transmission 0
t1 = 0.1:0.001:0.199; % Transmission 1
t2 = 0.4:0.001:0.499; % Transmission 2
t3 = 0.7:0.001:0.799; % Transmission 3
```

Generate random binary data corresponding to the previously defined time intervals.

```
d0 = randi([0 1],1000,2);  % 1000 samples
d1 = randi([0 1],100,2);   % 100 samples
d2 = randi([0 1],100,2);   % 100 samples
d3 = randi([0 1],100,2);   % 100 samples
```

Create a flat fading 2x2 MIMO channel System object with the `Sum of sinusoids` fading technique. So that results can be repeated, specify a seed using a name-value pair. As the `InitialTime` property is not specified, the fading channel will be simulated from time 0. Enable the path gains output port.

```
mimoChan1 = comm.MIMOChannel('SampleRate',1000, ...
    'MaximumDopplerShift',5, ...
    'RandomStream','mt19937ar with seed', ...
    'Seed',17, ...
    'FadingTechnique','Sum of sinusoids', ...
    'PathGainsOutputPort',true);
```

Create a clone of the MIMO channel System object. Set the `InitialTimeSource` property to `Input port` so that the fading channel offset time can be specified as an input argument to the `mimoChan` function.

```
mimoChan2 = clone(mimoChan1);
mimoChan2.InitialTimeSource = 'Input port';
```

Pass random binary data through the first channel object, `mimoChan1`. Data is transmitted over all 1000 time samples. For this example, only the complex path gain is needed.

```
[~,pg0] = mimoChan1(d0);
```

Pass random data through the second channel object, `mimoChan2`, where the initial time offsets are provided as input arguments.

```
[~,pg1] = mimoChan2(d1,0.1);
[~,pg2] = mimoChan2(d2,0.4);
[~,pg3] = mimoChan2(d3,0.7);
```

Compare the number of samples processed by the two channels using the `info` method. You can see that 1000 samples were processed by `mimoChan1` while only 300 were processed by `mimoChan2`.

```
G = info(mimoChan1);
H = info(mimoChan2);
[G.NumSamplesProcessed H.NumSamplesProcessed]
```

```
ans = 1×2
```

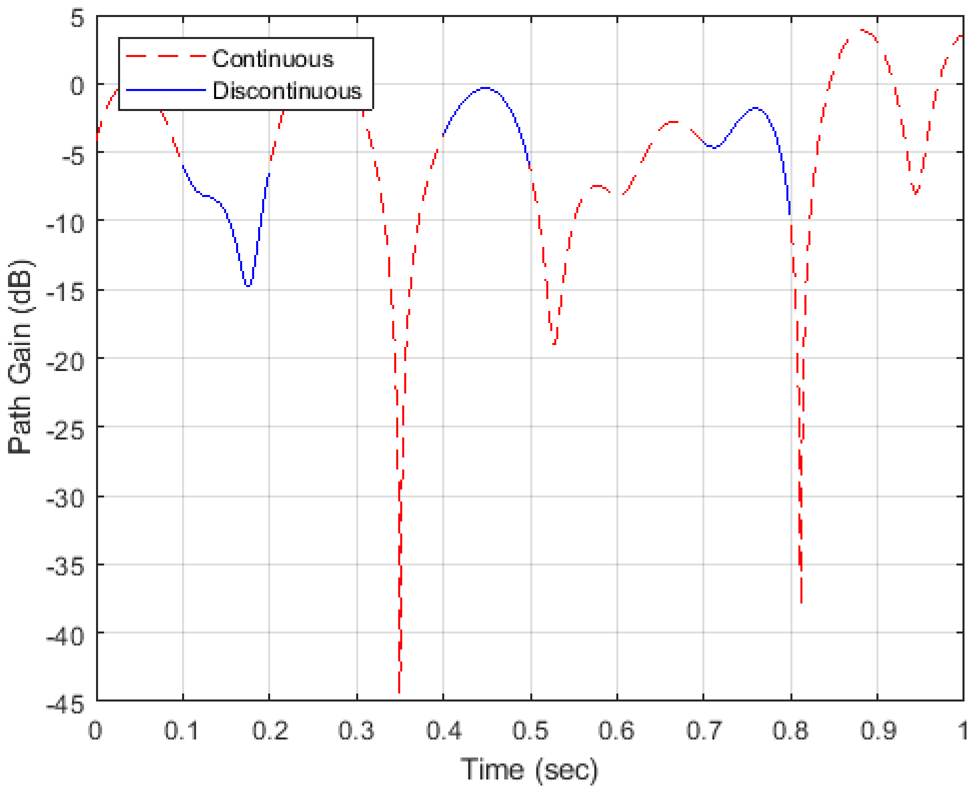
```
        1000         300
```

Convert the path gains into decibels for the path corresponding to the first transmit and first receive antenna.

```
pathGain0 = 20*log10(abs(pg0(:,1,1,1)));
pathGain1 = 20*log10(abs(pg1(:,1,1,1)));
pathGain2 = 20*log10(abs(pg2(:,1,1,1)));
pathGain3 = 20*log10(abs(pg3(:,1,1,1)));
```

Plot the path gains for the continuous and discontinuous cases. Observe that the gains for the three segments perfectly match the gain for the continuous case. The alignment of the two highlights that the sum-of-sinusoids technique is ideally suited to the simulation of packetized data as the channel characteristics are maintained even when data is not transmitted.

```
plot(t0,pathGain0,'r--')
hold on
plot(t1,pathGain1,'b')
plot(t2,pathGain2,'b')
plot(t3,pathGain3,'b')
grid
xlabel('Time (sec)')
ylabel('Path Gain (dB)')
legend('Continuous','Discontinuous','location','nw')
```



Input Arguments

`obj` — System object to get information from

System object

System object to get information from, specified as a `comm.MIMOChannel`, `comm.RayleighChannel`, or `comm.RicianChannel` System object.

Output Arguments

`infostruct` — Structure containing object information

structure

Structure containing these fields with information about the System object.

`ChannelFilterDelay` — Channel filter delay

positive integer

Channel filter delay in samples, returned as a positive integer.

`ChannelFilterCoefficients` — Channel filter coefficients

matrix

Channel filter coefficients, returned as a matrix. The coefficient matrix is used to convert path gains to channel filter tap gains for each sample and each pair of transmit and receive antennas.

`NumSamplesProcessed` — Number of samples processed by the channel object

positive integer

Number of samples processed by the channel object since the last reset, returned as a positive integer.

`LastFrameTime` — Last frame ending time

positive scalar

Last frame ending time in seconds, returned as a positive scalar. Use this value to confirm the simulation time.

Dependencies

This property applies when the `FadingTechnique` property is `'Sum of sinusoids'` and the `InitialTimeSource` property is `'Input port'`.

See Also

Objects
`comm.MIMOChannel` | `comm.RayleighChannel` | `comm.RicianChannel`

Introduced in R2012a

constellation

Package: `comm`

Calculate or plot ideal signal constellation

Syntax

```
symbols = constellation(obj)
constellation(obj)
```

Description

`symbols = constellation(obj)` returns the numerical values of the constellation.

`constellation(obj)` generates a constellation plot for the object.

Examples

Plot QPSK Reference Constellation

Create a QPSK modulator.

```
mod = comm.QPSKModulator;
```

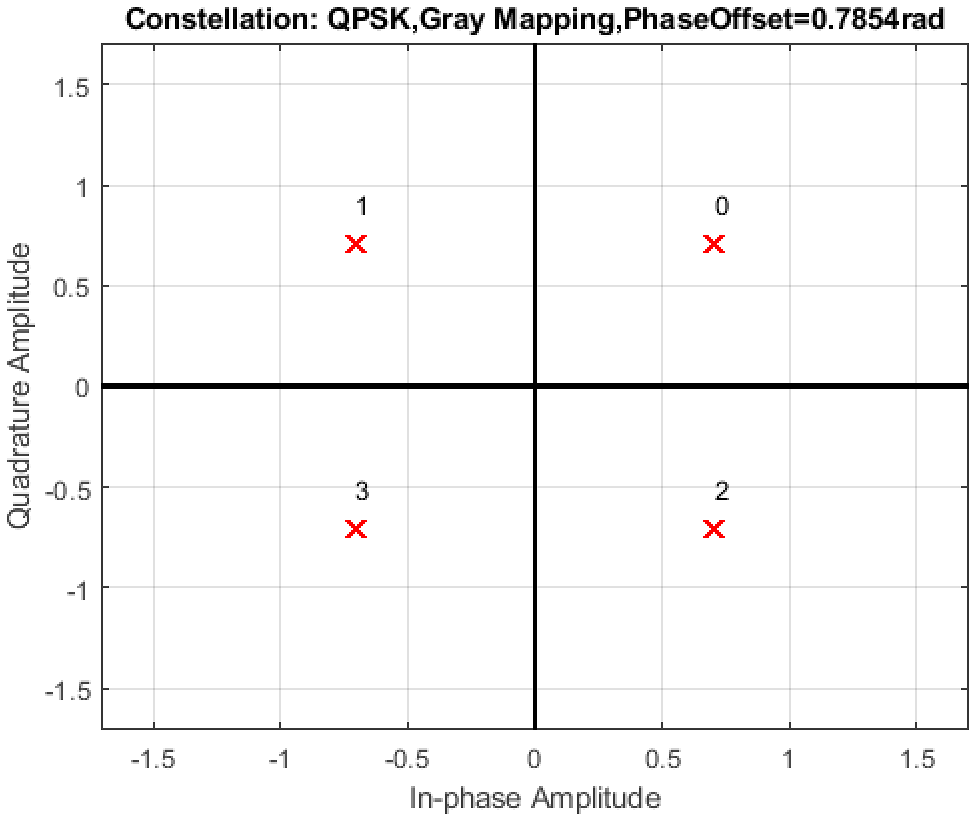
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 4×1 complex

   0.7071 + 0.7071i
  -0.7071 + 0.7071i
  -0.7071 - 0.7071i
   0.7071 - 0.7071i
```

Plot the constellation.

```
constellation(mod)
```

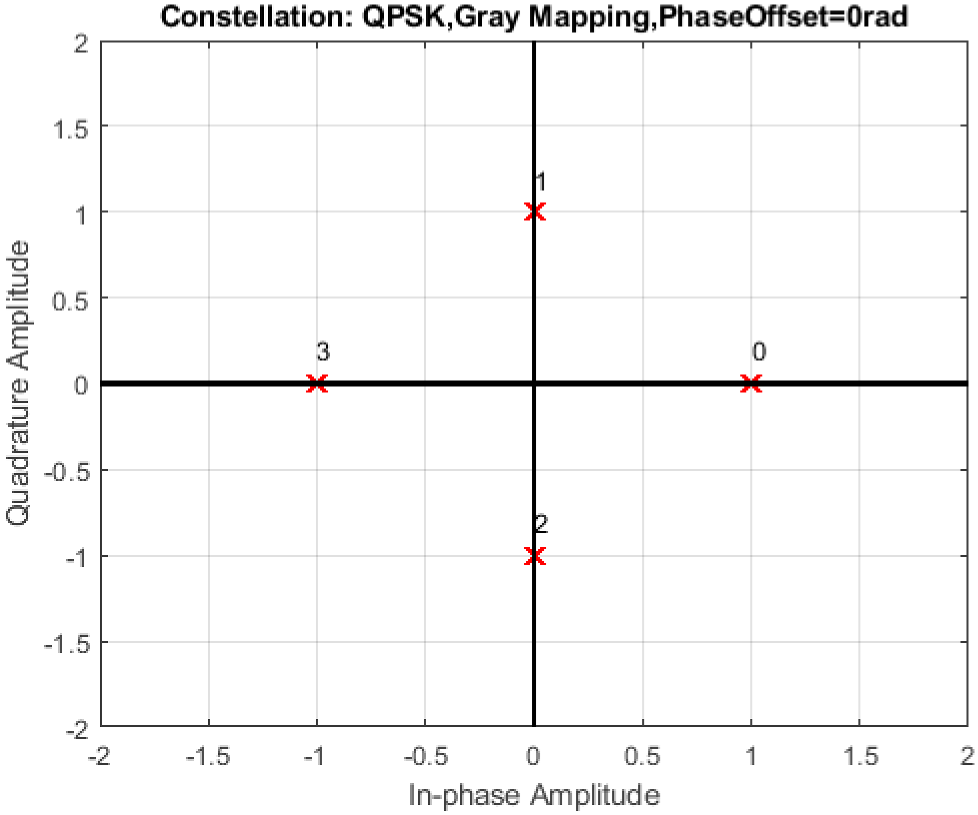


Create a PSK demodulator having 0 phase offset.

```
demod = comm.QPSKDemodulator('PhaseOffset',0);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



Calculate BPSK Modulator Reference Constellation

Create a BPSK Modulator System object™ and calculate the reference constellation values.

Create a `comm.BPSKModulator` System object.

```
h = comm.BPSKModulator;
```

Calculate and display the reference constellation values by calling the `constellation` function.

```
refC = constellation(h)
```

```
refC = 2×1 complex

   1.0000 + 0.0000i
  -1.0000 + 0.0000i
```

Plot PSK Reference Constellation

Create a PSK modulator.

```
mod = comm.PSKModulator;
```

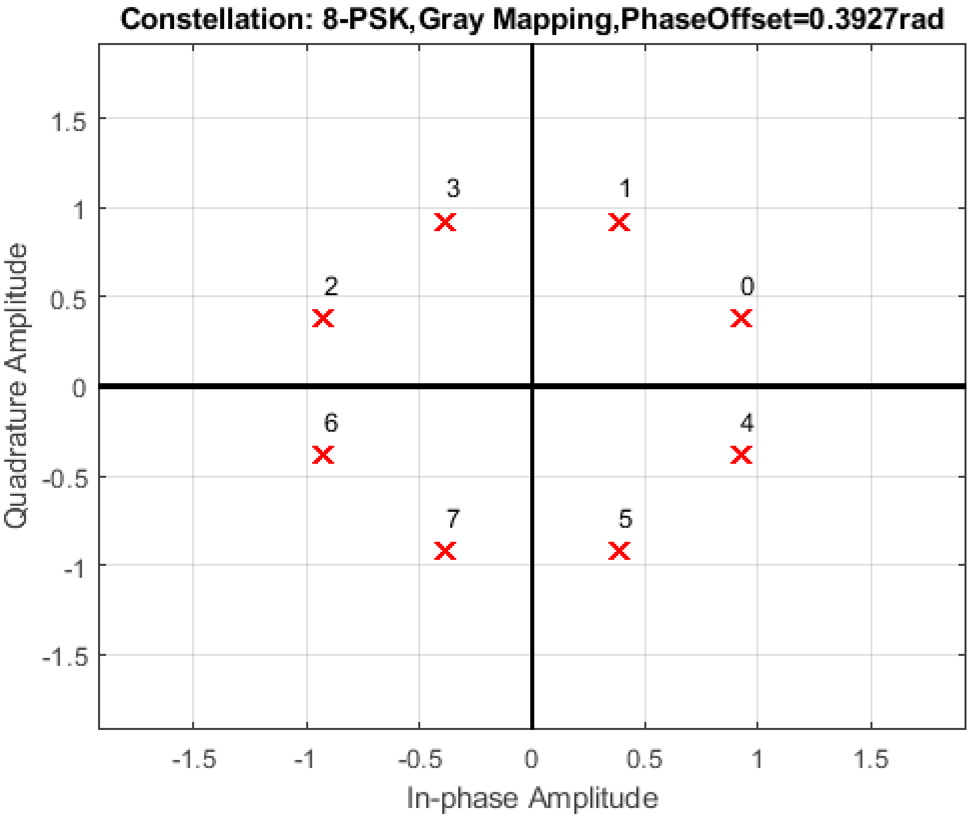
Determine the reference constellation points.

```
refC = constellation(mod)
```

```
refC = 8×1 complex

   0.9239 + 0.3827i
   0.3827 + 0.9239i
  -0.3827 + 0.9239i
  -0.9239 + 0.3827i
  -0.9239 - 0.3827i
  -0.3827 - 0.9239i
   0.3827 - 0.9239i
   0.9239 - 0.3827i
```

Plot the constellation.

```
constellation(mod)
```

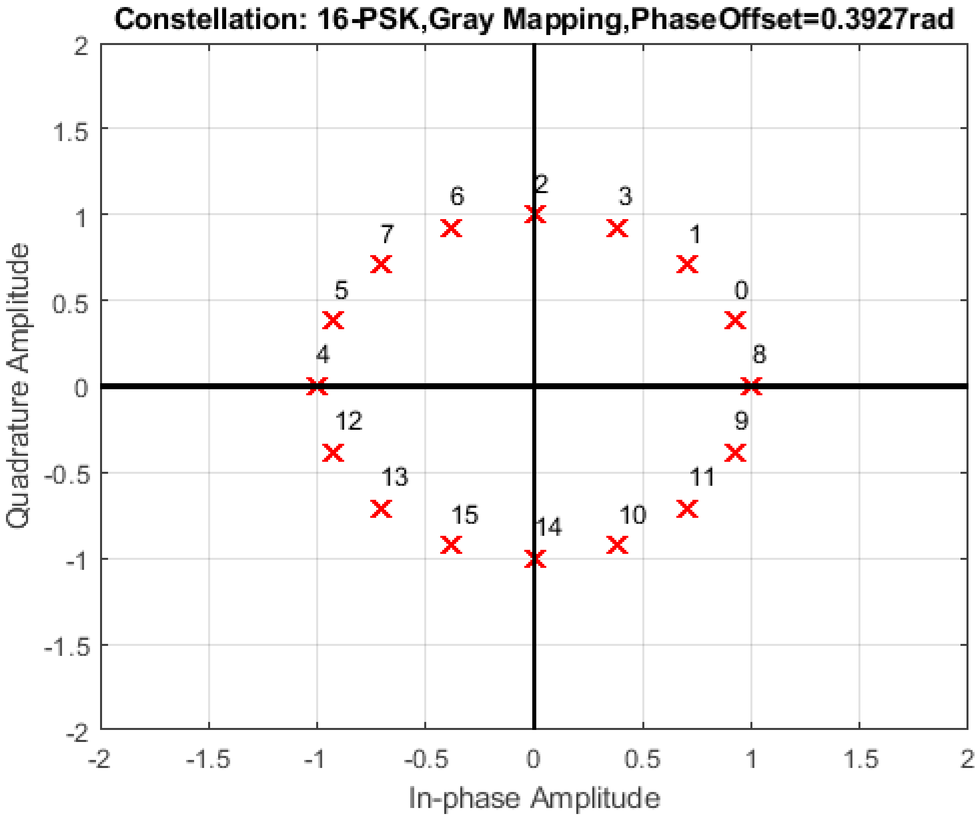


Create a PSK demodulator having modulation order 16.

```
demod = comm.PSKDemodulator(16);
```

Plot its reference constellation. The `constellation` method works for both modulator and demodulator objects.

```
constellation(demod)
```



Input Arguments

obj — System object to return constellation for
System object

System object to return constellation for, specified as a modulator System object.

Data Types: `object`

Output Arguments

symbols — Constellation points
complex vector (default)

Constellation points, returned as a complex vector.

Data Types: `double` | `single`
Complex Number Support: Yes

See Also

Objects
`comm.BPSKModulator` | `comm.OQPSKModulator` | `comm.PSKModulator`

Introduced in R2012a

info

Package: comm

Provide dimensioning information for OFDM modulator

Syntax

```
infostruct = info(hMod)
```

Description

`infostruct = info(hMod)` returns the input, pilot, and output data dimensions for the specified OFDM modulator System object.

Examples

Determine OFDM Modulator Data Dimensions

Get the OFDM modulator data dimensions by using the `info` object function.

Construct an OFDM modulator System object™ with user-specified pilot indices, an inserted DC null, and specify two transmit antennas.

```
hMod = comm.OFDMModulator('NumGuardBandCarriers',[4;3], ...
    'PilotInputPort',true, ...
    'PilotCarrierIndices',cat(3,[12; 26; 40; 54], ...
    [11; 25; 39; 53]), ...
    'InsertDCNull',true, ...
    'NumTransmitAntennas',2);
```

Use the `info` object function to get the modulator input data, pilot input data, and output data sizes.

```
info(hMod)
```

```
ans = struct with fields:
     DataInputSize: [48 1 2]
    PilotInputSize: [4 1 2]
        OutputSize: [80 2]
```

Input Arguments

hMod — OFDM modulator

System object

OFDM modulator, specified as a `comm.OFDMModulator` System object.

Output Arguments

`infostruct` — Dimensions of structure for OFDM modulator

struct

Dimensions of structure for OFDM modulator, returned as a structure containing these fields.

`DataInputSize` — Dimensions of input data

3-D array

Dimensions of input data, returned as a 3-D array of numeric values. The dimensions of this field are N_{data}-by-N_{sym}-by-N_{t}, where N_{data} is the number of data subcarriers such that $N_{\text{data}} = N_{\text{FFT}} - N_{\text{leftG}} - N_{\text{rightG}} - N_{\text{DCNull}} - N_{\text{pilot}} - N_{\text{custNull}}$.

For variable definitions, see Variable Definitions.

`PilotInputSize` — Dimensions of the pilot input data

3-D array

Dimensions of the pilot input array, returned as a 3-D array of numeric values. The output dimensions of this field are N_{pilot}-by-N_{sym}-by-N_{t}.

`OutputSize` — Dimensions of modulator output data

3-D array

Dimensions of the modulator output data, returned as a 3-D array of numeric values. The dimensions of this field are $((N_{\text{FFT}} + N_{\text{CP}})\times N_{\text{sym}})$-by-$N_{\text{t}}$.

For variable definitions, see Variable Definitions.

Data Types: `struct`

More About

List of Variables

The variables mentioned in this table are defined in this table:

Variable Definitions

Variable	Description
N_{FFT}	Number of subcarriers
N_{leftG}	Number of subcarriers in the left guard band
N_{rightG}	Number of subcarriers in the right guard band
N_{DCNull}	Number of subcarriers in the DC null (either `0` or `1`)
N_{pilot}	Number of pilot subcarriers
N_{custNull}	Number of subcarriers used for custom nulls (applies only when the `PilotCarrierIndices` property of input `hMod` is a 3-D array)
N_{t}	Number of transmit antennas
N_{CP}	Length of cyclic prefix.

See Also

Functions
`ofdmmod`

Objects
`comm.OFDMModulator`

Introduced in R2014a

info

Package: `comm`

Return characteristic information about channel filter

Syntax

```
infostruct = info(chanFilt)
```

Description

`infostruct = info(chanFilt)` returns a structure containing characteristic information about the `comm.ChannelFilter` System object.

Examples

Determine Channel Filter Structure Dimensions

In a distributed MIMO system, send the same signal from two geographically separate transmitters (Tx) and combine the received signals at one single receiver (Rx) to explore spatial diversity. The two Txs are not co-located and they experience different multipath (path delays) to the Rx. Specify the path delays respectively.

```
delay1 = [0 1.5 2.3 5.2 6.6];
delay2 = [0 3.7 6.2];
```

Configure one channel filter object per Tx.

```
chanFilt1 = comm.ChannelFilter('PathDelays', delay1);
chanFilt2 = comm.ChannelFilter('PathDelays', delay2);
```

Use the `info` object function to get the ChannelFilterDelay and ChannelFilterCoefficients.

```
info(chanFilt1)
```

```
ans = struct with fields:
           ChannelFilterDelay: 6
    ChannelFilterCoefficients: [5x21 double]
```

```
info(chanFilt2)
```

```
ans = struct with fields:
           ChannelFilterDelay: 4
    ChannelFilterCoefficients: [3x19 double]
```

Input Arguments

`chanFilt` — Channel filter
System object

Channel filter, specified as a `comm.ChannelFilter` System object.

Output Arguments

`infostruct` — Characteristic information about channel filter
struct

Characteristic information about channel filter, returned as a structure containing these fields:

`ChannelFilterDelay` — Channel filter delay
positive real scalar

Channel filter delay, returned as a positive real scalar.

Data Types: `double`

`ChannelFilterCoefficients` — Channel filter coefficients
vector | matrix

Channel filter coefficients, returned as a vector or a matrix.

Data Types: `double`

See Also

Objects
`comm.ChannelFilter`

Introduced in R2020b

showResourceMapping

Package: `comm`

Show the subcarrier mapping of the OFDM symbols created by the OFDM modulator System object

Syntax

```
showResourceMapping(hMod)
showResourceMapping(hMod,ci)
```

Description

`showResourceMapping(hMod)` shows a visualization of the subcarrier mapping for the OFDM symbols created by the OFDM modulator System object.

`showResourceMapping(hMod,ci)` uses the values in input `ci` to number the subcarrier indices that the function displays. The subcarrier indices are numbered from 1 to `ci`N_{FFT}, where N_{FFT} is the number of FFT points.

Examples

Create and Modify OFDM Modulator

Create and display an OFDM modulator System object™ with default property values.

```
hMod = comm.OFDMModulator
```

```
hMod = 
  comm.OFDMModulator with properties:

               FFTLength: 64
    NumGuardBandCarriers: [2x1 double]
             InsertDCNull: false
           PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 1
     NumTransmitAntennas: 1
```

Modify the number of subcarriers and symbols.

```
hMod.FFTLength = 128;
hMod.NumSymbols = 2;
```

Verify that the number of subcarriers and the number of symbols changed.

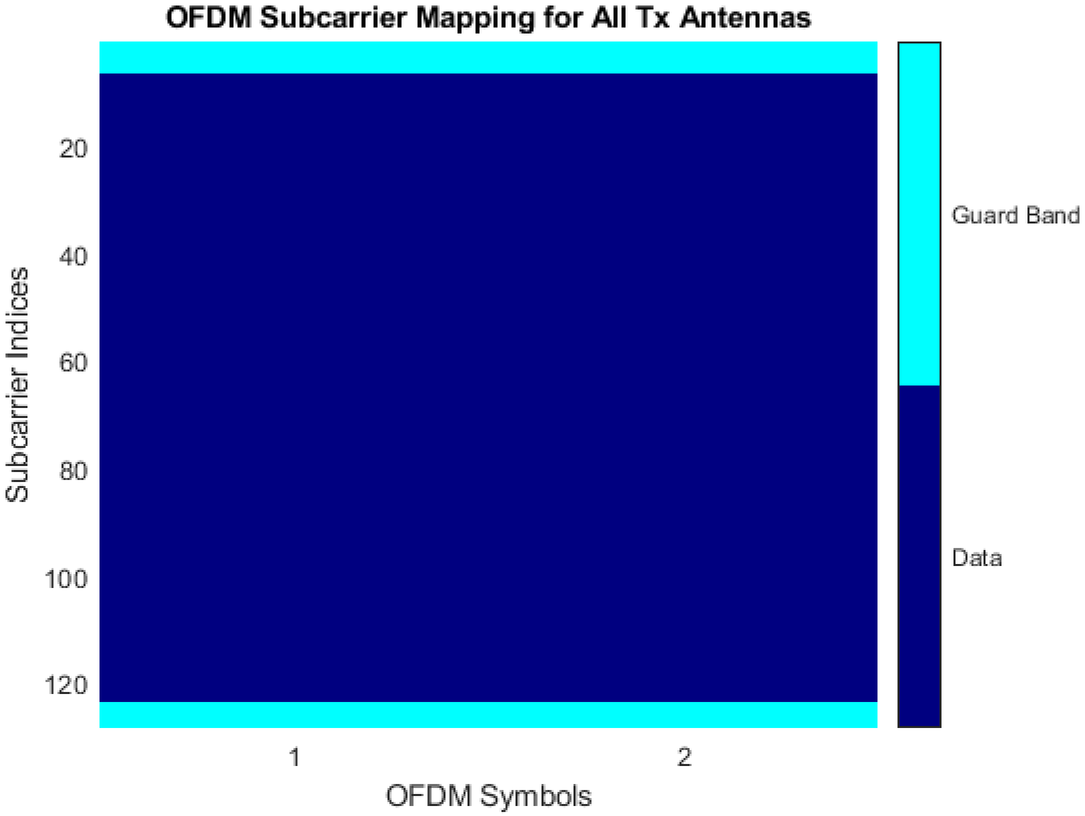
```
disp(hMod)
```

```
  comm.OFDMModulator with properties:
```

```
               FFTLength: 128
    NumGuardBandCarriers: [2x1 double]
            InsertDCNull: false
           PilotInputPort: false
      CyclicPrefixLength: 16
               Windowing: false
              NumSymbols: 2
     NumTransmitAntennas: 1
```

Use the `showResourceMapping` object function to show the mapping of data, pilot, and null subcarriers in the time-frequency space.

```
showResourceMapping(hMod)
```



Input Arguments

hMod — System object to visualize
`comm.OFDMModulator` System object

OFDM modulator, specified as a `comm.OFDMModulator` System object.

ci — Subcarrier indices to visualize
`[1,…,`N_{FFT}`]` (default) | two-element row vector of integers

Subcarrier indices to visualize, specified as a two-element row vector of integers. `ci` should satisfy diff(`ci`) = $N_{\text{FFT}} - 1$.

See Also

Functions
`ofdmmod`

Objects
`comm.OFDMModulator`

Introduced in R2014a

info

Package: `comm`

Characteristic information about ray-tracing channel

Syntax

`chanInfo = info(rtchan)`

Description

`chanInfo = info(rtchan)` returns a structure containing characteristic information about the input ray-tracing channel.

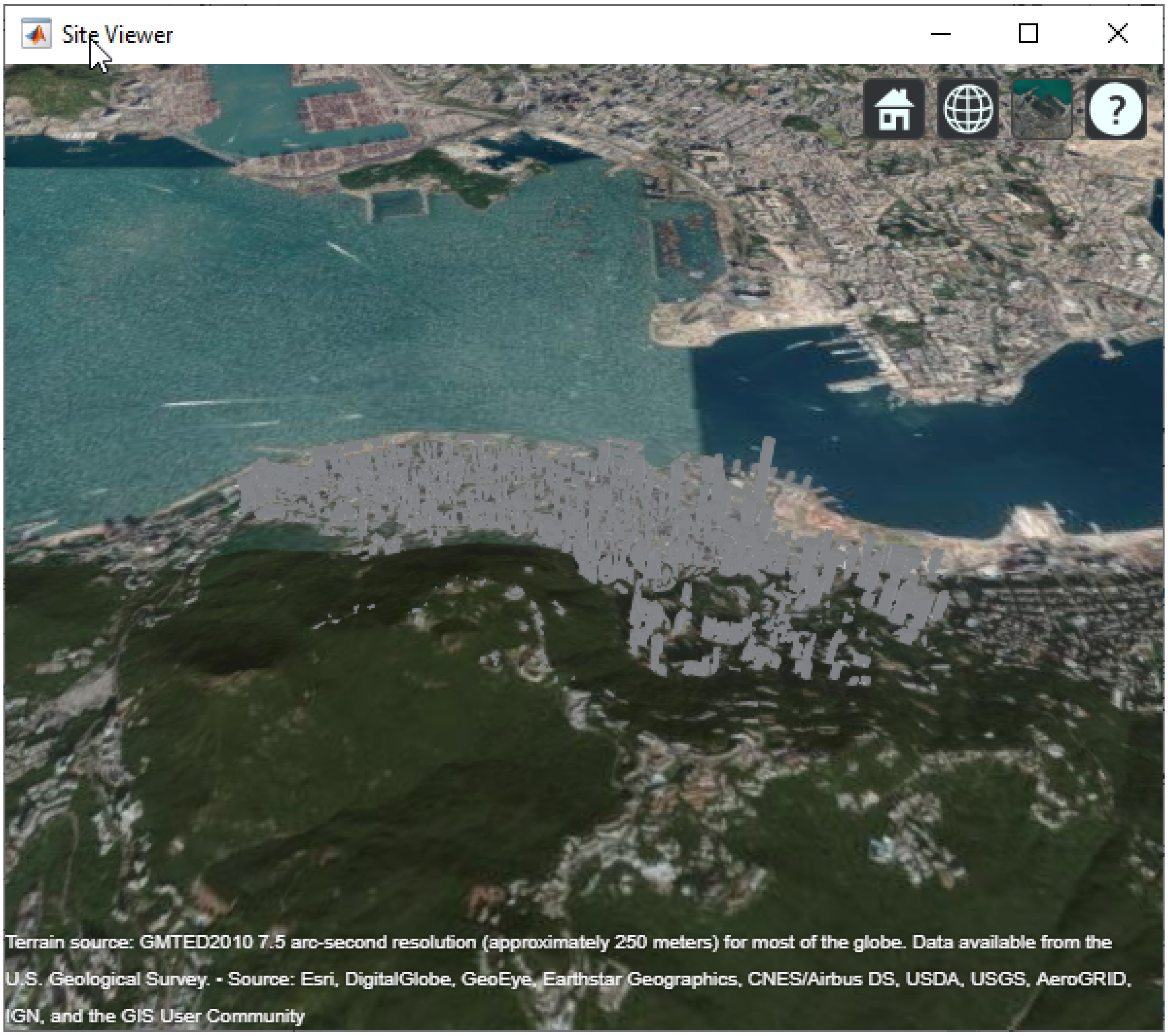
Examples

Return Information for Ray-Tracing Channel Between Two Sites in Hong Kong

Perform ray tracing between two sites in Hong Kong, China, build a multipath channel model using the ray-tracing result, and view the ray-tracing channel information.

Create a Site Viewer map display of buildings in Hong Kong. For more information about the osm file, see [1] on page 4-0   .

```
sv = siteviewer("Buildings","hongkong.osm");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create transmitter and receiver sites.

```
tx = txsite("Latitude",22.2789,"Longitude",114.1625, ...
    "AntennaAngle",30, ... % azimuth
    "AntennaHeight",10,"TransmitterFrequency",28e9);
rx = rxsite("Latitude",22.2799,"Longitude",114.1617, ...
    "AntennaAngle",120, ... % azimuth
    "AntennaHeight",1);
```

Perform ray tracing to find rays with up to 2 reflections.

```
rays = raytrace(tx,rx,"NumReflections",[0 1 2]);
```

Create a channel model by using the transmitter site, receiver site, and calculated rays between the sites. Return information from the ray-tracing channel.

```
rtchan = comm.RayTracingChannel(rays{1},tx,rx);
info(rtchan)
```

```
ans = struct with fields:
             CarrierFrequency: 2.8000e+10
             CoordinateSystem: 'Geographic'
        TransmitArrayLocation: [3×1 double]
         ReceiveArrayLocation: [3×1 double]
          NumTransmitElements: 1
           NumReceiveElements: 1
           ChannelFilterDelay: 4
    ChannelFilterCoefficients: [7×20 double]
          NumSamplesProcessed: 0
```

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

`rtchan` — Ray-tracing channel

`comm.RayTracingChannel` System object

Ray-tracing channel, specified as a `comm.RayTracingChannel` System object.

Output Arguments

`chanInfo` — Ray tracing channel characteristic information

structure

Ray tracing channel characteristic information, returned as a structure containing these fields. The ray tracing channel is specified by the input `rtchan`.

`CarrierFrequency` — Carrier frequency

positive scalar

Carrier frequency in Hz, returned as a positive scalar.

`CoordinateSystem` — Coordinate system

`'Cartesian'` | `'Geographic'`

Coordinate system, returned as `'Cartesian'` or `'Geographic'`. Any `comm.Ray` objects, specified by the `PropagationRays` property of the `rtchan` input, that have their `PathSpecification` property set to `'Locations'`, must have the same `CoordinateSystem` property setting.

Dependencies

This property applies when at least one `comm.Ray` object in the `PropagationRays` property of the `rtchan` input has `PathSpecification` set to `'Locations'`.

`TransmitArrayLocation` — Transmit array location

three element column vector

Transmit array location, returned as a three element column vector. Any `comm.Ray` objects, specified by the `PropagationRays` property of the `rtchan` input, that have their `PathSpecification` property set to `'Locations'`, must have the same `TransmitterLocation` property setting.

Dependencies

This property applies when at least one `comm.Ray` object in the `PropagationRays` property of the `rtchan` input has `PathSpecification` set to `'Locations'`.

`ReceiveArrayLocation` — Receive array location

three element column vector

Receive array location, returned as a three element column vector. Any `comm.Ray` objects, specified by the `PropagationRays` property of the `rtchan` input, that have their `PathSpecification` property set to `'Locations'`, must have the same `ReceiverLocation` property setting.

Dependencies

This property applies when at least one `comm.Ray` object in the `PropagationRays` property of the `rtchan` input has `PathSpecification` set to `'Locations'`.

`NumTransmitElements` — Number of elements in transmit array

positive integer

Number of elements in the transmit array, returned as a positive integer.

`NumReceiveElements` — Number of elements in receive array

positive integer

Number of elements in the receive array, returned as a positive integer.

`ChannelFilterDelay` — Channel filter delay

nonnegative integer

Channel filter delay in samples, returned as a nonnegative integer.

`ChannelFilterCoefficients` — Channel filter coefficients

N_P-by-N_H matrix

Channel filter coefficients, returned as an N_P-by-N_H matrix. This coefficient matrix is used to convert channel impulse responses to channel filter tap gains for each sample, and then for each pair of transmit and receive antenna elements. N_P is the number of paths (specifically, the number of rays as indicated by the length of the `PropagationRays` property of the `rtchan` input). N_H is the number of impulse response samples (specifically, the number of channel filter taps).

`NumSamplesProcessed` — Number of samples processed by channel object

nonnegative integer

Number of samples processed by the channel object since its last reset, returned as a nonnegative integer.

See Also

Objects
`arrayConfig` | `comm.Ray` | `comm.RayTracingChannel`

Functions
`showProfile`

Introduced in R2020b

showProfile

Package: `comm`

Plot temporal and spatial profiles of ray-tracing channel

Syntax

```
showProfile(rtchan)
showProfile(rtchan,'ArrayPattern',false)
```

Description

`showProfile(rtchan)` plots the power delay profile (PDP), angle of departure (AoD), and angle of arrival (AoA) information for the ray-tracing channels in a single figure with three subplots.

- The PDP subplot is derived from the propagation delay, path loss, phase shift, pattern gain at transmit array and pattern gain at receive array for each ray specified by the `PropagationRays` property of the `rtchan` input.
- The AoD and AoA subplots show the 3-D directions of the rays in the local coordinate system (LCS).
  - When the `TransmitArray` or `ReceiveArray` properties are objects from the “Phased Array System Toolbox” software, the AoD and AoA subplots also show the directivity pattern of the arrays.

`showProfile(rtchan,'ArrayPattern',false)` optionally turns off the directivity pattern in the AoD and AoA subplots. This option applies only when the `TransmitArray` or `ReceiveArray` property in the `comm.RayTracingChannel` System object are objects from the “Phased Array System Toolbox” software.

Examples

Show Temporal and Spatial Profiles of Ray-Tracing Channel

Perform ray-tracing between two sites, build a multipath channel model using the ray tracing result, and show the temporal and spatial profiles of the channel.

Create a transmitter site and a receiver site. Perform ray tracing to find rays with up to 2 reflections between the sites.

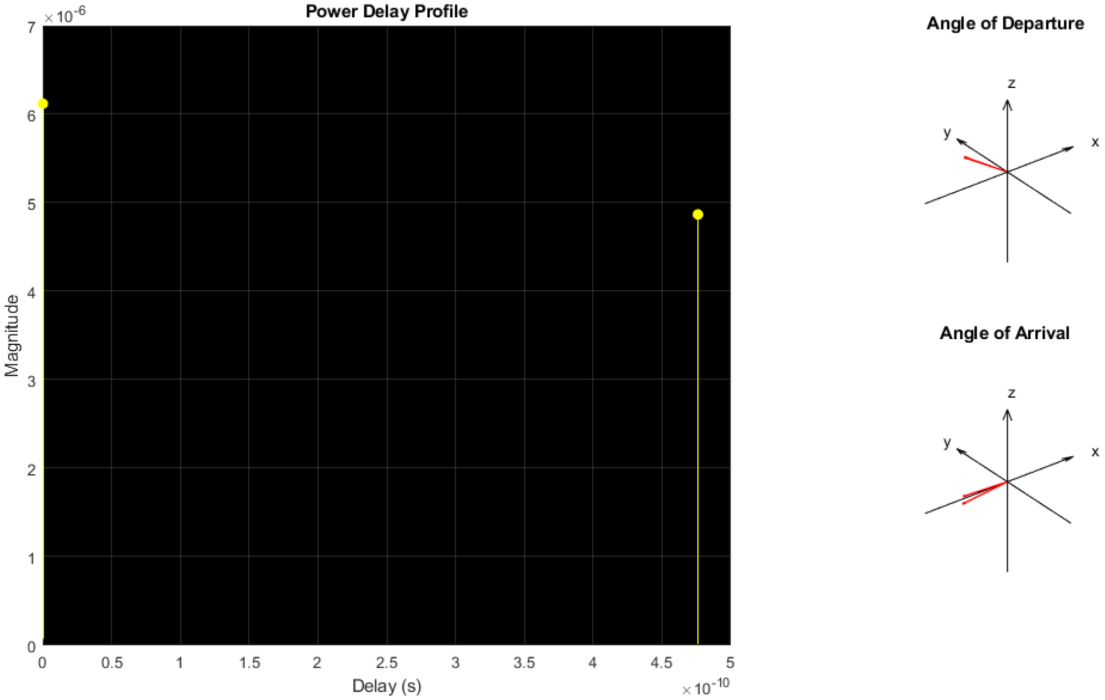
```
tx = txsite("Latitude",22.2789,"Longitude",114.1625, ...
    "AntennaAngle",30,"AntennaHeight",10,"TransmitterFrequency",28e9);
rx = rxsite("Latitude",22.2799,"Longitude",114.1617, ...
    "AntennaAngle",120,"AntennaHeight",1);
rays = raytrace(tx,rx,"NumReflections",[0 1 2]);
```

Create a channel model using transmitter site, receiver site, and calculated rays between the sites.

```
rtchan = comm.RayTracingChannel(rays{1},tx,rx);
```

Show temporal and spatial profiles of the channel.

```
showProfile(rtchan);
```



Input Arguments

`rtchan` — Ray-tracing channel

`comm.RayTracingChannel` System object

Ray-tracing channel, specified as a `comm.RayTracingChannel` System object.

See Also

Objects
`comm.Ray` | `comm.RayTracingChannel`

Functions
`info`

Introduced in R2020b

info

Package: `comm`

Characteristic information about carrier synchronizer

Syntax

```
infostruct = info(carrSynch)
```

Description

`infostruct = info(carrSynch)` returns a structure containing characteristic information for the `CarrierSynchronizer` System object.

Examples

Determine Carrier Synchronizer Loop Parameters

Create a carrier synchronizer object.

```
csync = comm.CarrierSynchronizer;
```

Determine the normalized pull-in range, the maximum frequency lock delay, and the maximum phase lock delay by using the `info` method.

```
syncInfo = info(csync)
```

```
syncInfo = struct with fields:
    NormalizedPullInRange: 0.0628
    MaxFrequencyLockDelay: 1.5787e+04
        MaxPhaseLockDelay: 130
```

The normalized pull-in range is 0.0628 rad/sec. Convert the pull-in range to Hz. This represents the maximum normalized frequency offset that can be corrected by the carrier synchronizer.

```
foffsetmax = syncInfo.NormalizedPullInRange/(2*pi)
```

```
foffsetmax = 0.0100
```

The time to acquire a frequency lock is 15787 s, and the time to acquire a phase lock is 130 s.

The overall acquisition time, `Tlock`, is well approximated by the sum of the frequency and phase lock terms.

```
Tlock = syncInfo.MaxFrequencyLockDelay + syncInfo.MaxPhaseLockDelay
```

```
Tlock = 1.5917e+04
```

Input Arguments

`carrSynch` — System object to get information from
System object

System object to get information from.

Output Arguments

`infostruct` — Structure containing object information
struct

Structure containing these fields with information about the System object.

`NormalizedPullInRange` — Normalized pull in range
scalar

Normalized pull in range in radians, returned as a scalar. `NormalizedPullInRange` is the largest frequency offset (rad), normalized by the loop bandwidth, for which the synchronizer can acquire lock.

`MaxFrequencyLockDelay` — Maximum frequency lock delay
positive integer

Maximum frequency lock delay, returned as a positive integer. `MaxFrequencyLockDelay` is the number of samples required for the synchronizer to acquire frequency lock.

`MaxPhaseLockDelay` — Maximum phase lock delay
positive integer

Maximum phase lock delay, returned as a positive integer. `MaxPhaseLockDelay` is the number of samples required for the synchronizer to acquire phase lock.

Data Types: `struct`

See Also

Objects
`comm.CarrierSynchronizer`

Introduced in R2015a

info

Package: `comm`

Characteristic information about the equalizer object

Syntax

`infostruct = info(obj)`

Description

`infostruct = info(obj)` returns a structure containing characteristic information for the System object.

Examples

Display Decision Feedback Equalizer Latency

Display latency value for a decision feedback equalizer object.

Create a decision feedback equalizer object.

```
dfe = comm.DecisionFeedbackEqualizer;
```

Display latency value for a decision feedback equalizer object by using the `info` object function.

```
info(dfe)
```

```
ans = struct with fields:
    Latency: 2
```

Input Arguments

`obj` — System object to get information from
System object

System object to get information from.

Output Arguments

`infostruct` — Structure containing object information
structure

Structure containing fields with information about the System object.

`Latency` — Equalizer latency
scalar

Equalizer latency, returned as a scalar.

See Also

Objects
`comm.DecisionFeedbackEqualizer` | `comm.LinearEqualizer`

Introduced in R2012a

maxstep

Package: comm

Maximum step size for LMS equalizer convergence

Syntax

```
mumax = maxstep(eq,x)
```

Description

mumax = maxstep(eq,x) predicts a bound on the step size to provide convergence of the mean values of the coefficients of the equalizer defined by the eq System object. The set input signal sequences in x are assumed to have zero mean or nearly so.

Examples

Decision Feedback Equalize BPSK-Modulated Signal

Create a BPSK modulator and an equalizer System object™, specifying a decision feedback LMS equalizer having eight forward taps, five feedback taps, and a step size of 0.03.

```
bpsk = comm.BPSKModulator;
eqdfe_lms = comm.DecisionFeedbackEqualizer('Algorithm','LMS', ...
    'NumForwardTaps',8,'NumFeedbackTaps',5,'StepSize',0.03);
```

Change the reference tap index of the equalizer.

```
eqdfe_lms.ReferenceTap = 4;
```

Build a set of test data. Receive the data by convolving the signal.

```
x = bpsk(randi([0 1],1000,1));
rxsig = conv(x,[1 0.8 0.3]);
```

Use maxstep to find the maximum permitted step size.

```
mxStep = maxstep(eqdfe_lms,rxsig)
```

```
mxStep = 0.1028
```

Equalize the received signal. Use the first 200 symbols as the training sequence.

```
y = eqdfe_lms(rxsig,x(1:200));
```

Input Arguments

eq — Equalizer object
System object

Equalizer object, specified as a `comm.LinearEqualizer` or `comm.DecisionFeedbackFEqualizer` System object.

x — Input signal
column vector

Input signal, specified as a column vector. The input signal vector length must be equal to an integer multiple of the InputSamplesPerSymbol property. For more information, see “Symbol Tap Spacing” on page 3-543.

Data Types: `double`
Complex Number Support: Yes

Output Arguments

`mumax` — Prediction of maximum step size for LMS equalizer convergence
scalar

Prediction of maximum step size for LMS equalizer convergence, returned as a scalar.

See Also

Objects
`comm.DecisionFeedbackEqualizer` | `comm.LinearEqualizer`

Introduced in R2019a

mmseweights

Package: comm

Linear equalizer MMSE tap weights

Syntax

```
weights = mmseweights(eq,chTaps,EbN0)
```

Description

`weights = mmseweights(eq,chTaps,EbN0)` calculated minimum mean squared error (MMSE) solution for the linear equalizer, `eq` System object given the channel delay taps, `chTaps`, and signal to noise ratio, `EbN0`.

Examples

Calculate MMSE Weights for Linear Equalizer

Calculate the minimum mean squared error (MMSE) solution and use the weights for the linear equalizer taps weights.

Initialize simulation variables.

```
M = 4; % QPSK
numSymbols = 10000;
numTrainingSymbols = 1000;
chtaps = [1 0.5*exp(1i*pi/6) 0.1*exp(-1i*pi/8)];
EbN0 = 20;
```

Generate QPSK modulated symbols. Apply delayed multipath channel filtering and AWGN impairments to the symbols.

```
data = randi([0 M-1], numSymbols, 1);
tx = pskmod(data, M, pi/4);
rx = awgn(filter(chtaps,1,tx),25,'measured');
```

Create a linear equalizer System object configured to use CMA algorithm and input the taps weights. Calculate the MMSE weights. Set the initial tap weights to the calculated MMSE weights. Equalize the impaired symbols.

```
eq = comm.LinearEqualizer('Algorithm','CMA','AdaptWeights',false,'InitialWeightsSource','Property
```

```
eq = 
  comm.LinearEqualizer with properties:

                Algorithm: 'CMA'
                  NumTaps: 5
                 StepSize: 0.0100
            Constellation: [1x4 double]
    InputSamplesPerSymbol: 1
```

```
        AdaptWeightsSource: 'Property'
              AdaptWeights: false
      InitialWeightsSource: 'Property'
            InitialWeights: [5x1 double]
        WeightUpdatePeriod: 1
```

```
wgts = mmseweights(eq,chtaps,EbN0)
```

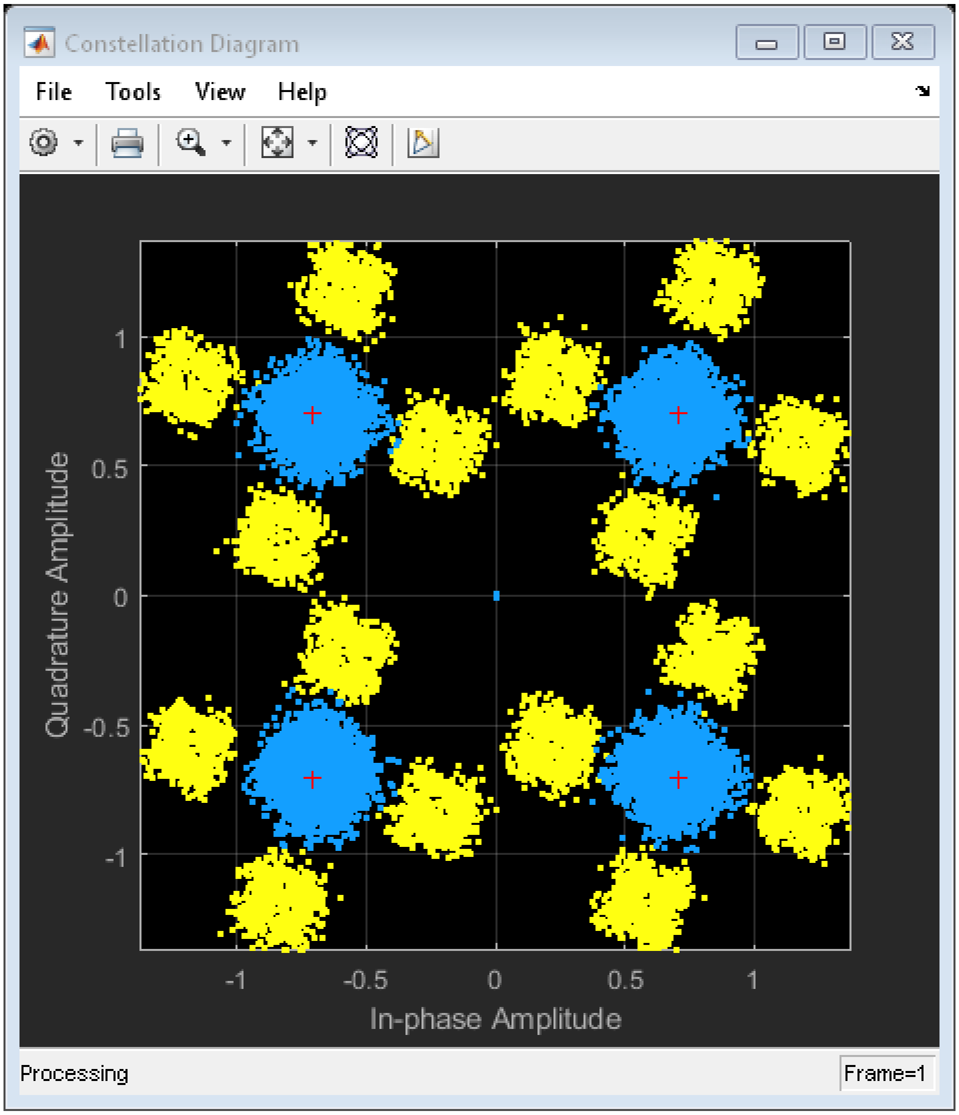
```
wgts = 5×1 complex

   0.0005 - 0.0068i
   0.0103 + 0.0117i
   0.9694 - 0.0019i
  -0.3987 + 0.2186i
   0.0389 - 0.1756i
```

```
eq.InitialWeights = wgts;
```

```
[y,err,weights] = eq(rx);
```

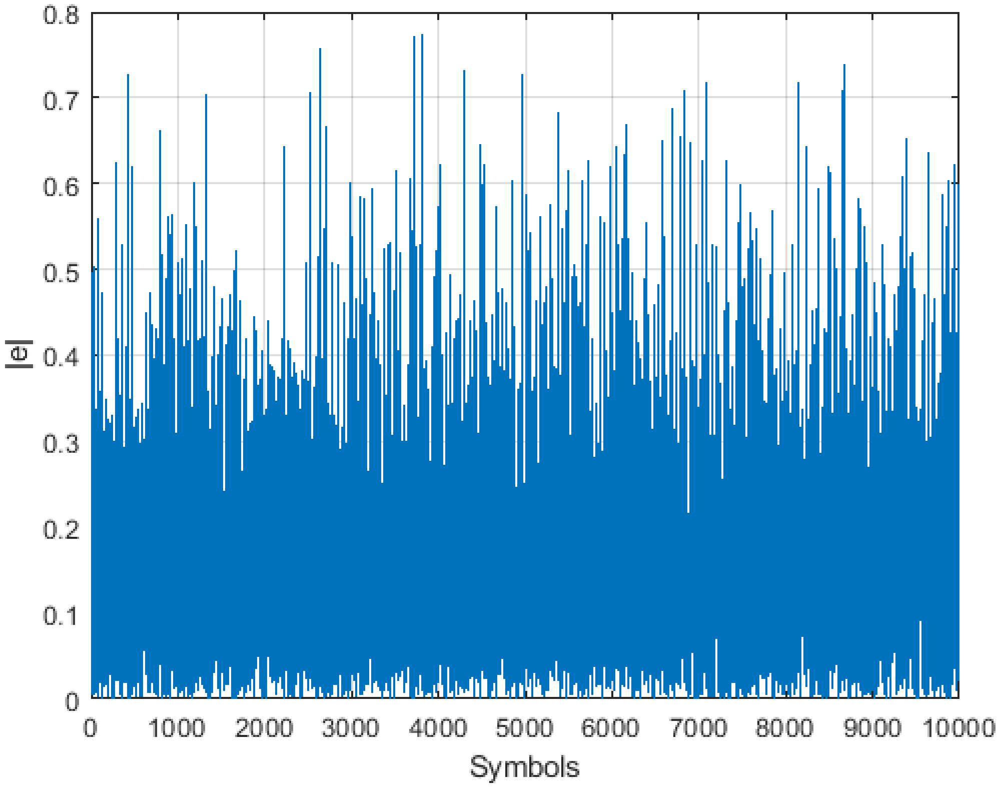
Plot constellation of impaired and equalized symbols.

```
constell = comm.ConstellationDiagram('NumInputPorts',2);
constell(rx,y)
```



Plot the equalizer error signal and compute the error vector magnitude of the equalized symbols.

```
plot(abs(err))
grid on; xlabel('Symbols'); ylabel('|e|')
```

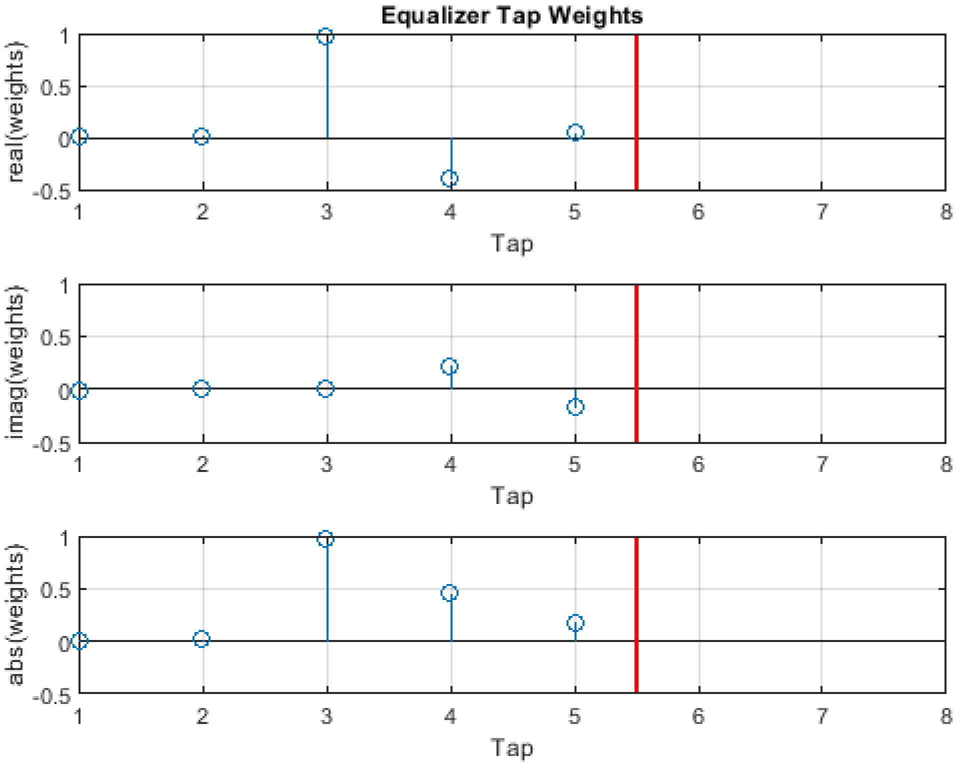


```
errevm = comm.EVM;
evm = errevm(tx,y)
```

```
evm = 140.6177
```

Plot equalizer tap weights.

```
subplot(3,1,1); stem(real(weights)); ylabel('real(weights)'); xlabel('Tap'); grid on; axis([1 8 
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
title('Equalizer Tap Weights')
subplot(3,1,2); stem(imag(weights)); ylabel('imag(weights)'); xlabel('Tap'); grid on; axis([1 8 
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
subplot(3,1,3); stem(abs(weights)); ylabel('abs(weights)'); xlabel('Tap'); grid on; axis([1 8 -0
line([eq.NumTaps+0.5 eq.NumTaps+0.5], [-0.5 1], 'Color', 'r', 'LineWidth', 1)
```



Input Arguments

eq — Equalizer object

System object

Equalizer object, specified as a `comm.LinearEqualizer` System object.

chTaps — Channel delay taps

vector

Channel delay taps, specified as a vector.

Data Types: `double`
Complex Number Support: Yes

EbN0 — Signal to noise ratio

scalar

Signal to noise ratio of the channel, specified as a scalar.

Data Types: `double`

Output Arguments

`weights` — Weights for linear equalizer
vector

Weights for linear equalizer, returned as a vector.

See Also

Objects
`comm.LinearEqualizer`

Introduced in R2019a

visualize

Package: comm

Visualize spectrum mask of phase noise

Syntax

```
visualize(phznoise)
```

Description

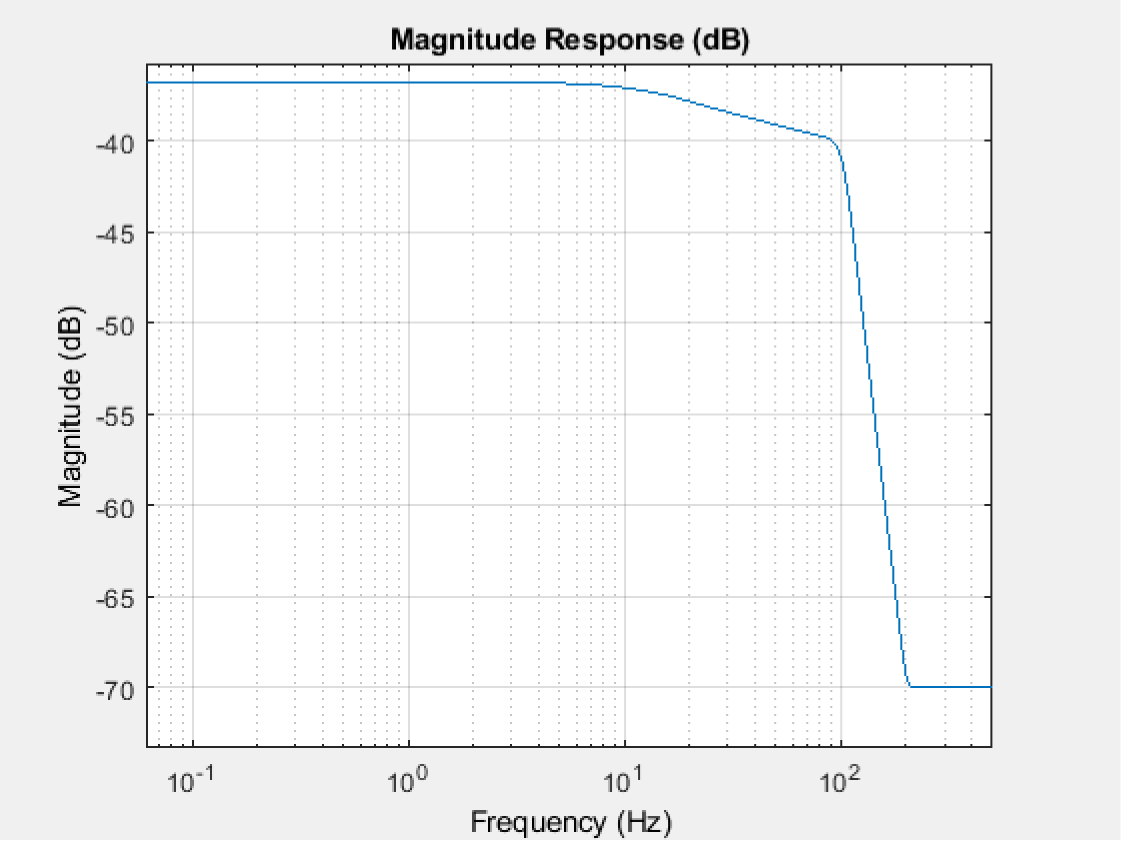
visualize(phznoise) displays the magnitude response of the filter defined by the comm.PhaseNoise System object. The function uses the fvtool function to display the magnitude response.

Examples

Visualize Spectrum Mask of Phase Noise

Create a phase noise object and display the spectral mask.

```
pnoise = comm.PhaseNoise('Level',[-40 -70],'FrequencyOffset',[100 200], ...
    'SampleRate',1000);
visualize(pnoise)
```



Input Arguments

`phznoise` — Phase noise

comm.PhaseNoise System object

Phase noise that defines the spectrum mask that is displayed, specified as a comm.PhaseNoise System object.

See Also

Functions
`fvtool`

Objects
`comm.PhaseNoise`

Introduced in R2012a

addCustomTerrain

Add custom terrain data

Syntax

```
addCustomTerrain(terrainName,files)
addCustomTerrain( ___ ,Name,Value)
```

Description

`addCustomTerrain(terrainName,files)` adds the terrain data specified with a user-defined `terrainName` and `files`. You can use this function to add custom terrain data in Site Viewer and other RF propagation functions. You can access the custom terrain data in the current and future sessions of MATLAB until you call `removeCustomTerrain`.

Note In Antenna Toolbox, `addCustomTerrain` function converts terrain elevation data from orthometric to ellipsoidal for visualization and when performing Euclidean distance or angle calculations between locations for example for free space path loss.

`addCustomTerrain( ___ ,Name,Value)` adds custom terrain data with additional options specified by one or more name-value pairs.

Examples

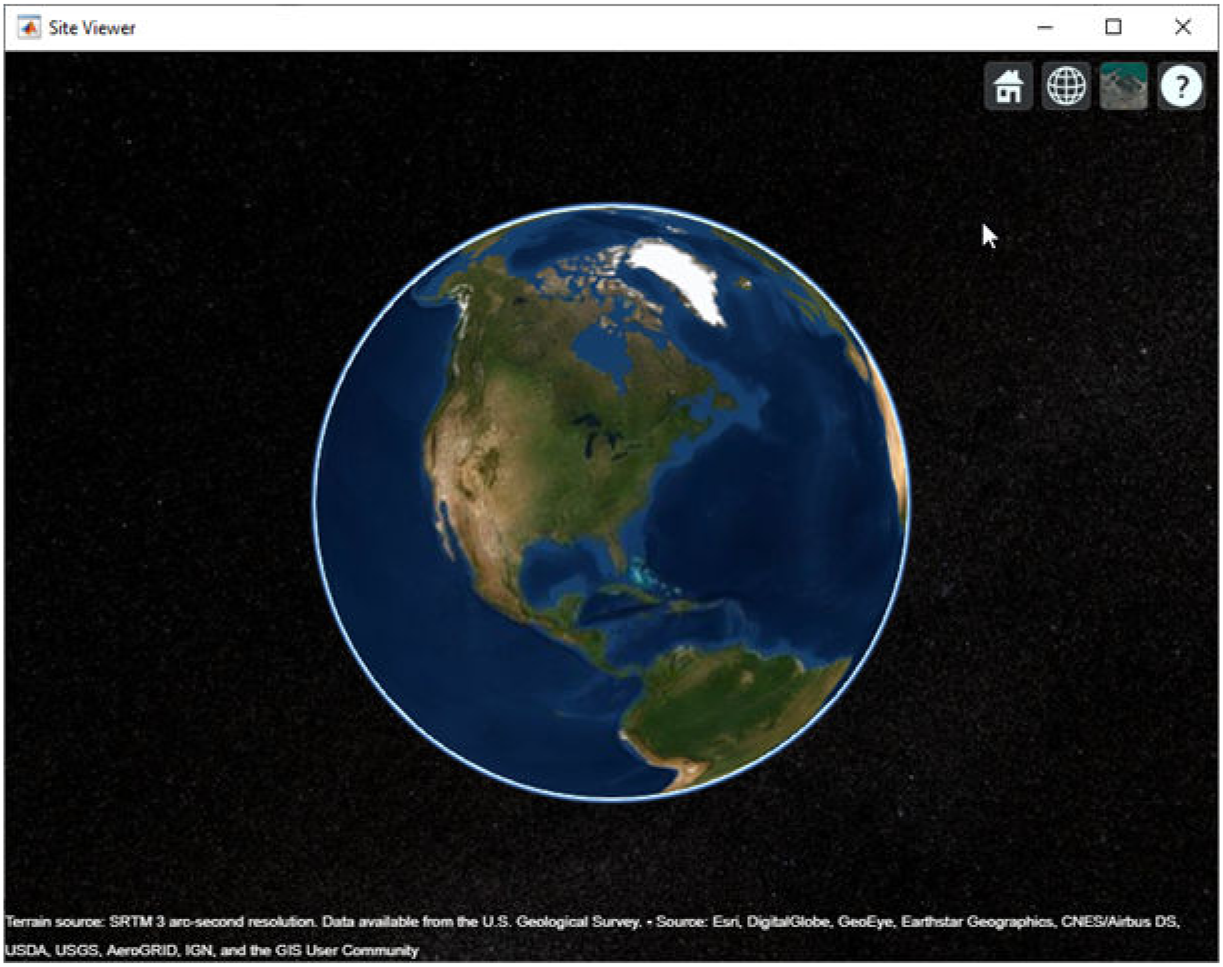
Site Viewer Maps Using Custom Terrain

Add terrain for a region around Boulder, CO. The DTED file was downloaded from the "SRTM Void Filled" data set available from the U.S. Geological Survey.

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available " + ...
   "from the U.S. Geological Survey.";
addCustomTerrain("southboulder",dtedfile,"Attribution",attribution)
```

Use the custom terrain name in Site Viewer.

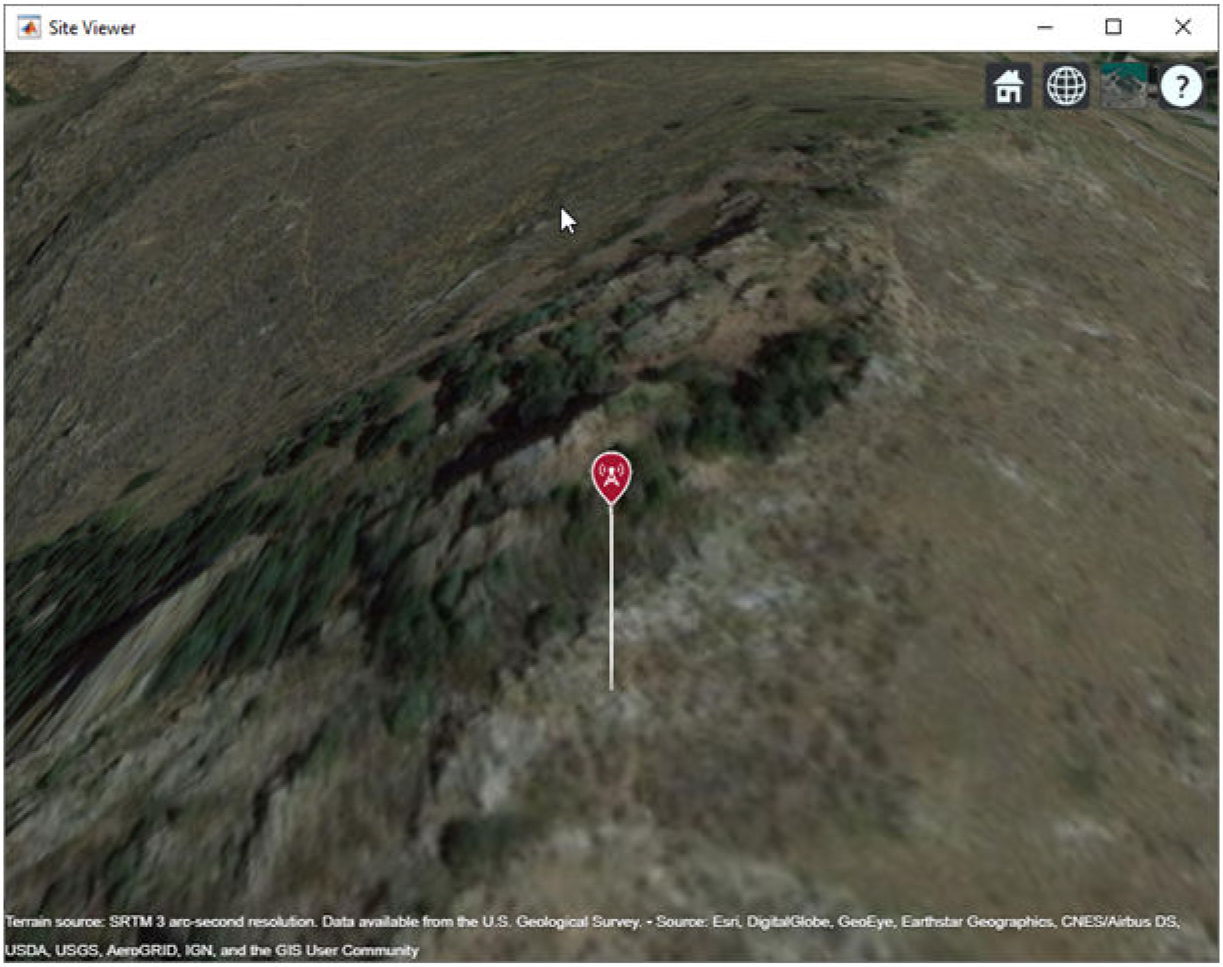
```
viewer = siteviewer("Terrain","southboulder");
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a site with the terrain region.

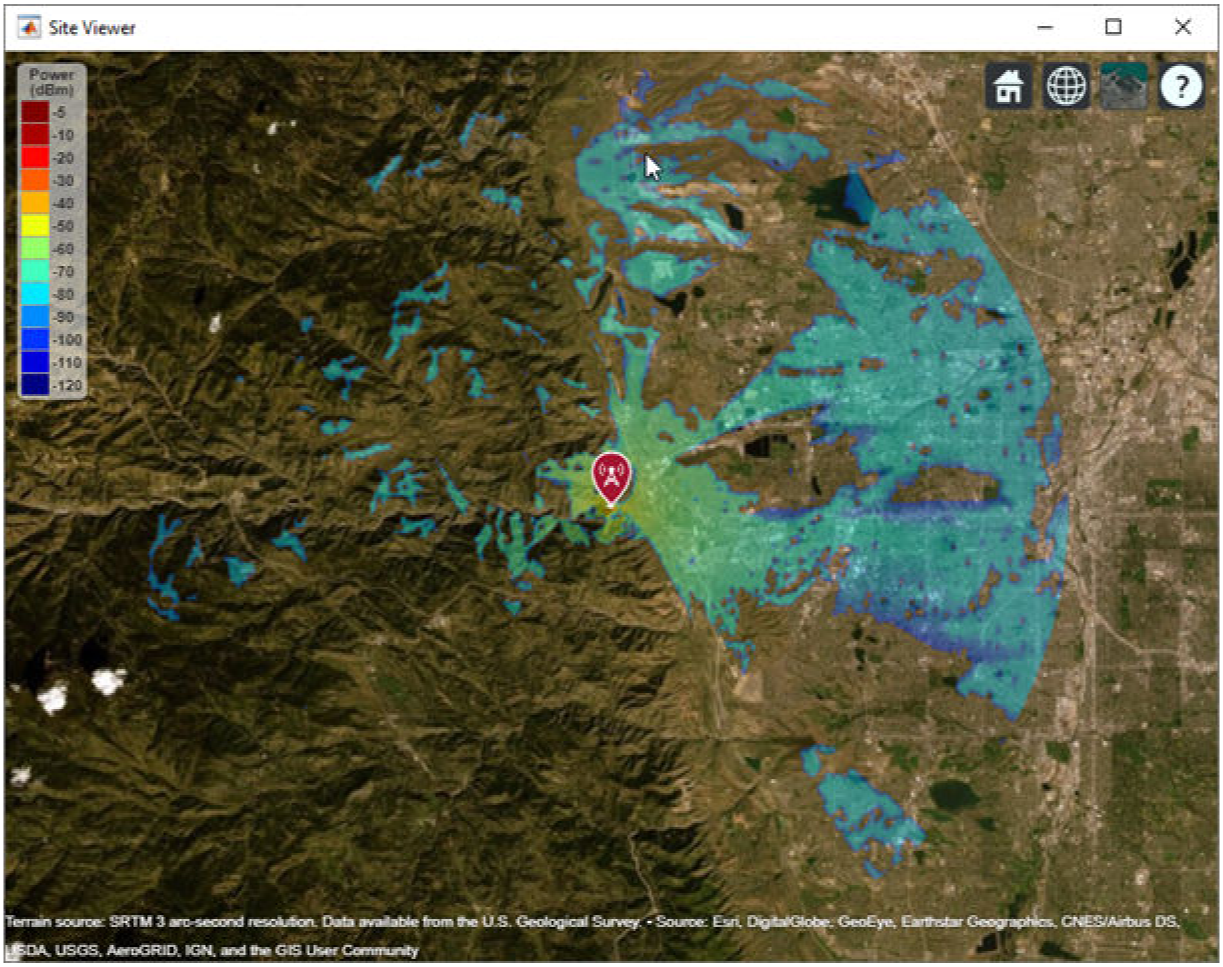
```
mtzion = txsite("Name","Mount Zion", ...
   "Latitude",39.74356, ...
   "Longitude",-105.24193, ...
   "AntennaHeight", 30);
show(mtzion)
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a coverage map of the area within 20 km of the transmitter site.

```
coverage(mtzion, ...
   "MaxRange",20000, ...
   "SignalStrengths",-100:-5)
```


Site Viewer
Power (dBm)
-5
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Remove the custom terrain.

```
close(viewer)
removeCustomTerrain("southboulder")
```

Input Arguments

terrainName — User-defined identifier for terrain data

string scalar | character vector

User-defined identifier for terrain data, specified as a string scalar or a character vector.

Data Types: char | string

files — List of DTED files

string scalar | character vector | cell array of character vectors

List of DTED files, specified as a string scalar, a character vector or a cell array of character vectors.

Note If you specify multiple files, they must combine to define a complete rectangular geographic region. If not, you must set the name-value pair 'FillMissing' to 'true'.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'FillMissing',true`

`Attribution` — Attribution of custom terrain data

character vector | string scalar

Attribution of custom terrain data, specified as a character vector or a string scalar. The attribution is displayed on the Site Viewer map. By default, the value is empty.

Data Types: `char` | `string`

`FillMissing` — Fill data of missing files with value 0

`false` (default) | `true`

Fill data of missing files with value 0, specified as `true` or `false`. Missing file values are required to complete a rectangular geographic region with the input `files`.

Data Types: `logical`

`WriteLocation` — Name of folder to write extracted terrain files to

character vector | string scalar

Name of folder to write extracted terrain files to, specified as a character vector or a string scalar. The folder must exist and have write permissions. By default, `addCustomTerrain` writes extracted terrain files to a temporary folder that it generates using the `tempname` function.

Data Types: `char` | `string`

See Also

`removeCustomTerrain` | `siteviewer`

Introduced in R2019b

angle

Angle between sites

Syntax

```
[az,el] = angle(site1,site2)
[az,el] = angle(site1,site2,path)
[az,el] = angle(___,Name,Value)
```

Description

`[az,el] = angle(site1,site2)` returns the azimuth and elevation angles between site 1 and site

`[az,el] = angle(site1,site2,path)` returns the angles using a specified path type, either Euclidean or great circle path.

`[az,el] = angle(___,Name,Value)` returns the azimuth and elevation angles with additional options specified by name-value pairs.

Examples

Angle Between Sites

Create transmitter and receiver sites.

```
tx = txsite('Name','MathWorks','Latitude',42.3001,'Longitude',-71.3504);
rx = rxsite('Name','Fenway Park','Latitude',42.3467,'Longitude',-71.0972);
```

Get the azimuth and elevation angles between the sites.

```
[az,el] = angle(tx,rx)
```

```
az = 14.0142
```

```
el = -0.2816
```

Get the azimuth angle between sites in degrees clockwise from north.

```
azFromEast = angle(tx,rx); % Unit: degrees counter-clockwise from east
azFromNorth = -azFromEast + 90 % Convert angle to clockwise from north
```

```
azFromNorth = 75.9858
```

Angle Between Sites When Path is Great Circle

Create transmitter and receiver sites.

```
tx = txsite('Name','MathWorks','Latitude',42.3001,'Longitude',-71.3504);
rx = rxsite('Name','Fenway Park','Latitude',42.3467,'Longitude',-71.0972);
```

Get the azimuth and elevation angles between the sites.

```
[az,el] = angle(tx,rx,'greatcircle')
```

```
az = 14.0635
```

```
el = 0
```

Input Arguments

site1,site2 — Transmitter or receiver site

txsite or rxsite object

Transmitter or receiver site, specified as a txsite or rxsite object. You can use array inputs to specify multiple sites.

path — Measurement path type

'euclidean' or 'greatcircle'

Measurement path type, specified as one of the following:

- 'euclidean': Uses the shortest path through space connecting the antenna center positions of the site 1 and site 2.
- 'greatcircle': Uses the shortest path on the surface of the earth connecting the latitude and longitude locations of site 1 and site 2. This path uses a spherical Earth model.

Data Types: char

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Map','siteviewer1'

Map — Map for visualization or surface data

siteviewer object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of 'Map and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
'geographic'	• siteviewer[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are 'none', 'gmted2010', or the name of the custom terrain data added using addCustomTerrain	• current siteviewer or new siteviewer if none are open. • 'gmted2010' if called with an output.

Coordinate System	Valid map values	Default map value
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

a. Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

`az` — Azimuth angle between site 1 and site 2

M-by-*N* arrays

Azimuth angle between site 1 and site 2, returned as *M*-by-*N* arrays in degrees. *M* is the number of sites in sites 2 and *N* is the number of sites in sites 1. The azimuth angle is expressed in degrees counter-clockwise from the east (for geographic sites), or from the global x-axis around the global z-axis (for Cartesian sites), ranging from -180 to 180

`el` — Elevation angle between site 1 and site 2

M-by-*N* arrays

Elevation angle between site 1 and site 2, returned as *M*-by-*N* arrays in degrees. *M* is the number of sites in sites 2 and *N* is the number of sites in sites 1 The elevation angle is expressed in degrees from the horizontal (or X-Y) plane, ranging from -90 to 90.

When the path type specified is `'greatcircle'`, elevation angle is always zero.

See Also

`distance`

Introduced in R2019b

clearMap

Clear map visualizations

Syntax

```
clearMap(viewer)
```

Description

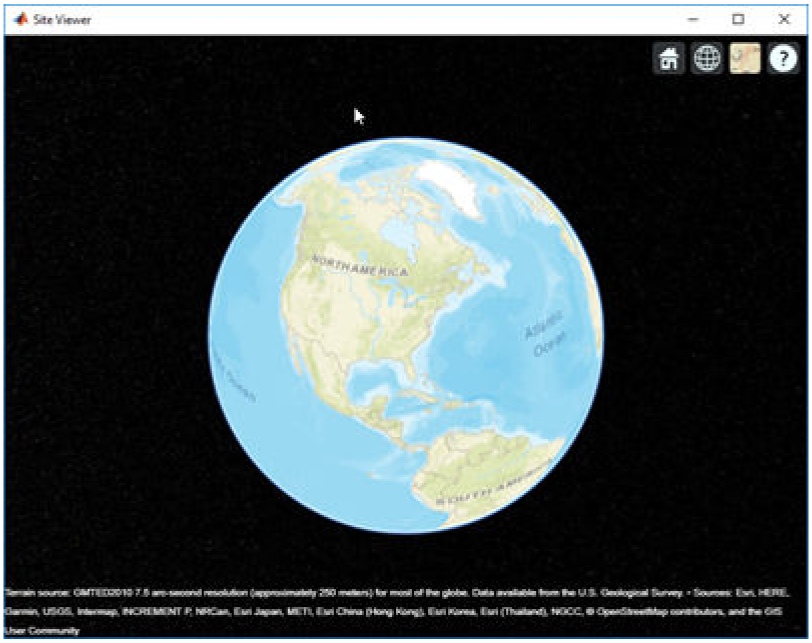
`clearMap(viewer)` removes all visualizations from the map.

Examples

View Transmitter Site On Site Viewer

1 Launch a Site Viewer with streets basemap.

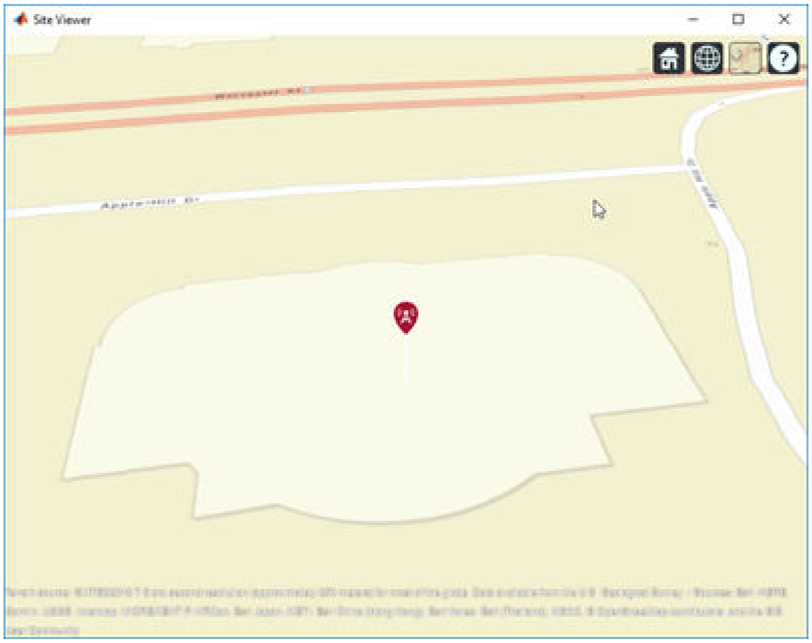
```
viewer = siteviewer("Basemap","streets");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Sources: Esri, HERE, Garmin, USGS, Intermap, INCREMENT P, NRCan, Esri Japan, METI, Esri China (Hong Kong), Esri Korea, Esri (Thailand), NGCC, © OpenStreetMap contributors, and the GIS User Community

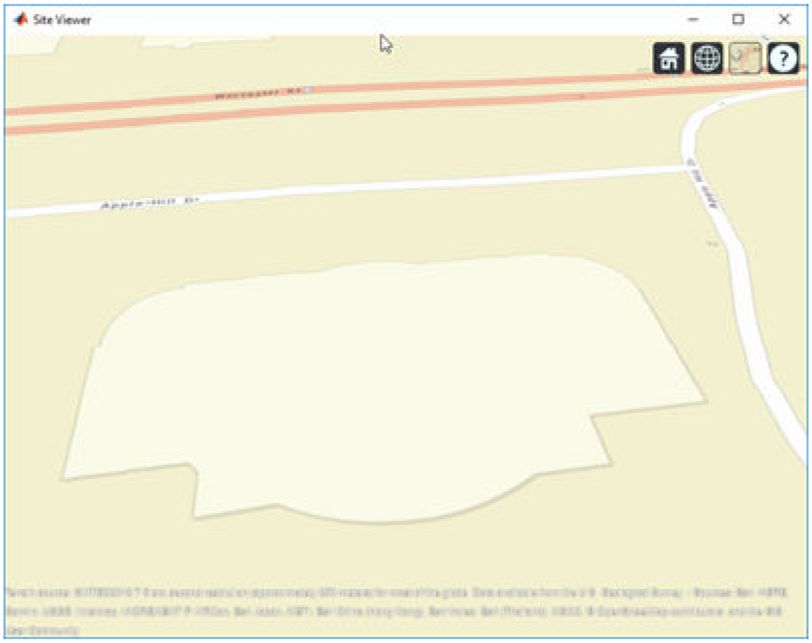
2 View a transmitter site on this map.

```
tx = txsite;
show(tx)
```



3 Clear the map.

```
t = timer('TimerFcn',@(~,~)disp('Fired.'),'StartDelay',3);
start(t)
wait(t)
clearMap(viewer)
```



Input Arguments

viewer — Map viewer for visualizing transmitter or receiver sites

siteviewer object

Map viewer for visualizing transmitter or receiver sites, specified as a `siteviewer` object.[1]

See Also

`close` | `siteviewer`

Introduced in R2019b

[1] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

close

Close map viewer window

Syntax

```
close(viewer)
```

Description

`close(viewer)` closes the map viewer window and deletes the handle

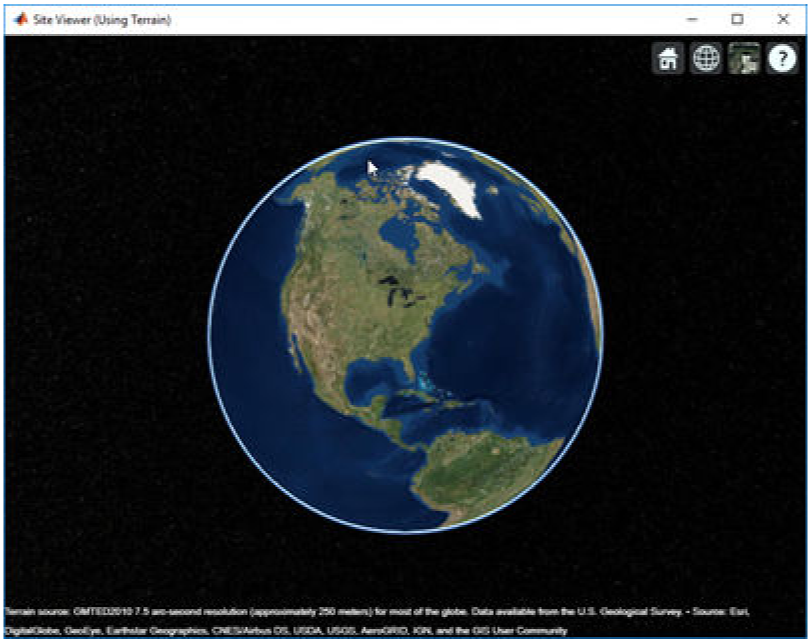
Examples

Compare Coverage Maps

Launch two Site Viewer windows.

One Site Viewer window uses the terrain model.

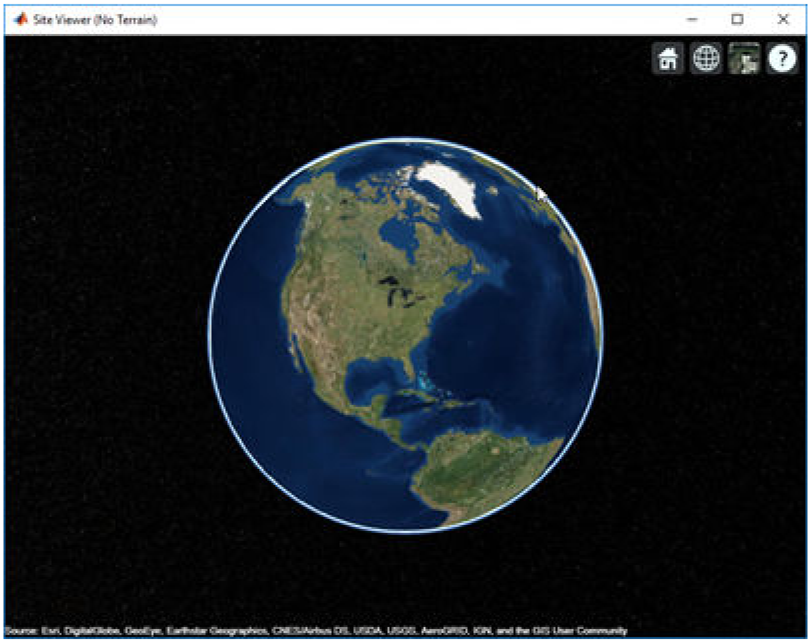
```
viewer1 = siteviewer("Terrain","gmted2010","Name","Site Viewer (Using Terrain)");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

The second Site Viewer window does not use the terrain model.

```
viewer2 = siteviewer("Terrain","none","Name","Site Viewer (No Terrain)");
```

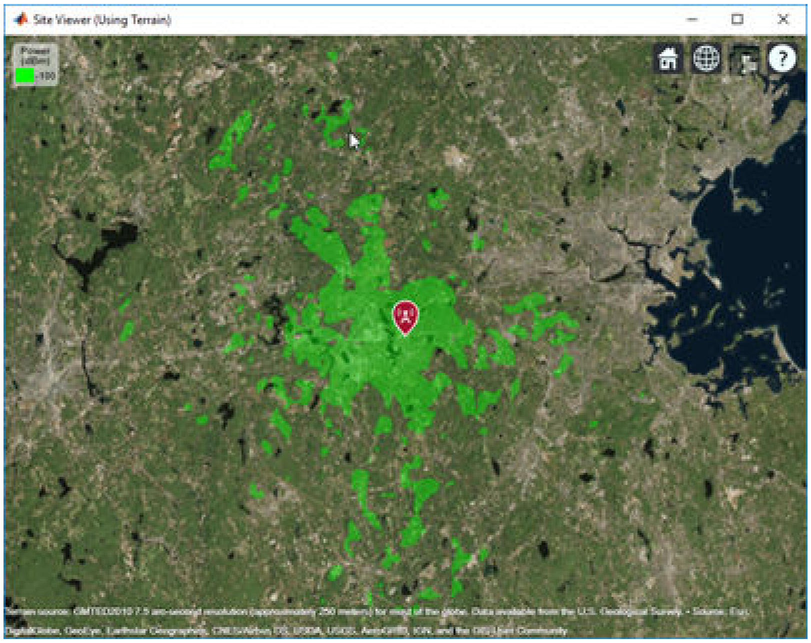

Site Viewer (No Terrain)
Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a transmitter site.

```
tx = txsite;
```

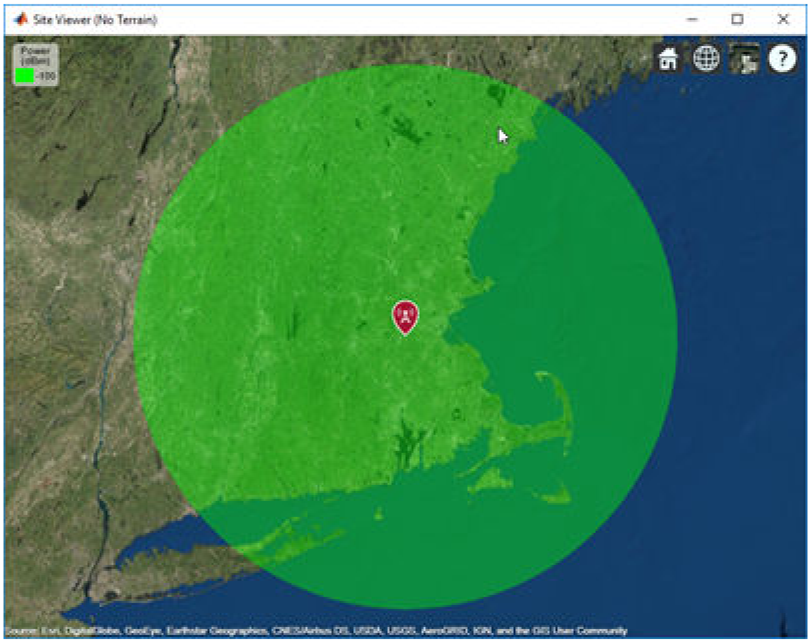
Generate a coverage map on each window. The map with terrain uses the Longley-Rice propagation model by default.

```
coverage(tx,"Map",viewer1)
```


Site Viewer (Using Terrain)
Power (dBm)
🟩 -100
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

The map without terrain uses the free-space model by default.

```
coverage(tx,"Map",viewer2)
```


Site Viewer (No Terrain)
Power (dBm)
-100
Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Close the maps.

```
close(viewer1)
close(viewer2)
```

Input Arguments

`viewer` — Map viewer for visualizing transmitter or receiver sites

`siteviewer` object

Map viewer for visualizing transmitter or receiver sites, specified as a `siteviewer` object.[2]

See Also

`clearMap` | `siteviewer`

Introduced in R2019b

[2] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

contour

Display contour map

Syntax

```
contour(pd)
contour( ___ ,Name,Value)
```

Description

`contour(pd)` creates a filled contour plot on a map. Contours are colored according to data values of corresponding locations.

`contour( ___ ,Name,Value)` creates a filled contour map with additional options specified by name-value pair arguments.

Examples

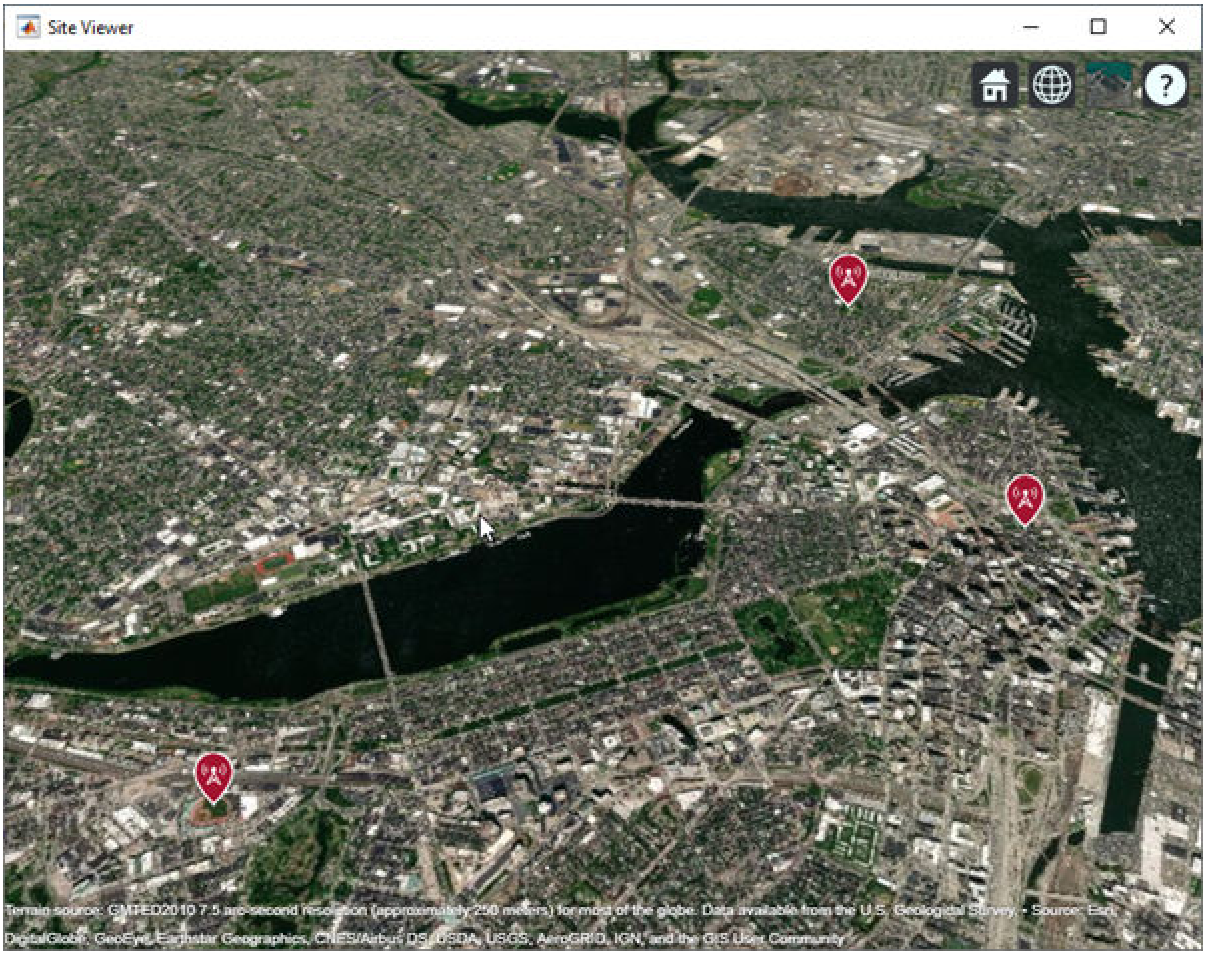
Capacity Map Using SINR Data

Define names and locations of sites around Boston.

```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

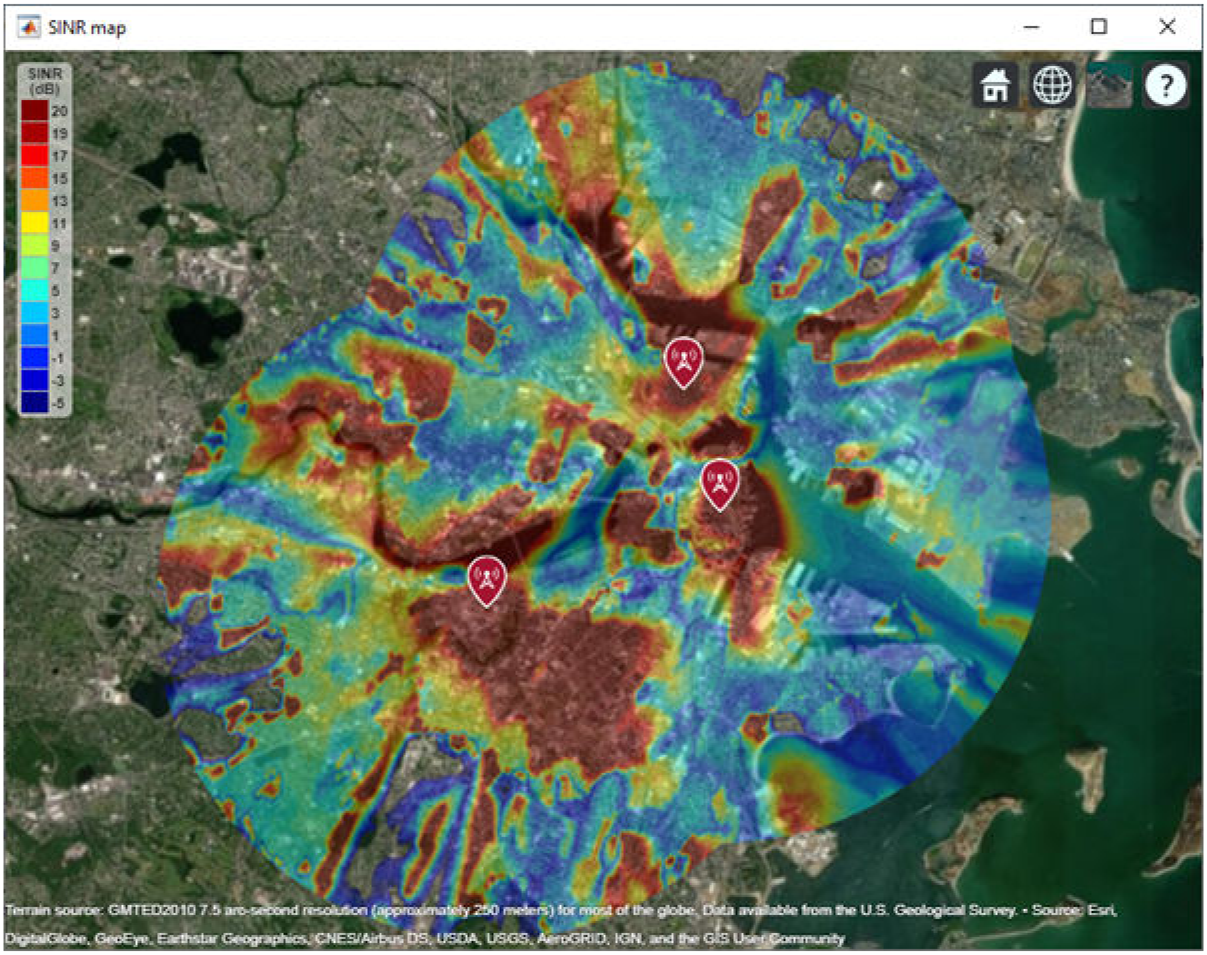
Create an array of transmitter sites.

```
txs = txsite("Name",names,...
       "Latitude",lats,...
       "Longitude",lons, ...
       "TransmitterFrequency",2.5e9);
show(txs)
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a signal-to-interference-plus-noise-ratio (SINR) map, where signal source for each location is selected as the transmitter site with the strongest signal.

```
sv1 = siteviewer("Name","SINR map");
sinr(txs,"MaxRange",5000)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Return SINR propagation data.

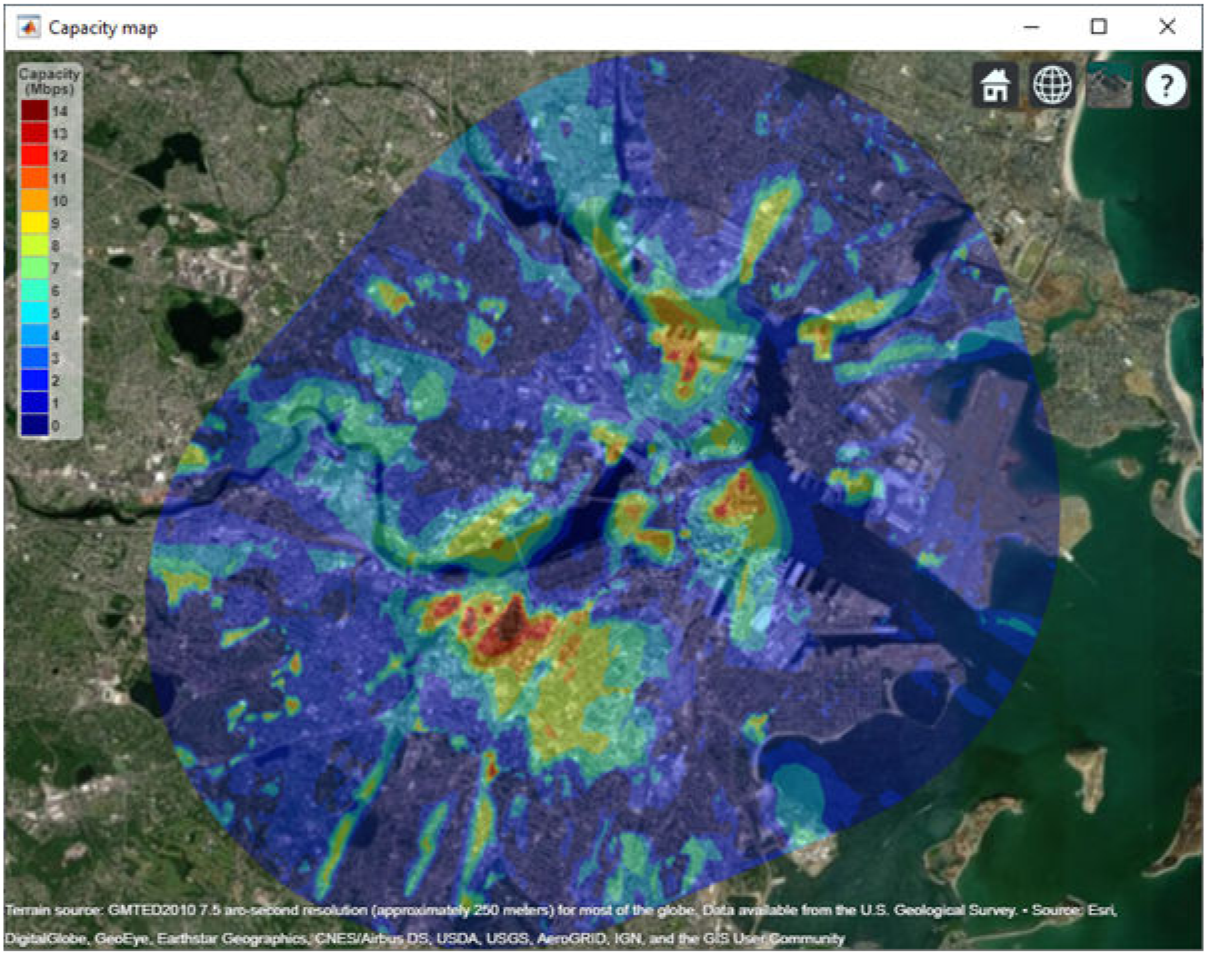
```
pd = sinr(txs,"MaxRange",5000);
[sinrDb,lats,lons] = getDataVariable(pd,"SINR");
```

Compute capacity using the Shannon-Hartley theorem.

```
bw = 1e6; % Bandwidth is 1 MHz
sinrRatio = 10.^(sinrDb./10); % Convert from dB to power ratio
capacity = bw*log2(1+sinrRatio)/1e6; % Unit: Mbps
```

Create new propagation data for the capacity map and display the contour plot.

```
pdCapacity = propagationData(lats,lons,"Capacity",capacity);
sv2 = siteviewer("Name","Capacity map");
legendTitle = "Capacity" + newline + "(Mbps)";
contour(pdCapacity,"LegendTitle",legendTitle);
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

pd — Propagation data

`propagationData` object (default)

Propagation data, specified as a `propagationData` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','power'`

DataVariableName — Data variable to contour map

`DataVariableName` (default) | character vector | string scalar

Data variable to contour map, specified as the comma-separated pair consisting of `'DataVariableName'` and a character vector or a string scalar corresponding to a variable name in the data table used to create the propagation data container object `pd`.

Data Types: char | string

Type — Type of value to plot
'custom' (default) | 'power' | 'efield' | 'sinr' | 'pathloss'

Type of value to plot, specified as the comma-separated pair consisting of 'Type' and one of the values in the 'Type' column:

Type	ColorLimits	LegendTitle
'custom'	[min(Data) max(Data)]	''
'power'	[-120 -5]	'Power (dBm)'
'efield'	[20 135]	'E-field (dBuV/m)'
'sinr'	[-5 20]	'SINR (dB)'
'pathloss'	[45 160]	'Path loss (dB)'

The default value for Levels is a linearly spaced vector bounded by ColorLimits.

Data Types: char | string

Levels — Data value levels to plot
numeric vector

Data value levels to plot, specified as the comma-separated pair consisting of 'Levels' and numeric vector. Each level is displayed as a different colored, filled contour on the map. The colors are selected using Colors if specified, or else Colormap and ColorLimits. Data points with values below the minimum level are not included in the plot.

The default value for Levels is a linearly spaced vector bounded by ColorLimits.

Data Types: double

Colors — Colors of data points
M-by-3 array of RGB | array of strings | cell array of character vectors

Colors of the filled contours, specified as the comma-separated pair consisting of 'Colors' and an M-by-3 array of RGB (red, blue, green) or an array of strings, or a cell array of character vectors. Colors are assigned element-wise to values in Levels for coloring the corresponding points. Colors cannot be used with Colormap and ColorLimits.

Data Types: double | char | string

Colormap — Color map for coloring points
'jet(256)' (default) | predefined colormap name | *M*-by-3 array of RGB triplets

Colormap for the coloring points, specified as the comma-separated pair consisting of 'Colormap' and a predefined colormap name or an *M*-by-3 array of RGB (red, blue, green) triplets that define *M* individual colors. Colormap cannot be used with Colors.

Data Types: double | char | string

ColorLimits — Color limits for color map
two-element vector

Color limits for the colormap, specified as the comma-separated pair consisting of `'ColorLimits'` and a two-element vector of the form [min max]. The color limits indicate the data level values that map to the first and last colors in the colormap. `ColorLimits` cannot be used with `Colors`.

Data Types: `double`

`Transparency` — Transparency of contour map
`0.4` (default) | numeric scalar in the range of [0,1]

Transparency of the contour plot, specified as a numeric scalar in the range [0,1], where `0` is completely transparent and `1` is completely opaque.

Data Types: `double`

`ShowLegend` — Show color legend on map
`true` (default) | `false`

Show color legend on map, specified as the comma-separated pair consisting of `'ShowLegend'` and `true` or `false`.

Data Types: `logical`

`LegendTitle` — Title of color legend
character vector | string scalar

Title of color legend, specified as the comma-separated pair consisting of `'LegendTitle'` and a character vector or a string scalar.

Data Types: `string` | `char`

`Map` — Map for surface data
`siteviewer` object

Map for surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[3] The default value is the current Site Viewer or a new Site Viewer, if none is open.

Data Types: `char` | `string`

See Also

Introduced in R2020a

[3] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

coverage

Display coverage map

Syntax

```
coverage(txs)
coverage(txs,propmodel)
coverage(txs,rxs)
coverage(txs,rxs,propmodel)
coverage( ___ ,Name,Value, ___ )
pd = coverage(txs, ___ )
```

Description

`coverage(txs)` displays the coverage map for the transmitter site. Each colored contour of the map defines an area where the corresponding signal strength is transmitted to the mobile receiver.

Note This function only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`coverage(txs,propmodel)` displays the coverage map based on the specified propagation model. The default propagation model is `'longley-rice'` when terrain is in use and `'freespace'` when terrain is not used.

`coverage(txs,rxs)` displays the coverage map based on the receiver site properties.

`coverage(txs,rxs,propmodel)` displays the coverage map based on the receiver site properties and specified propagation model.

`coverage( ___ ,Name,Value, ___ )` displays the coverage map using additional options specified by the `Name,Value` pairs.

`pd = coverage(txs, ___ )` returns computed coverage data in the propagation data object, `pd`. No plot is displayed and any graphical only name-value pairs are ignored.

Examples

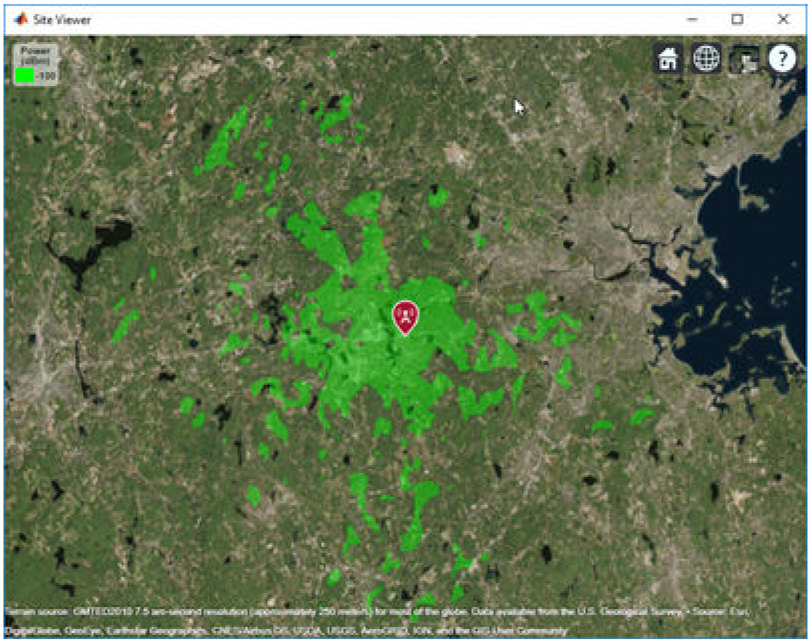
Coverage Map of Transmitter

Create a transmitter site at MathWorks headquarters.

```
tx = txsite('Name','MathWorks', ...
       'Latitude', 42.3001, ...
       'Longitude', -71.3503);
```

Show the coverage map.

```
coverage(tx)
```


Site Viewer
Power (dBm)
🟩 -100
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Coverage Map Using Transmitter and Receiver

Create a transmitter site at MathWorks headquarters.

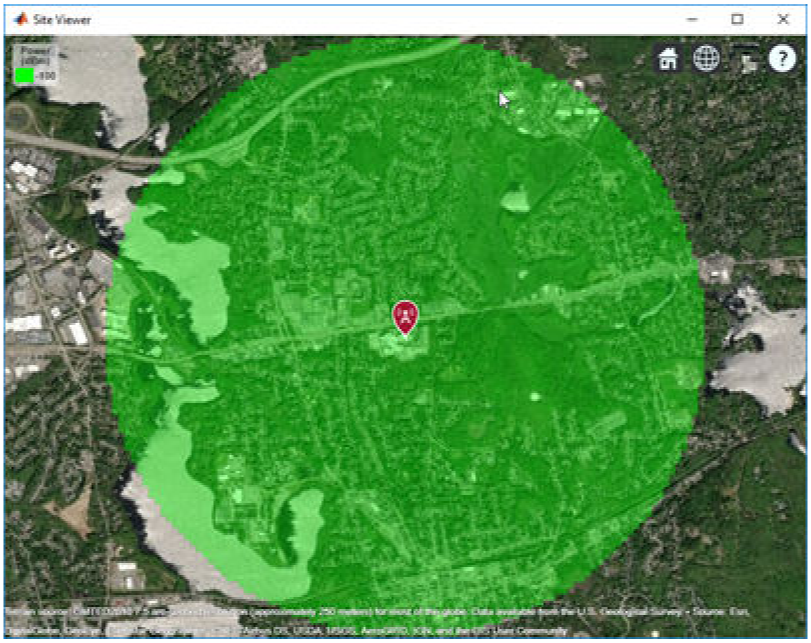
```
tx = txsite('Name','MathWorks', ...
        'Latitude', 42.3001, ...
        'Longitude', -71.3503);
```

Create a receiver site at Fenway Park with an antenna height of 1.2 m and system loss of 10 dB.

```
rx = rxsite('Name','Fenway Park', ...
       'Latitude',42.3467, ...
       'Longitude',-71.0972,'AntennaHeight',1.2,'SystemLoss',10);
```

Calculate the coverage area of the transmitter using a close-in propagation model.

```
coverage(tx,rx,'PropagationModel','closein')
```


Site Viewer
Power (dBm)
-100
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

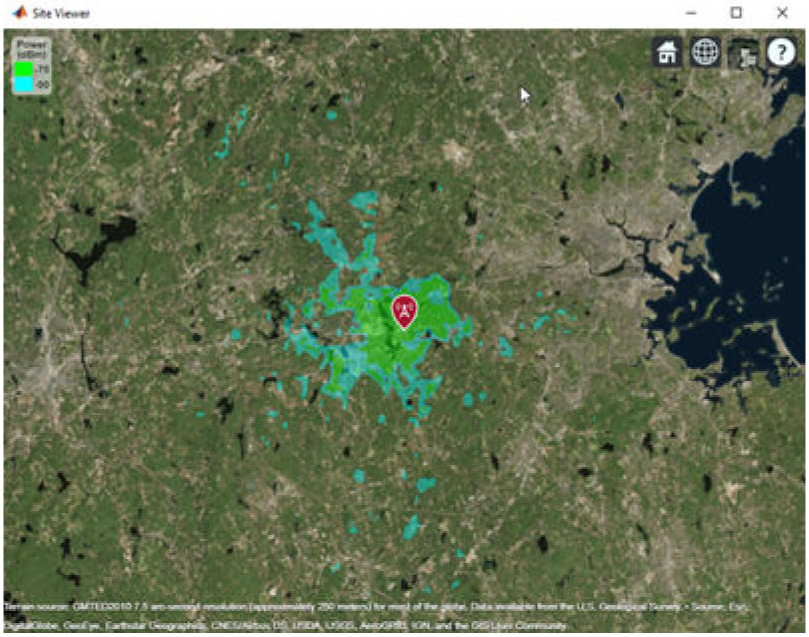
Coverage Map for Strong and Weak Signals

Define strong and weak signal strengths with corresponding colors.

```
strongSignal = -75;
strongSignalColor = "green";
weakSignal = -90;
weakSignalColor = "cyan";
```

Create a transmitter site and display the coverage map.

```
tx = txsite('Name','MathWorks','Latitude', 42.3001,'Longitude', -71.3503);
coverage(tx,'SignalStrengths',[strongSignal,weakSignal], ...
       'Colors', [strongSignalColor,weakSignalColor])
```


Site Viewer
Power (dBm)
-70
-90
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Combined Coverage Map of Multiple Transmitters

Define the names and the locations of sites around Boston.

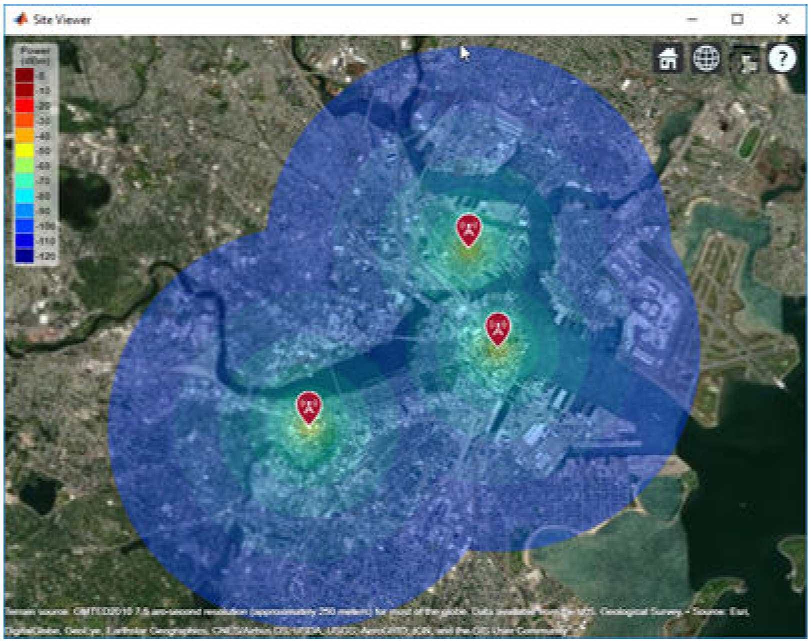
```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

Create the transmitter site array.

```
txs = txsite('Name', names,...
       'Latitude',lats,...
       'Longitude',lons, ...
       'TransmitterFrequency',2.5e9);
```

Display the combined coverage map for multiple signal strengths, using close-in propagation model.

```
coverage(txs,'close-in','SignalStrengths',-100:5:-60)
```

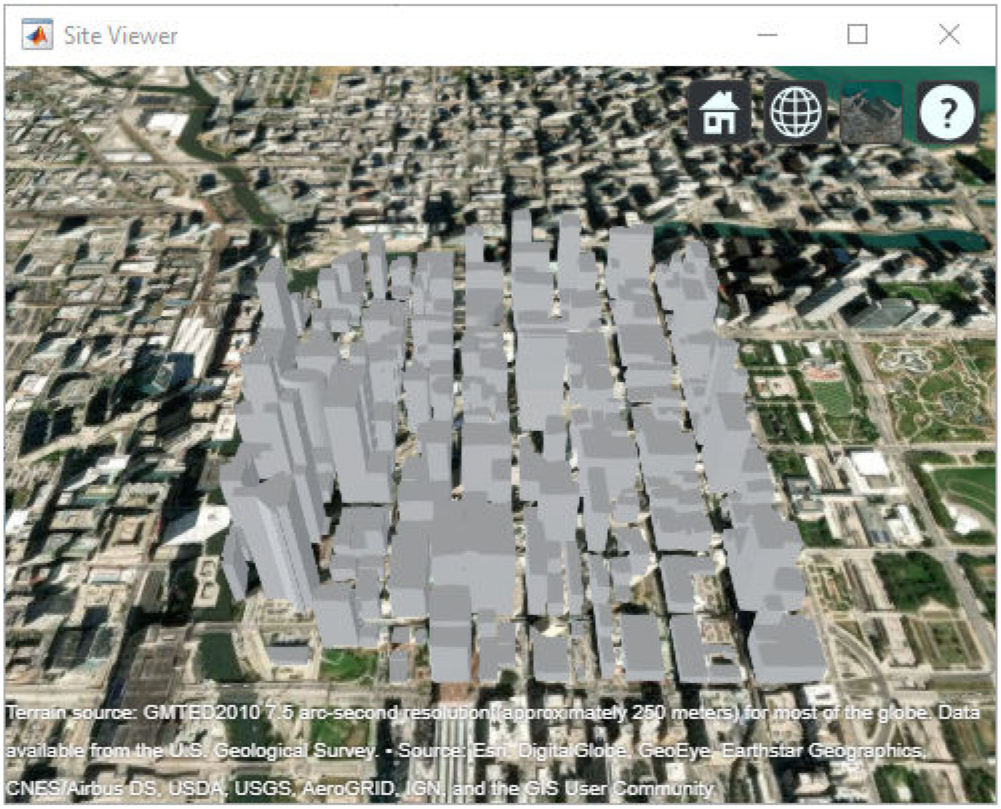


Site Viewer
Power (dBm)
-5
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Coverage Map Using Longley-Rice and Ray Tracing Method

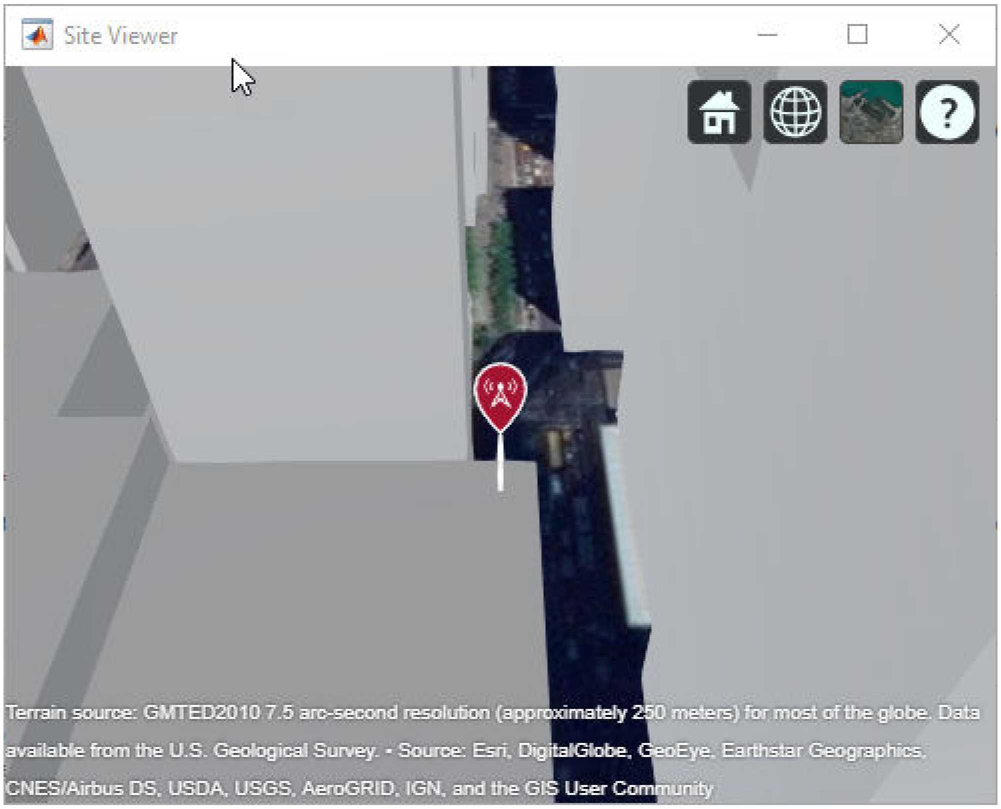
Launch Site Viewer using buildings in Chicago. For more information about the osm file, see [1] on page 4-0 .

```
viewer = siteviewer("Buildings","chicago.osm");
```


Site Viewer
?
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a transmitter site on the building.

```
tx = txsite('Latitude',41.8800, ...
   'Longitude',-87.6295, ...
   'TransmitterFrequency',2.5e9);
show(tx)
```

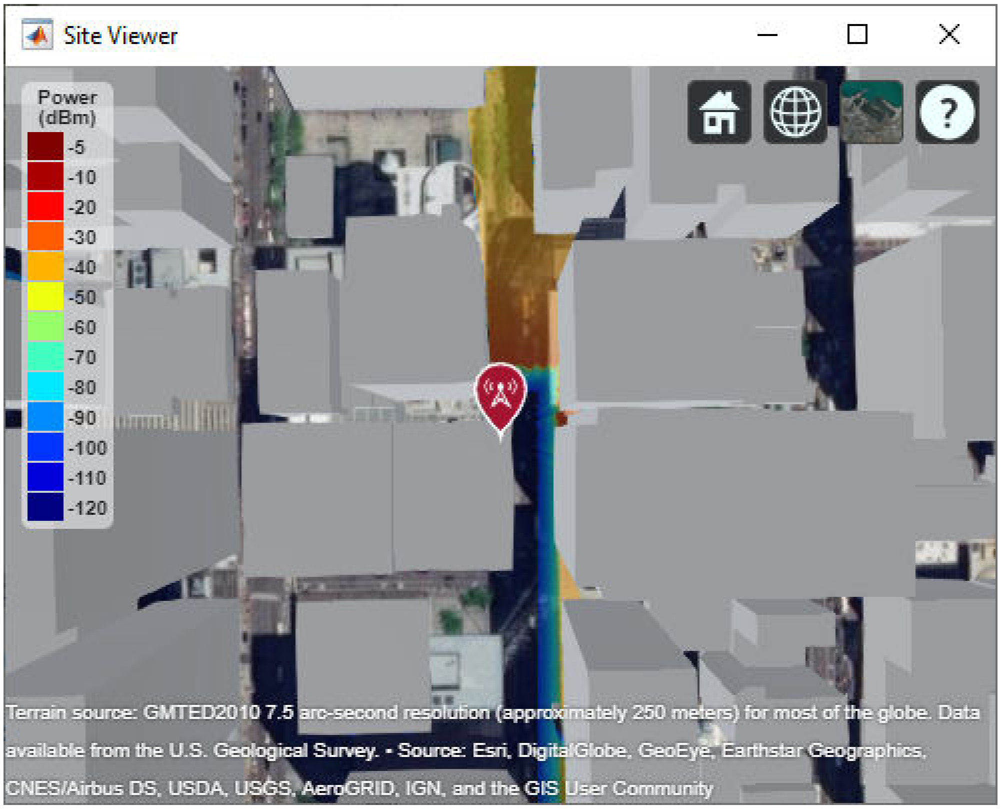


Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Coverage Map Using Longley-Rice Propagation Model

Create a coverage map of the city using the Longley-Rice propagation model.

```
coverage(tx,"SignalStrengths",-100:-5,"MaxRange",250,"Resolution",1)
```

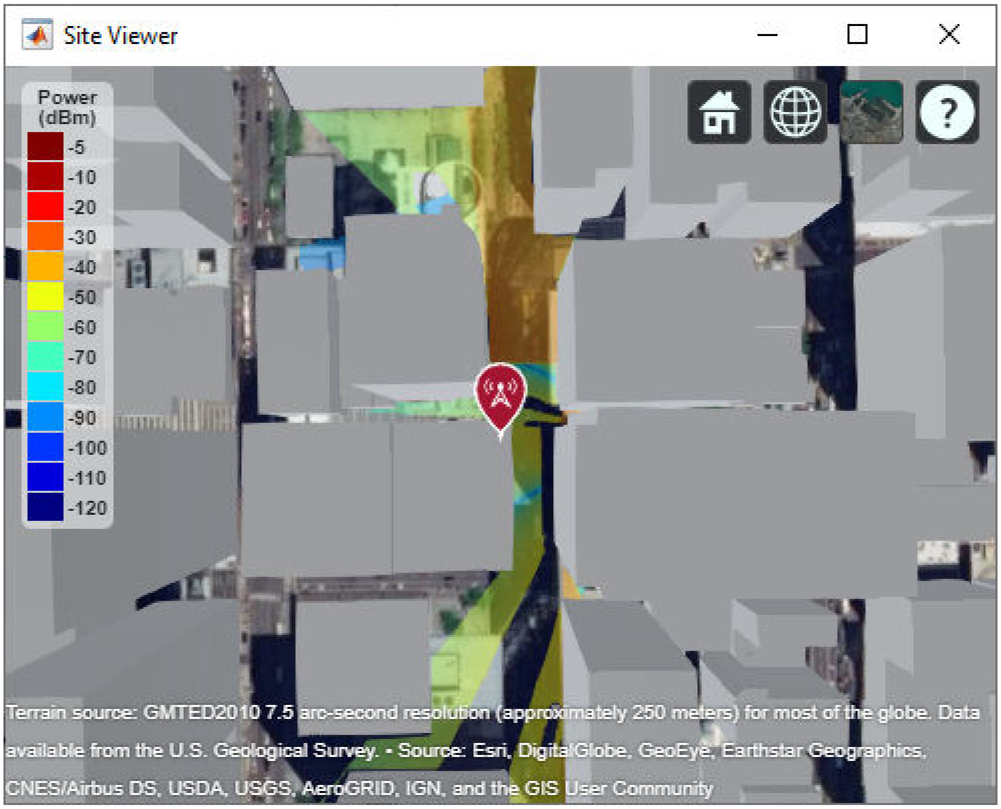

Site Viewer
Power (dBm)
-5
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Longley-Rice models over-the-rooftops propagation along vertical slices and obstructions tend to dominate the coverage region.

Coverage Map Using Ray Tracing Propagation Model

Create a coverage map of the city using the ray tracing image method propagation model.

```
coverage(tx,"raytracing-image-method","SignalStrengths",-100:-5,"MaxRange",250,"Resolution",2)
```


Site Viewer
Power (dBm)
-5
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

This coverage map shows new regions that are in service due to reflected propagation paths.

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

txs — Transmitter sites

txsite object | array of txsite objects

Transmitter site, specified as a `txsite` object. Use array inputs to specify multiple sites.

This function only supports plotting antenna sites when `CoordinateSystem` property is set to `'geographic'`.

rxs — Receiver sites

rxsite object | array of rxsite objects

Receiver site, specified as a `rxsite` object. Use array inputs to specify multiple sites.

This function only supports plotting antenna sites when `CoordinateSystem` property is set to `'geographic'`.

`propmodel` — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can also use the name-value pair `'PropagationModel'` to specify this parameter. You can also use the `propagationModel` function to define this input. The default propagation model is `'longley-rice'` when terrain is enabled and `'freespace'` when terrain is disabled.

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','power'`

Type — Type of signal strength to compute

`'power'` (default) | `'efield'`

Type of signal strength to compute, specified as the comma-separated pair consisting of `'Type'` and `'power'` or `'efield'`.

When type is `'power'`, `SignalStrengths` is expressed in power units (dBm) of the signal at the mobile receiver input. When type is `'efield'`, `SignalStrengths` is expressed in electric field strength units (dBμV/m) of signal wave incident on the antenna.

Data Types: `char`

`SignalStrengths` — Signal strengths to display on coverage map

numeric vector

Signal strengths to display on coverage map, specified as the comma-separated pair consisting of `'SignalStrengths'` and a numeric vector.

Each strength uses different colored filled contour on the map. The default value is `-100` dBm if the `'Type'` name-value pair is `'power'` and `40` dBμV/m if `'Type'` is `'efield'`.

Data Types: `double`

`PropagationModel` — Propagation model to use for path loss calculations

`'longley-rice'` (default) | `'freespace'` | `'close-in'` | `'rain'` | `'gas'` | `'fog'` | `'raytracing-image-method'` | propagation model object

Propagation model to use for the path loss calculations, specified as the comma-separated pair consisting of `'PropagationModel'` and one of the following:

- `'freespace'` - Free space propagation model
- `'rain'` - Rain propagation model
- `'gas'` - Gas propagation model
- `'fog'` - Fog propagation model
- `'close-in'` - Close-in propagation model
- `'longley-rice'` - Longley-Rice propagation model

- 'tirem' - Tirem propagation model
- 'raytracing-image-method' - -Raytracing propagation model using method of images.

The default propagation model is 'longley-rice' when terrain is enabled and 'freespace' when terrain is disabled. If 'raytracing-image-method' is specified, the value of 'MaxNumReflections' property must be lesser than 1.

Data Types: char

MaxRange — Maximum range of coverage map from each transmitter site

numeric scalar

Maximum range of coverage map from each transmitter site, specified as a positive numeric scalar in meters representing great circle distance. MaxRange defines the region of interest on the map to plot. The default value is automatically computed based on the propagation model type as shown:

Propagation Model	MaxRange
Basic or urban	Range of minimum value in SignalStrengths.
Terrain	30 km or distance to the furthest building.
Multipath	500 m

Data Types: double

Resolution — Resolution of coverage map

'auto' (default) | numeric scalar

Resolution of coverage map, specified as the comma-separated pair consisting of 'Resolution' and a numeric scalar in meters.

The resolution of 'auto' computes the maximum value scaled to 'MaxRange'. Decreasing the resolution increases the quality of the coverage map and the time required to create it.

Data Types: char | double

ReceiverGain — Mobile receiver gain

2.1 (default) | numeric scalar

Mobile receiver gain, specified as the comma-separated pair consisting of 'ReceiverGain' and a numeric scalar in dB. The receiver gain value includes the mobile receiver antenna gain and system loss.

The receiver gain computes received signal strength when the 'Type' is 'power'.

If receiver site argument rx is passed to coverage, the default value is the maximum gain of the receiver antenna with the system loss subtracted. Otherwise the default value is 2.1.

Data Types: char | double

ReceiverAntennaHeight — Mobile receiver antenna height above ground elevation

1 (default) | numeric scalar

Mobile receiver antenna height above ground elevation, specified as the comma-separated pair consisting of 'ReceiverAntennaHeight' and a numeric scalar in meters.

If receiver site argument `rx` is passed to coverage, the default value is the `AntennaHeight` of the receiver. Otherwise the default value is 1.

Data Types: `double`

`Colors` — Colors of filled contours on coverage map
M-by-3 array of RGB triplets | array of strings | cell array of character vectors

Filled contours color of coverage map, specified as the comma-separated pair consisting of `'Colors'` and an *M*-by-3 array of RGB triplets, an array of strings, or a cell array of character vectors.

Colors are assigned element-wise to `'SignalStrengths'` values for coloring the corresponding filled contours.

`'Colors'` cannot be used with `'ColorLimits'` or `'ColorMap'`.

For more information, see `ColorSpec (Color Specification)`.

Data Types: `char` | `string` | `double`

`ColorLimits` — Color limits for colormap
two-element vector

Color limits for colormap, specified as the comma-separated pair consisting of `'ColorLimits'` and a two-element vector of type `[min max]`.

The color limits indicate the signal level values that map to the first and last colors on the colormap.

The default value is `[-120 -5]` if the `'Type'` name-value pair is `'power'` and `[20 135]` if `'Type'` is `'efields'`.

`'ColorLimits'` cannot be used with `'Color'`.

Data Types: `double`

`ColorMap` — Colormap filled contours for coverage map
`'jet'` (default) | predefined color map | *M*-by-3 array of RGB triplets

Colormap filled contours on coverage map, specified as the comma-separated pair consisting of `'ColorMap'` and a predefined colormap or *M*-by-3 array of RGB triplets, where *M* defines individual colors.

`'ColorMap'` cannot be used with `'Colors'`.

Data Types: `char` | `double`

`ShowLegend` — Show signal strength color legend on map
`true` (default) | `false`

Show signal strength color legend on map, specified as the comma-separated pair consisting of `'ShowLegend'` and `true` or `false`.

Data Types: `logical`

`Transparency` — Transparency of coverage map
`0.4` (default) | numeric scalar

Transparency of coverage map, specified as the comma-separated pair consisting of `'Transparency'` and a numeric scalar in the range `0` to `1`. `0` is transparent and `1` is opaque.

Data Types: `double`

`Map` — Map for visualization of surface data
`siteviewer` object

Map for visualization of surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[4]

Data Types: `char` | `string`

Output Arguments

pd — Coverage data
`propagationData` object

Coverage data, returned as a `propagationData` object consisting of *Latitude* and *Longitude*, and a signal strength variable corresponding to the plot type. Name of the `propagationData` is `"Coverage Data"`.

See Also

`link` | `propagationModel` | `sigstrength` | `sinr`

Topics
`ColorSpec (Color Specification)`

Introduced in R2019b

[4] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

distance

Distance between sites

Syntax

```
d = distance(site1,site2)
d = distance(site1,site2,path)
d = distance( ___ ,Name,Value)
```

Description

`d = distance(site1,site2)` returns the distance in meters between site1 and site2.

`d = distance(site1,site2,path)` returns the distance using a specified path type, either Euclidean or great circle path.

`d = distance( ___ ,Name,Value)` returns the distance with additional options specified by name-value pairs.

Examples

Distance Between Transmitter and Receiver Site

Create transmitter and receiver sites.

```
tx = txsite('Name','MathWorks','Latitude',42.3001,'Longitude',-71.3504);
rx = rxsite('Name','Fenway Park','Latitude',42.3467,'Longitude',-71.0972);
```

Get the Euclidean distance in km between the sites.

```
dme = distance(tx,rx)
```

```
dme = 2.1504e+04
```

```
dkm = dme / 1000
```

```
dkm = 21.5037
```

Get the great circle distance between the two sites.

```
dmg = distance(tx,rx,'greatcircle')
```

```
dmg = 2.1451e+04
```

Input Arguments

site1,site2 — Transmitter or receiver site
txsite or rxsite object

Transmitter or receiver site, specified as a `txsite` or `rxsite`. You can use array inputs to specify multiple sites.

`path` — Measurement path type
`'euclidean'` | `'greatcircle'`

Measurement path type, specified as one of the following:

- `'euclidean'`: Uses the shortest path through space that connects the antenna center positions of the site 1 and site 2.
- `'greatcircle'`: Uses the shortest path on the surface of the earth that connects the latitude and longitude locations of site 1 and site 2. This path uses a spherical Earth model.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Map','siteviewer1'`

`Map` — Map for visualization or surface data
`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
`'geographic'`	• `siteviewer`[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are `'none'`, `'gmted2010'`, or the name of the custom terrain data added using `addCustomTerrain`	• current siteviewer or new siteviewer if none are open. • `'gmted2010'` if called with an output.
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

d — Distance between sites
M-by-*N* numeric array

Distance between sites, returned as *M*-by-*N* arrays in degrees. *M* is the number of sites in site 2 and *N* is the number of sites in site 1.

See Also

`angle`

Introduced in R2019b

elevation

Elevation of site

Syntax

```
z = elevation(site)
z = elevation( ___ ,Name,Value)
```

Description

`z = elevation(site)` returns the ground or building surface elevation of antenna site in meters. Elevation is measured relative to mean sea level using earth gravitational model, EGM-96. If the site coincides with a building, elevation is measured at the top of the building. Otherwise, elevation is measured at the ground.

Note This function only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`z = elevation( ___ ,Name,Value)` returns the ground elevation of the antenna in meters with additional options specified by name-value pairs.

Examples

Elevation at Mount Washington

Compute and display the elevation at Mount Washington in meters.

```
mtwash = txsite('Name','Mt Washington','Latitude',44.2706, ...
       'Longitude',-71.3033);
z = elevation(mtwash)
```

```
z = 1.8675e+03
```

Input Arguments

`site` — Transmitter or receiver site

`txsite` or `rxsite` object | array of `txsite` or `rxsite` objects

Transmitter or receiver site, specified as a `txsite` or `rxsite` object or an array of `txsite` or `rxsite` objects.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: 'Map','siteviewer1'

Map — Map for visualization or surface data
siteviewer object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of 'Map and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
'geographic'	• siteviewer[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are 'none', 'gmted2010', or the name of the custom terrain data added using addCustomTerrain	• current siteviewer or new siteviewer if none are open. • 'gmted2010' if called with an output.
'cartesian'	'none', triangulation object or name of an STL file.	'none'

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: char | string

Output Arguments

z — Ground or building surface elevation of antenna site
M-by-1 matrix

Ground or building surface elevation of the antenna site, returned as an *M*-by-1 matrix with each element unit in meters. *M* is the number of sites in site.

See Also

angle | distance | rxsite | txsite

Introduced in R2019b

getDataVariable

Get data variable values of data points in propagation data object

Syntax

```
datavariable = getDataVariable(pd)
[datavariable,lat,lon] = getDataVariable(pd)
[ ___ ] = getDataVariable(pd,varname)
```

Description

`datavariable = getDataVariable(pd)` returns the values of the data points in the propagation data object. The data is processed such that the missing values are removed and duplicate location data are replaced with mean values.

`[datavariable,lat,lon] = getDataVariable(pd)` returns the location coordinates of the data points in the propagation data object.

`[ ___ ] = getDataVariable(pd,varname)` returns the values of the data points corresponding to the `varname` variable.

Examples

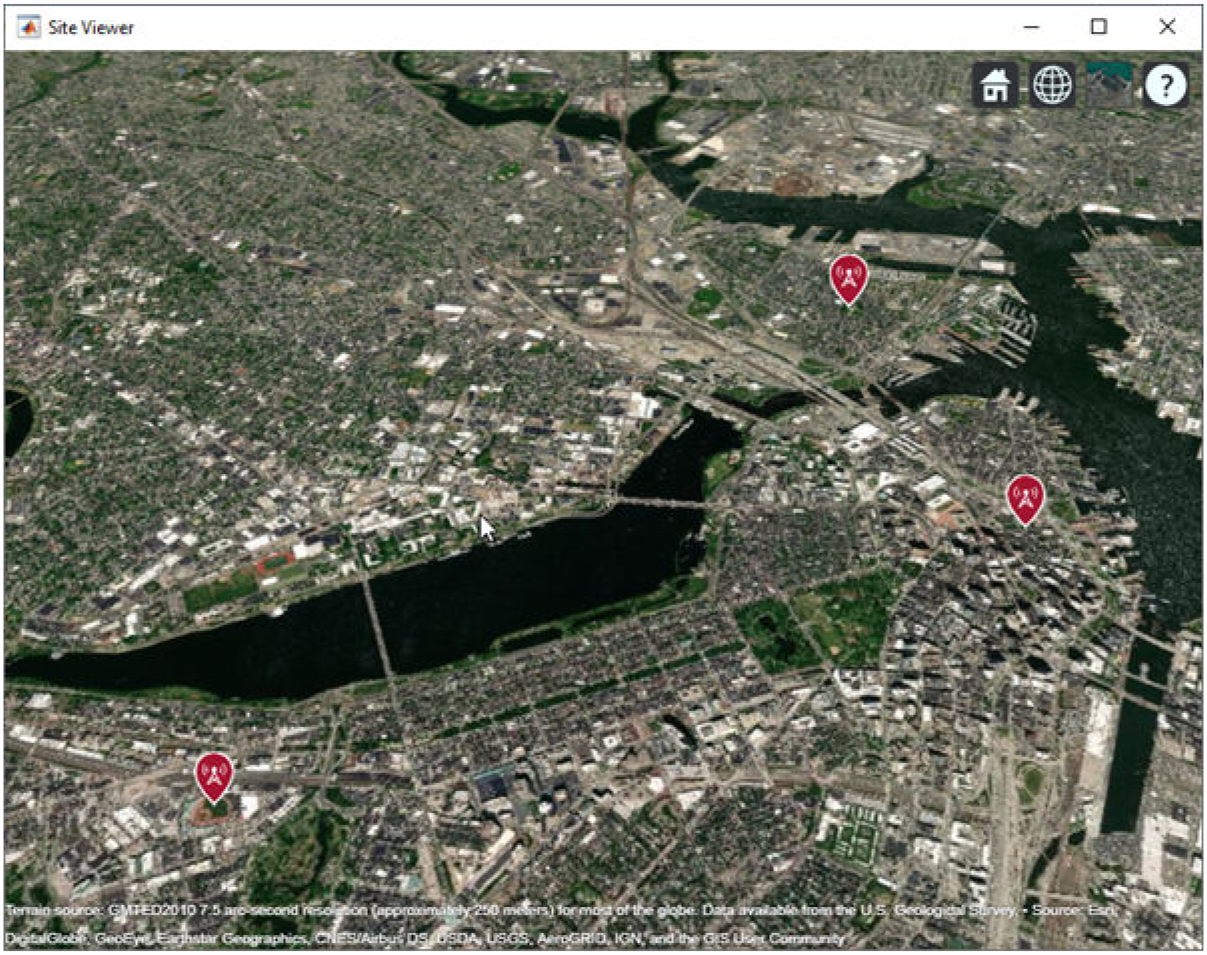
Capacity Map Using SINR Data

Define names and locations of sites around Boston.

```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

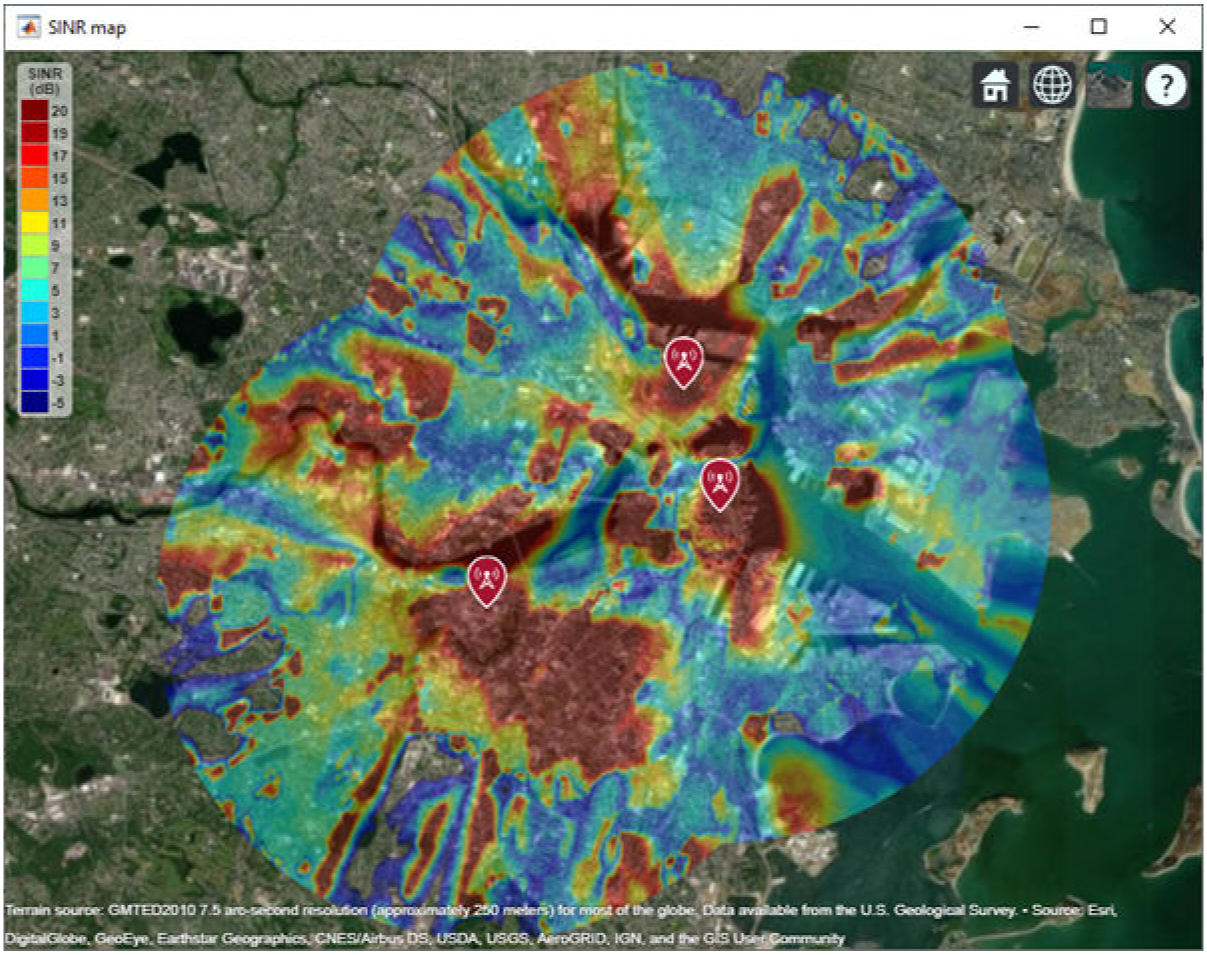
Create an array of transmitter sites.

```
txs = txsite("Name",names,...
       "Latitude",lats,...
       "Longitude",lons, ...
       "TransmitterFrequency",2.5e9);
show(txs)
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a signal-to-interference-plus-noise-ratio (SINR) map, where signal source for each location is selected as the transmitter site with the strongest signal.

```
sv1 = siteviewer("Name","SINR map");
sinr(txs,"MaxRange",5000)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Return SINR propagation data.

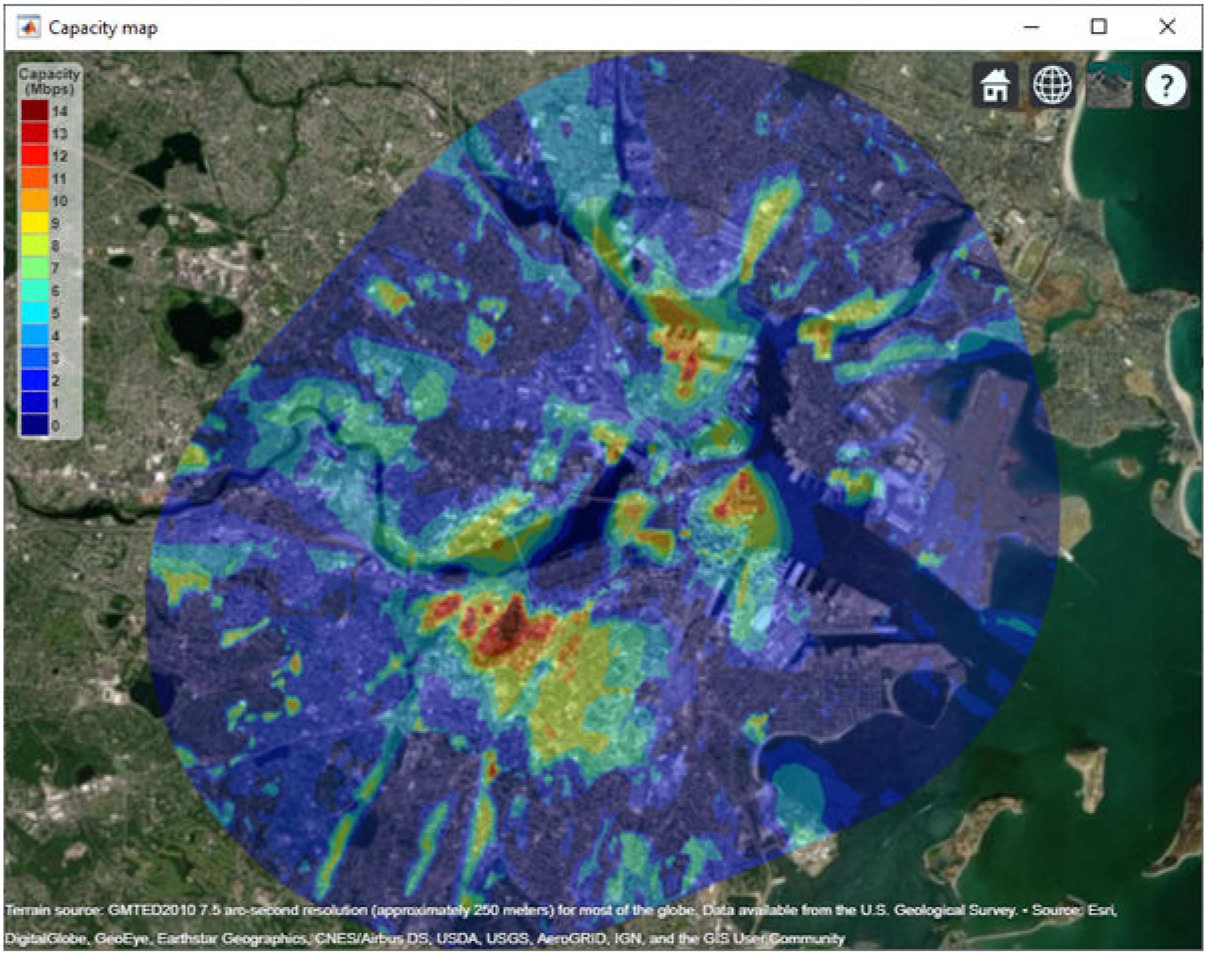
```
pd = sinr(txs,"MaxRange",5000);
[sinrDb,lats,lons] = getDataVariable(pd,"SINR");
```

Compute capacity using the Shannon-Hartley theorem.

```
bw = 1e6; % Bandwidth is 1 MHz
sinrRatio = 10.^(sinrDb./10); % Convert from dB to power ratio
capacity = bw*log2(1+sinrRatio)/1e6; % Unit: Mbps
```

Create new propagation data for the capacity map and display the contour plot.

```
pdCapacity = propagationData(lats,lons,"Capacity",capacity);
sv2 = siteviewer("Name","Capacity map");
legendTitle = "Capacity" + newline + "(Mbps)";
contour(pdCapacity,"LegendTitle",legendTitle);
```


Capacity map
Capacity (Mbps)
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

pd — Propagation data

`propagationData` object (default)

Propagation data, specified as a `propagationData` object.

`varname` — Variable name in data table

character vector | string scalar

Variable name in the data table, specified as a character vector or a string scalar. This variable name must correspond to a variable with numeric data other than the latitude or longitude data.

Output Arguments

`datavariable` — Values of data points

column vector

Values of data points in the propagation data object, returned as a column vector.

`lat` — Latitude of data points
M-by-1 vector

Latitude of data points, returned as an *M*-by-`1` vector with each element unit in degrees.

`lon` — Longitude of data points
M-by-1 vector

Longitude of data points, returned as an *M*-by-`1` matrix with each element unit in degrees. The output is wrapped so that the values are in the range `[-180 180]`.

See Also

Introduced in R2020a

hide

Hide site location on map

Syntax

```
hide(site)
hide(___,Name,Value)
```

Description

`hide(site)` hides the site location of the antenna site on a map.

Note This function only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`hide(___,Name,Value)` hides the site location with additional options specified by one or more name-value pairs.

Examples

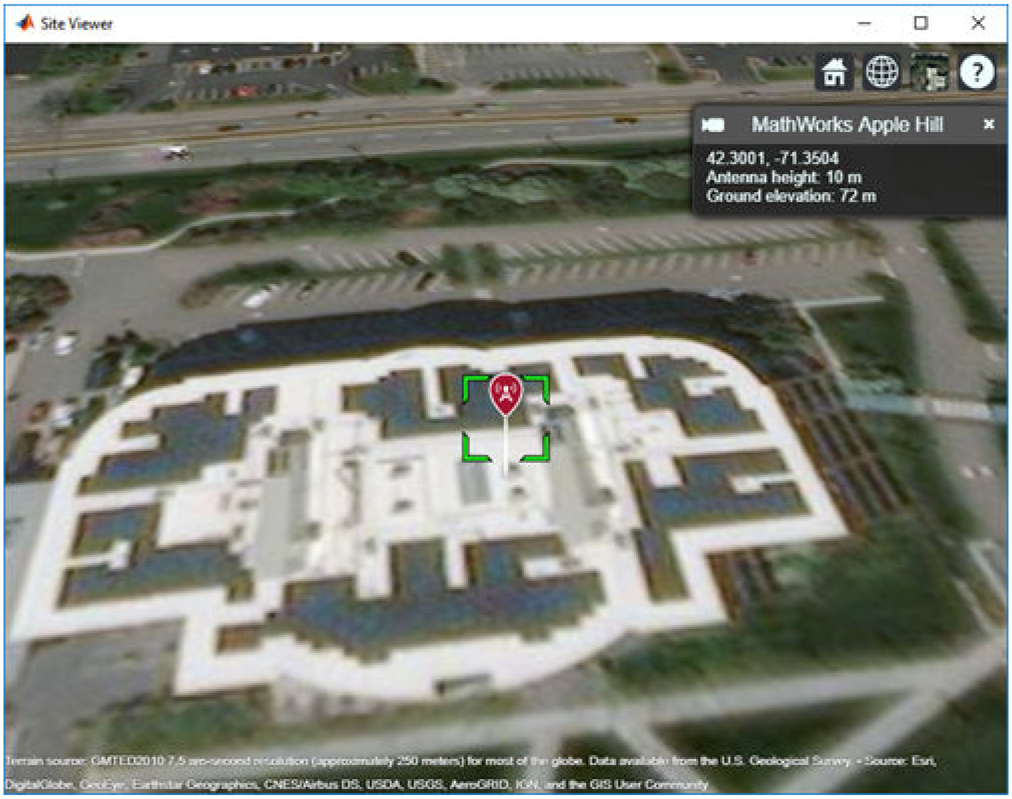
Show and Hide Transmitter Site

Create a transmitter site.

```
tx = txsite('Name','MathWorks Apple Hill',...
       'Latitude',42.3001, ...
       'Longitude',-71.3504);
```

Show the transmitter site.

```
show(tx)
```

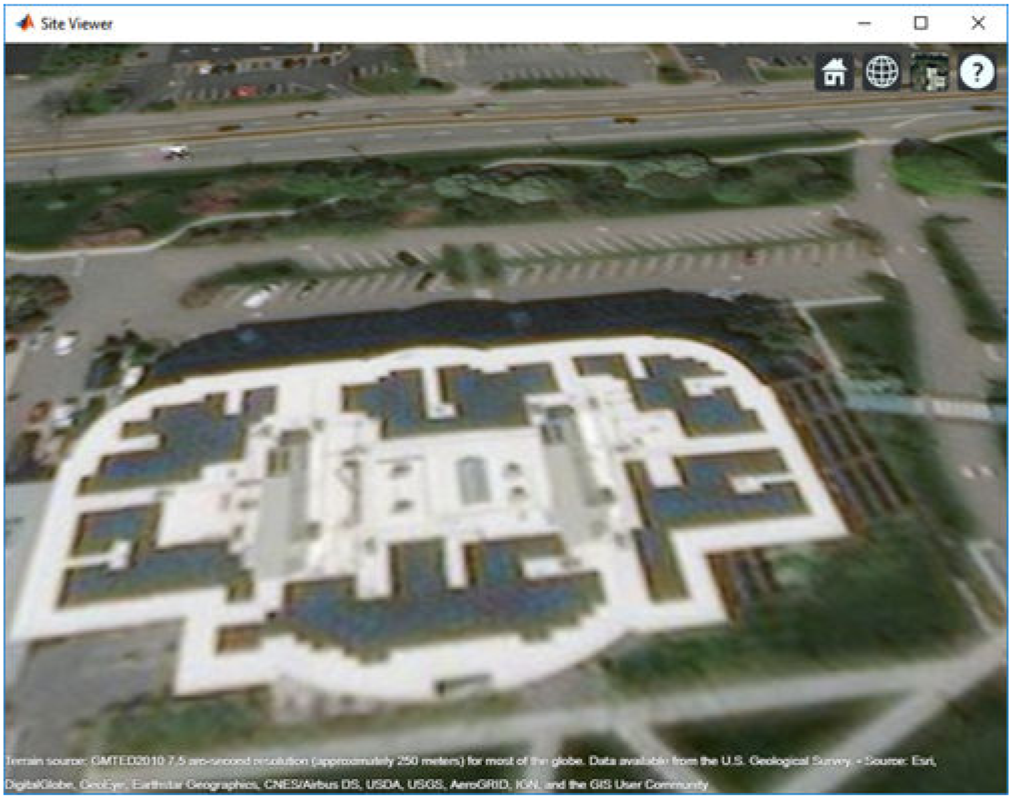


Site Viewer
MathWorks Apple Hill
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

42.3001, -71.3504
Antenna height: 10 m
Ground elevation: 72 m

Hide the transmitter site.

```
hide(tx)
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

`site` — Transmitter or receiver site
`txsite` or `rxsite` object | array of `txsite` or `rxsite` objects

Transmitter or receiver site, specified as a `txsite` or `rxsite` object or an array of `txsite` or `rxsite` objects.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Map','siteviewer1'`

`Map` — Map for visualization of surface data
`siteviewer` object

Map for visualization of surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[5]

Data Types: `char` | `string`

[5] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

See Also

`show`

Introduced in R2019b

interp

Geographic data interpolation

Syntax

```
interpvalue = interp(pd,lat,lon)
interpvalue = interp(pd,Name,Value)
```

Description

`interpvalue = interp(pd,lat,lon)` returns interpolated values from the propagation data for each query point specified in latitude and longitude vectors. The interpolation is performed using a scattered data interpolation method. Values corresponding to query points outside the data region are assigned a `NaN`.

`interpvalue = interp(pd,Name,Value)` returns interpolated values with additional options specified by name-value pair arguments.

Examples

Transmitter Site Service Areas

Define names and locations of sites around Boston.

```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

Create array of transmitter sites.

```
txs = txsite("Name", names,...
       "Latitude",lats,...
       "Longitude",lons, ...
       "TransmitterFrequency",2.5e9);
```

Compute received power data for each transmitter site.

```
maxr = 20000;
pd1 = coverage(txs(1),"MaxRange",maxr);
pd2 = coverage(txs(2),"MaxRange",maxr);
pd3 = coverage(txs(3),"MaxRange",maxr);
```

Compute rectangle containing locations of all data.

```
locs = [location(pd1); location(pd2); location(pd3)];
[minlatlon, maxlatlon] = bounds(locs);
```

Create grid of locations over rectangle.

```
gridlength = 300;
latv = linspace(minlatlon(1),maxlatlon(1),gridlength);
```

```
lonv = linspace(minlatlon(2),maxlatlon(2),gridlength);
[lons,lats] = meshgrid(lonv,latv);
lats = lats(:);
lons = lons(:);
```

Get data for each transmitter at grid locations using interpolation.

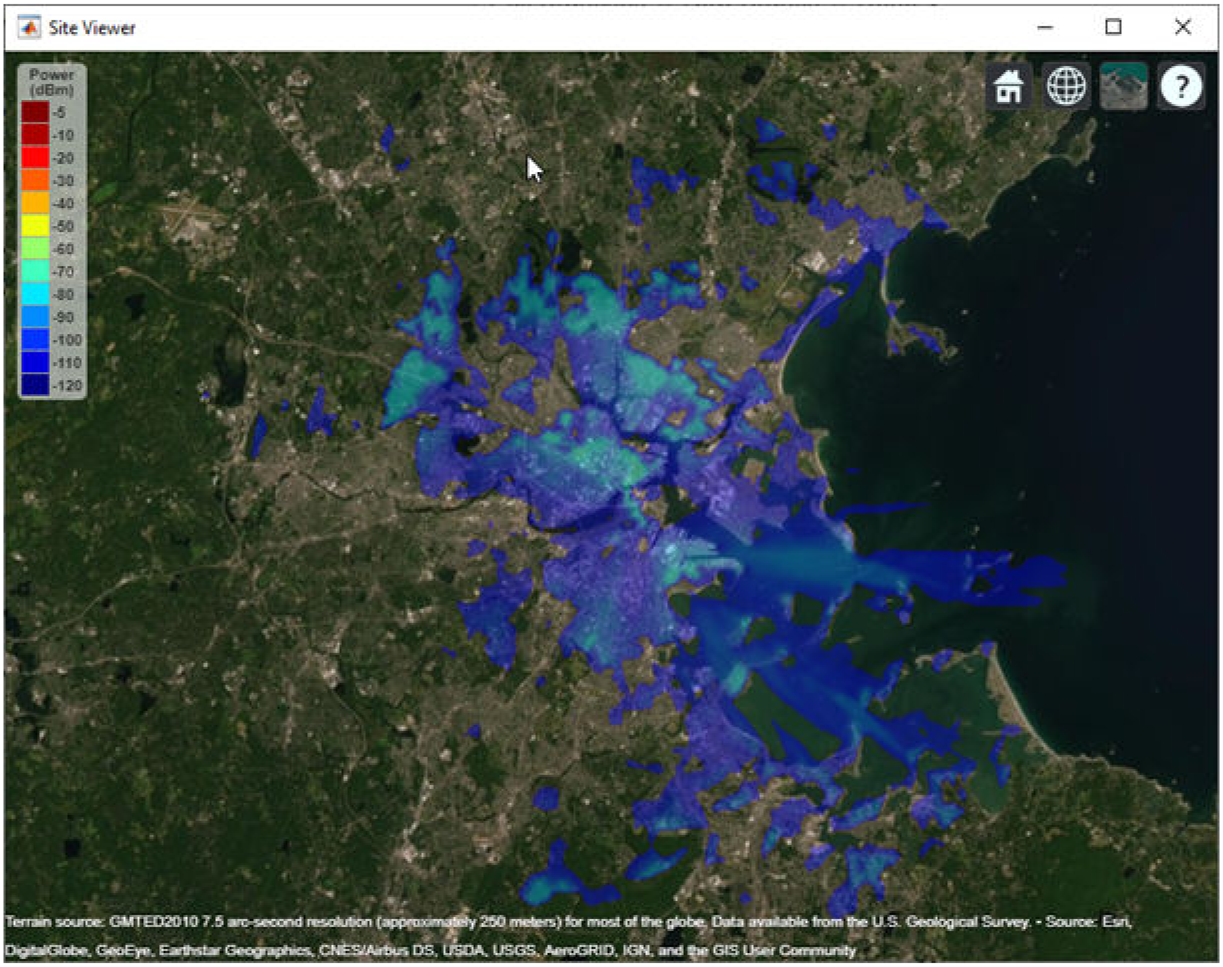
```
v1 = interp(pd1,lats,lons);
v2 = interp(pd2,lats,lons);
v3 = interp(pd3,lats,lons);
```

Create propagation data containing minimum received power values.

```
minReceivedPower = min([v1 v2 v3],[],2,"includenan");
pd = propagationData(lats,lons,"MinReceivedPower",minReceivedPower);
```

Plot minimum received power, which shows the weakest signal received from any transmitter site. The area shown may correspond to the service area of triangulation using the three transmitter sites.

```
sensitivity = -110;
contour(pd,"Levels",sensitivity:-5,"Type","power")
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

pd — Propagation data
propagationData object (default)

Propagation data, specified as a propagationData object.

lat — Latitude coordinate values
vector

Latitude coordinate values, specified as a vector in decimal degrees with reference to Earth's ellipsoid. model WGS-84. The latitude coordinates must be in the range [-90 90].

lon — Longitude coordinate values
vector

Longitude coordinate values, specified as a vector in decimal degrees with reference to Earth's ellipsoid. model WGS-84.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Method','linear'

DataVariableName — Data variable to interpolate
character vector | string scalar

Data variable to interpolate, specified as the comma-separated pair consisting of 'DataVariableName' and a character vector or string scalar corresponding to a variable name in the data table used to create the propagationData container object. The default value is the DataVariableName property in the propagationData.

Data Types: char | string

Method — Method used to interpolate data
'natural' (default) | 'nearest' | 'linear'

Method used to interpolate data, specified as the comma separated-pair consisting 'Method' and one of the following:

- 'natural' - Natural neighbor interpolation
- 'linear' - Linear interpolation
- 'nearest' - Nearest neighbor interpolation

Data Types: char | string

Output Arguments

interpvalue — Interpolated values from propagation data
numeric vector

Interpolated values from the propagation data for each query point specified in latitude and longitude vectors, returned as a numeric vector.

See Also

Introduced in R2020a

link

Display communication link on map

Syntax

```
link(rx,tx)
link(rx,tx,propmodel)
link( ___ ,Name,Value)
status = link( ___ )
```

Description

`link(rx,tx)` plots a one-way point-to-point communication link between a receiver site and transmitter site. The plot is color coded to identify the link success status.

`link(rx,tx,propmodel)` plots the communication link based on the specified propagation model.

`link( ___ ,Name,Value)`plots a communication link using additional options specified by `Name,Value` pairs.

`status = link( ___ )` returns the success status of the communication link as `true` or `false`.

Examples

Communication Link Between Transmitter and Receiver

Create a transmitter site.

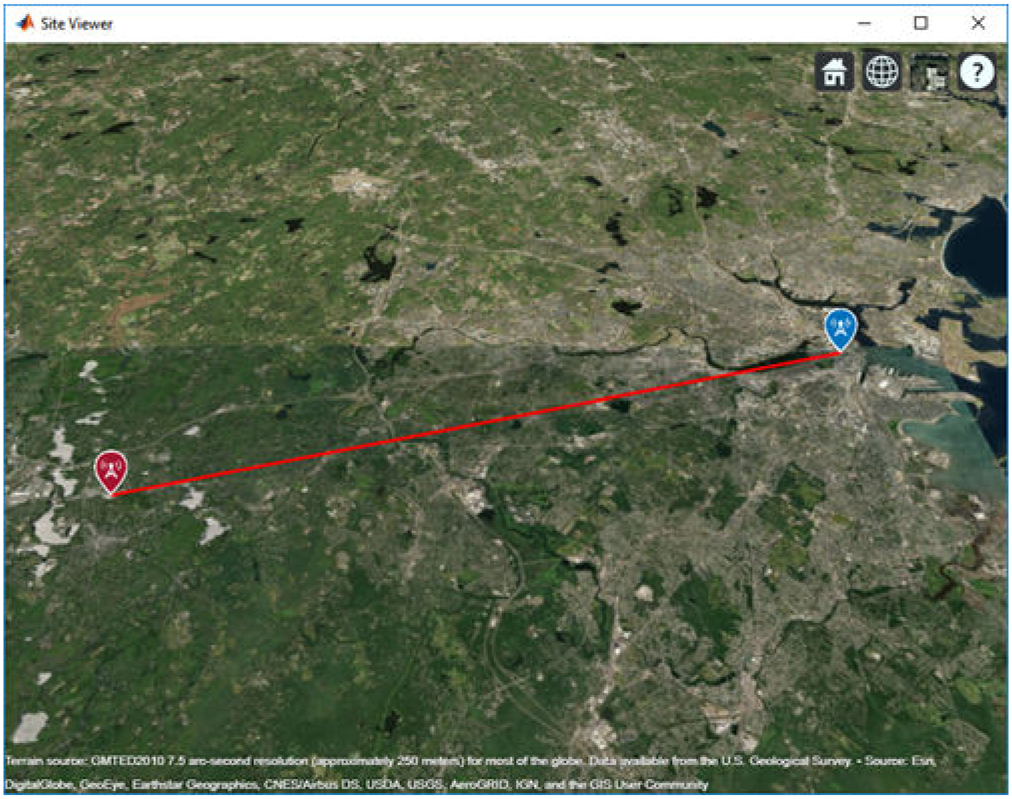
```
tx = txsite('Name','MathWorks', ...
        'Latitude', 42.3001, ...
        'Longitude', -71.3503);
```

Create a receiver site with sensitivity defined in dBm.

```
 rx = rxsite('Name','Boston', ...
        'Latitude', 42.3601, ...
        'Longitude', -71.0589, ...
        'ReceiverSensitivity', -90);
```

Plot the communication link between the transmitter and the receiver.

```
link(rx,tx)
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

rx — Receiver site

`rxsite` object | array of `rxsite` objects

Receiver site, specified as a `rxsite` object. You can use array inputs to specify multiple sites.

tx — Transmitter site

`txsite` object | array of `txsite` objects

Transmitter site, specified as a `txsite` object. You can use array inputs to specify multiple sites.

propmodel — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can use the `propagationModel` function to define this input. The default value depends on the coordinate system used by the input sites:

Coordinate System	Default propagation model value
`'geographic'`	• `'longley-rice'` when you use a terrain. • `'freespace'` when you do not use a terrain.

Coordinate System	Default propagation model value
'cartesian'	• 'freespace' when Map is set to none. • 'raytracing-image-method' when Map is set to the name of an STL file or a triangulation object.

You can also use the name-value pair 'PropagationModel' to specify this parameter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Type','power'

PropagationModel — Propagation model to use for path loss calculations

'longley-rice' (default) | 'freespace' | 'close-in' | 'rain' | 'gas' | 'fog' | 'raytracing-image-method' | propagation model object

Propagation model to use for the path loss calculations, specified as the comma-separated pair consisting of 'PropagationModel' and one of the following:

- 'freespace' - Free space propagation model
- 'rain' - Rain propagation model
- 'gas' - Gas propagation model
- 'fog' - Fog propagation model
- 'close-in' - Close-in propagation model
- 'lonley-rice' - Longely-Rice propagation model
- 'tirem' - Tirem propagation model
- 'ray-tracing-image-method' - -Raytracing propagation model using method of images.

The default propagation model is 'longley-rice' when terrain is enabled and 'freespace' when terrain is disabled.

Terrain propagation models including 'longley-rice' and 'tirem' are only supported for sites with CoordinateSystem property set to 'geographic'.

Data Types: char

SuccessColor — Color of successful links

'green' (default) | RGB triplet | character vector

Color of successful links, specified as the comma-separated pair consisting of 'SuccessColor and an RGB triplet or character vector. For more information, see ColorSpec (Color Specification).

Data Types: char | double

FailColor — Color of unsuccessful links

'red' (default) | RGB triplet | character vector

Color of unsuccessful links, specified as the comma-separated pair consisting of `'FailColor` and RGB triplet or character vector. For more information, see `ColorSpec (Color Specification)`.

Data Types: `char` | `double`

`Map` — Map for visualization or surface data
`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
`'geographic'`	• `siteviewer`[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are `'none'`, `'gmted2010'`, or the name of the custom terrain data added using `addCustomTerrain`	• current siteviewer or new siteviewer if none are open. • `'gmted2010'` if called with an output.
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

`status` — Success status of communication link
M-by-*N* array

Success status of communication links, returned as an *M*-by-*N* arrays. *M* is the number of transmitter sites and *N* is the number of receiver sites.

See Also

`coverage` | `los` | `propagationModel` | `sigstrength` | `sinr`

Topics

`ColorSpec (Color Specification)`

Introduced in R2019b

location

Location coordinates at a given distance and angle from site

Syntax

```
sitelocation = location(site)
[lat,lon] = location(site)
[ ___ ] = location(site,distance,azimuth)
```

Description

`sitelocation = location(site)` returns the site location of the antenna.

`[lat,lon] = location(site)` returns the latitude and longitude of the antenna site.

Note This syntax only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`[ ___ ] = location(site,distance,azimuth)` returns the new location achieved by moving the antenna site by the distance specified in the direction of the azimuth angle. The location is calculated by moving along a great circle path using a spherical Earth model.

Note This syntax only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

Examples

Location of Antenna Site

Create a site 1 km north of a given site.

Create the first transmitter site.

```
tx = txsite('Name','MathWorks', ...
       'Latitude',42.3001, ...
       'Longitude',-71.3504);
```

Calculate the location 1 km north of the first site.

```
[lat,lon] = location(tx,1000,90)
```

```
lat = 42.3091
```

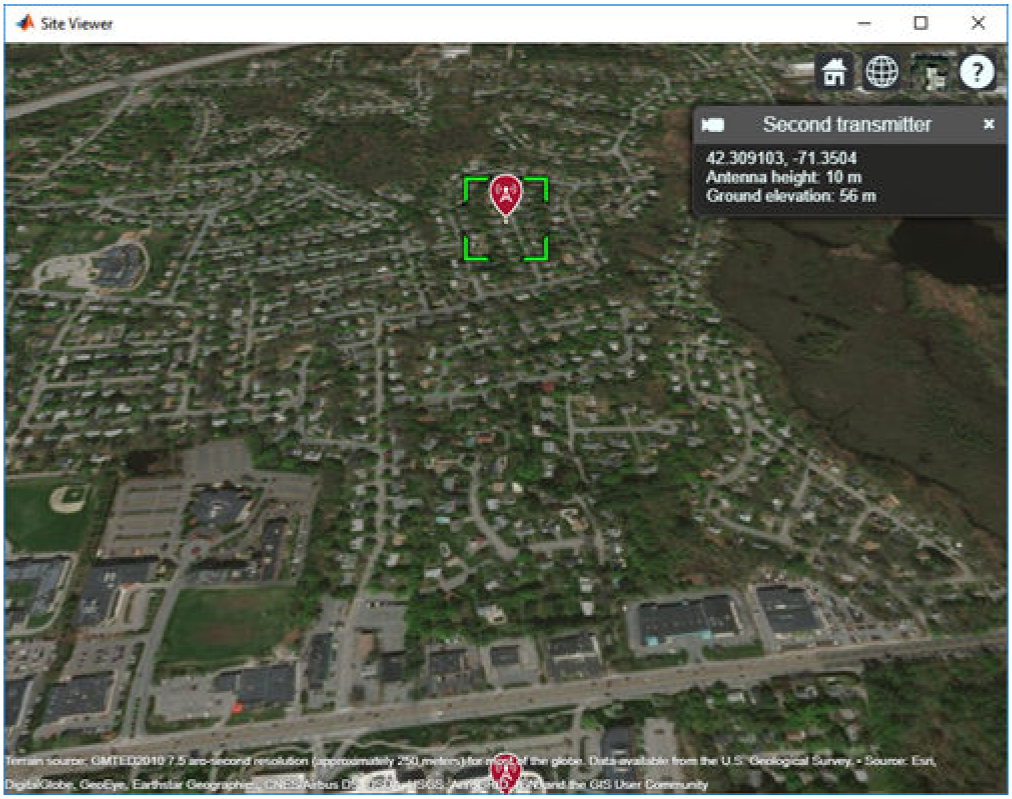
```
lon = -71.3504
```

Create a second transmitter site at the location specified by `lat` and `lon`.

```
tx2 = txsite('Name','Second transmitter', ...
       'Latitude',lat, ...
       'Longitude',lon);
```

Show the two transmitter sites.

```
show([tx,tx2])
```



Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Second transmitter

42.309103, -71.3504
Antenna height: 10 m
Ground elevation: 56 m

Input Arguments

site — Antenna site

scalar | array

Antenna site, specified as a scalar or an array. It is either a txsite or a rxsite object. For more information, see `txsite`, and `rxsite`

Note If `distance` or `azimuth` is a vector, then site must be a scalar.

distance — Distance to move antenna site

scalar | vector

Distance to move antenna site, specified as a scalar or vector in meters.

azimuth — Azimuth angle

scalar | vector

Azimuth angle, specified as a scalar or vector in degrees. Azimuth angle is measured counterclockwise from due east.

Output Arguments

`sitelocation` — Location of antenna site

M-by-2 matrix

Location of antenna site, returned as an *M*-by-`2` matrix with each element unit in degrees. *M* is the number of sites in sites. The location value includes the latitude and longitude of the antenna site.

If the antenna site has the `CoordinateSystem` property set to `'geographic'`, *L* is a 1-by-2 vector in degrees latitude and longitude. The output longitude wrapped so that values are in the range `[-180 180]`. If SITE has `CoordinateSystem` set to `'cartesian'`, *L* is a 1-by-3 vector.

`lat` — Latitude of one or more antenna sites

M-by-1 vector

Latitude of one or more antenna sites, returned as an *M*-by-`1` vector with each element unit in degrees. *M* is the number of sites in `site`.

`lon` — Longitude of one or more antenna sites

M-by-1 matrix

Longitude of one or more antenna sites, returned as an *M*-by-`1` matrix with each element unit in degrees. *M* is the number of sites in `site`. The output is wrapped so that the values are in the range `[-180 180]`.

See Also

`angle` | `distance` | `rxsite` | `txsite`

Introduced in R2019b

los

Plot or compute the line-of-sight (LOS) visibility between sites on a map

Syntax

```
los(site1,site2)
los(site1,site2,Name,Value)
vis = los(site1,site2,Name,Value)
```

Description

`los(site1,site2)` plots the LOS from site 1 to site 2. The plot is color coded to identify the visibility of the points along the LOS.

`los(site1,site2,Name,Value)` sets properties using one or more name-value pairs. For example, `los(site1,site2,'ObstructedColor','red')` plots the LOS using red to show blocked visibility.

`vis = los(site1,site2,Name,Value)` returns the status of the LOS visibility.

Examples

LOS from a Transmitter Site to a Receiver Site

Plot the LOS from the MathWorks Apple Hill campus to the MathWorks Lakeside campus.

Create a transmitter site with an antenna of height 30 m.

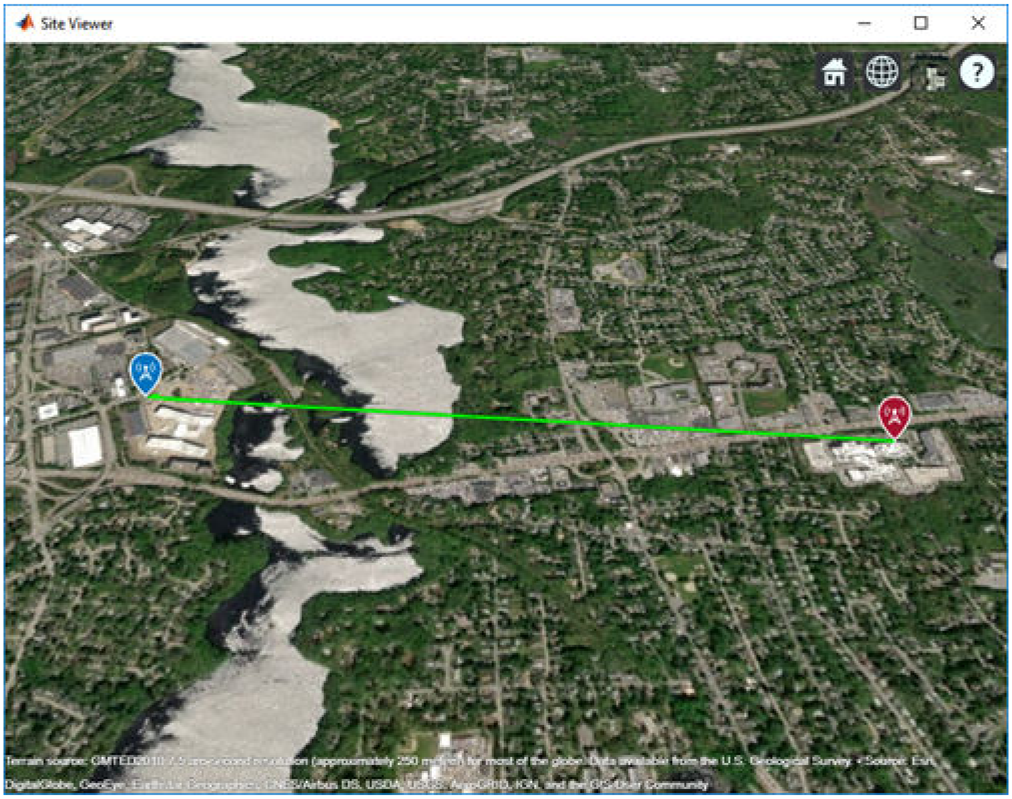
```
tx = txsite('Name','MathWorks Apple Hill',...
          'Latitude',42.3001,'Longitude',-71.3504,'AntennaHeight',30);
```

Create a receiver site with an antenna at ground level.

```
rx = rxsite('Name','MathWorks Lakeside', ...
           'Latitude',42.3021,'Longitude',-71.3764);
```

Plot the LOS between the two sites.

```
los(tx,rx);
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

LOS from a Transmitter Site to Two Receiver Sites

Create a transmitter site with an antenna of height 30 m.

```
tx = txsite('Name','MathWorks Apple Hill',...
          'Latitude',42.3001,'Longitude',-71.3504,'AntennaHeight',30);
```

Create two receiver sites with antennas at ground level.

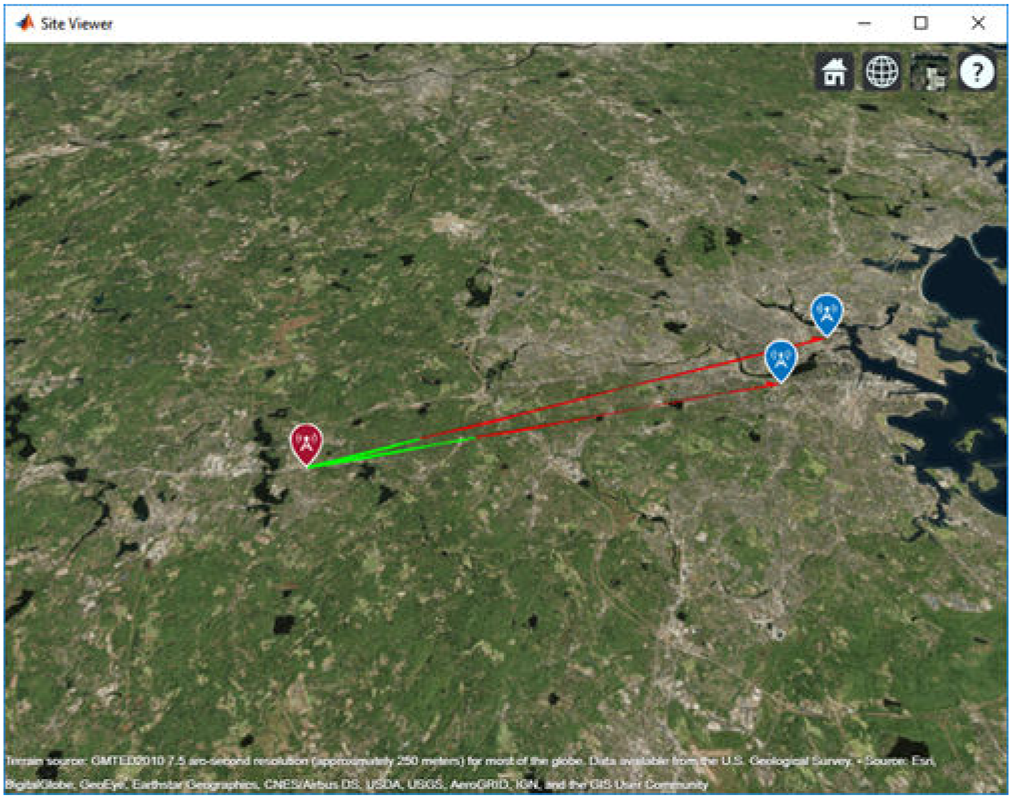
```
names = ["Fenway Park","Bunker Hill Monument"];
lats = [42.3467,42.3763];
lons = [-71.0972,-71.0611];
```

Create the receiver site array.

```
rxs = rxsite('Name', names,...
      'Latitude',lats,...
      'Longitude',lons);
```

Plot the lines of sight to the receiver sites. The red portion of the LOS represents obstructed visibility.

```
los(tx,rxs);
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

`site1` — Source antenna site

`txsite` object | `rxsite` object

Source antenna site, specified as a `txsite` object or a `rxsite` object. Site 1 must be a single site object.

`site2` — Target antenna site

`txsite` object | `rxsite` object | vector of `txsite` or `rxsite` objects

Target antenna site, specified as a `txsite` object or a `rxsite` object. Site 2 can be a single site object or a vector of multiple site objects.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ObstructedColor','blue'`

`VisibleColor` — Plot color for successful visibility

`'green'` (default) | RGB triplet | character vector | color name string

Plot color for successful visibility, specified as an RGB triplet, a character vector, or a color name specified as a string. For more information, see `ColorSpec (Color Specification)`.

`ObstructedColor` — Plot color for blocked visibility

`'red'` (default) | RGB triplet | character vector | color name string

Plot color for blocked visibility, specified as an RGB triplet, a character vector, or a color name specified as a string. For more information, see `ColorSpec (Color Specification)`.

`Resolution` — Sampling distance between two sites

`'auto'` (default) | numeric scalar

Resolution of sample locations used to compute line-of-sight visibility, specified as `'auto'` or a numeric scalar expressed in meters. `Resolution` defines the distance between samples on the great circle path using a spherical Earth model. If `Resolution` is `'auto'`, the function computes a value based on the distance between the sites.

`Map` — Map for visualization or surface data

`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
`'geographic'`	• `siteviewer`[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are `'none'`, `'gmted2010'`, or the name of the custom terrain data added using `addCustomTerrain`	• current siteviewer or new siteviewer if none are open. • `'gmted2010'` if called with an output.
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

`vis` — Status of LOS visibility

`'true'` | `'false'` | *n*-by 1 logical array

Status of LOS visibility, returned as `'true'` or `'false'`. If there are multiple target sites, the function returns a logical array of *n*-by-1.

See Also

`angle` | `distance` | `link`

Topics

`ColorSpec (Color Specification)`

Introduced in R2019b

pathloss

Path loss of radio wave propagation

Syntax

```
pl = pathloss(propmodel,rx,tx)
pl = pathloss( ___ ,Name,Value)
[pl,info] = pathloss( ___ )
```

Description

`pl = pathloss(propmodel,rx,tx)` returns the path loss of radio wave propagation at the receiver site from the transmitter site.

`pl = pathloss( ___ ,Name,Value)` returns the path loss using additional options specified by `Name,Value` pairs.

`[pl,info] = pathloss( ___ )` returns the path loss and the information about the propagation paths.

Examples

Path Loss of Receiver In Heavy Rain

Specify the transmitter and the receiver sites.

```
tx = txsite('Name','MathWorks Apple Hill',...
       'Latitude',42.3001, ...
       'Longitude',-71.3504, ...
       'TransmitterFrequency', 2.5e9);

rx = rxsite('Name','Fenway Park',...
       'Latitude',42.3467, ...
       'Longitude',-71.0972);
```

Create the propagation model for heavy rainfall rate.

```
pm = propagationModel('rain','RainRate',50)
```

```
pm = 
  Rain with properties:

    RainRate: 50
        Tilt: 0
```

Calculate the pathloss at the receiver using the rain propagation model.

```
pl = pathloss(pm,rx,tx)
```

```
pl = 127.1559
```

Input Arguments

`propmodel` — Propagation model

character vector or string

Propagation model, specified as a character vector or string.

Data Types: `char`

`rx` — Receiver site

`rxsite` object

Receiver site, specified as a `rxsite` object. You can use array inputs to specify multiple sites.

Data Types: `char`

`tx` — Transmitter site

`txsite` object

Transmitter site, specified as a `txsite` object. You can use array inputs to specify multiple sites.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Map','none'`

`Map` — Map for visualization or surface data

`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
`'geographic'`	• `siteviewer`[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are `'none'`, `'gmted2010'`, or the name of the custom terrain data added using `addCustomTerrain`	• current siteviewer or new siteviewer if none are open. • `'gmted2010'` if called with an output.
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

pl — Path loss
scalar | *M*-by-*N* arrays

Path loss, returned as a scalar or *M*-by-*N* cell arrays containing a row vector of path loss values in decibels. *M* is the number of TX sites and *N* is the number of RX sites.

Path loss is computed along the shortest path shortest path through space connecting the transmitter and receiver antenna centers.

For terrain propagation models, path loss is computed using terrain elevation profile that is computed at sample locations on the great circle path between the transmitter and the receiver. If `Map` is a `siteviewer` object with buildings specified, the terrain elevation is adjusted to include the height of the buildings.

`info` — Information corresponding to each propagation path
M-by-*N* struct array | *M*-by-*N* cell array containing vector of structs in each cell

Information corresponding to each propagation path, returned as a *M*-by-*N* cell array containing vector of structs in each cell for `ray-tracing-image-method` propagation model and *M*-by-*N* struct array fro all other propagation models. The field and values for the structures are:

- `PropagationDistance` - Total distance of propagation path returned as a double scalar in meters.
- `AngleOfDeparture` - Angle of departure of signal from transmitter site antenna returned as a 2-by-1 double vector of azimuth and elevation angles in degrees.
- `AngleOfArrival` - Angle of arrival of signal at receiver site antenna returned as a 2-by-1 double vector of azimuth and elevation angles in degrees.
- `NumReflections` - Number of reflections undergone by signal along propagation path, returned specified as `0`, `1`, or `2`. This field and value is only for `raytrtacing-image-method`.

Angle values in this structure are defined using the antenna's local East-North-Up coordinate system when `CoordinateSystem` is set to `geographic`. Angle values in this structure are defined using global Cartesian coordinate system when `CoordinateSystem` is set to `cartesian`. Azimuth angle is measured either from east (when `'geographic'`) or from the global x-axis around the global z-axis (when `'cartesian'`). Elevation angle is measured from the horizontal (or X-Y) plane to the antenna's x-axis in the range -90 to 90.

See Also

`propagationModel` | `range`

Introduced in R2019b

plot

Plot propagation data on map

Syntax

```
plot(pd)
plot(___,Name,Value)
```

Description

`plot(pd)` plots the propagation data on a map. Each data point is displayed as a circular marker that is colored according to the corresponding value.

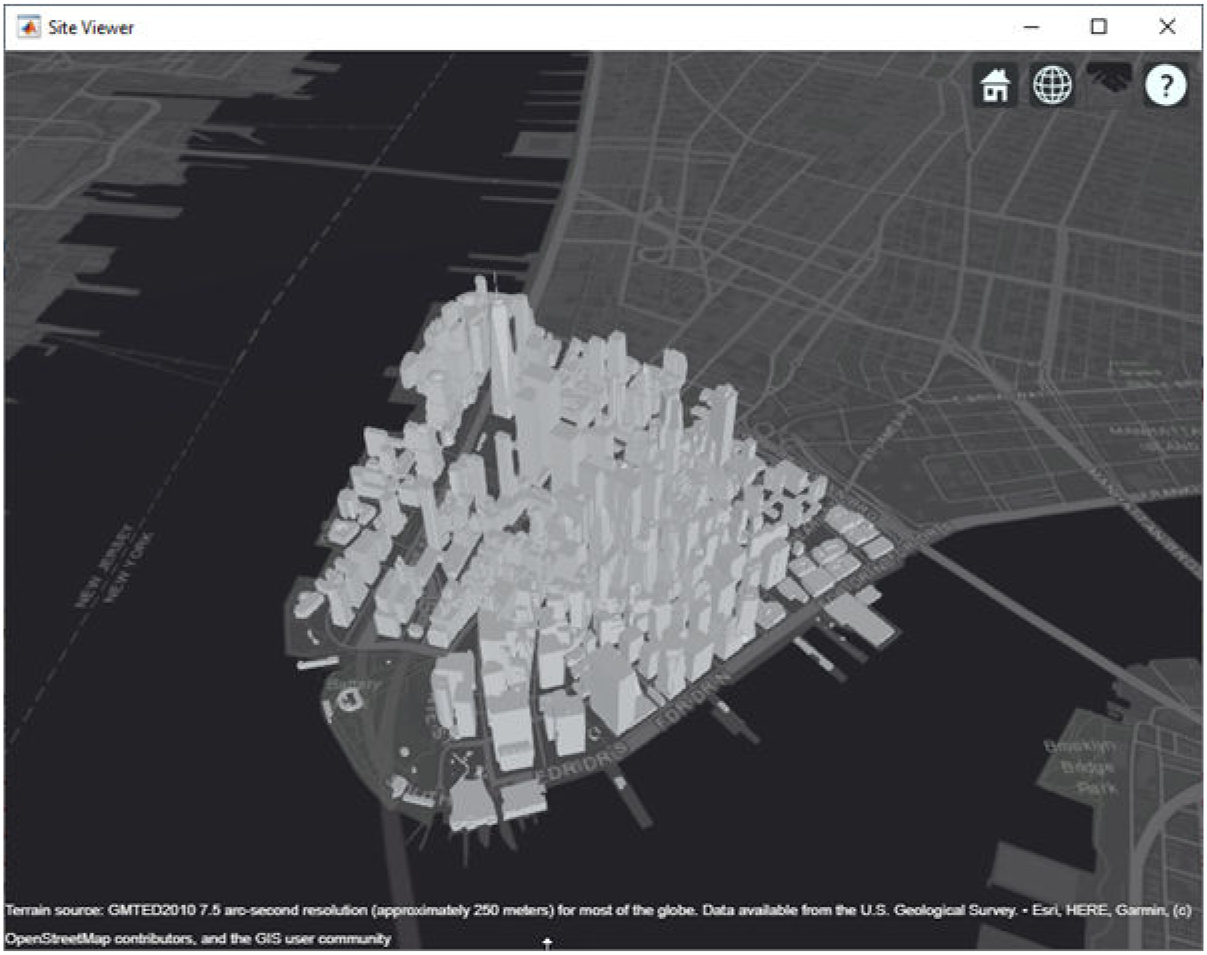
`plot(___,Name,Value)` plots the propagation data with additional options specified by name-value pair arguments.

Examples

Compute Signal Strength Data in Urban Environment

Launch Site Viewer with basemaps and building files for Manhattan. For more information about the osm file, see [1] on page 4-0 .

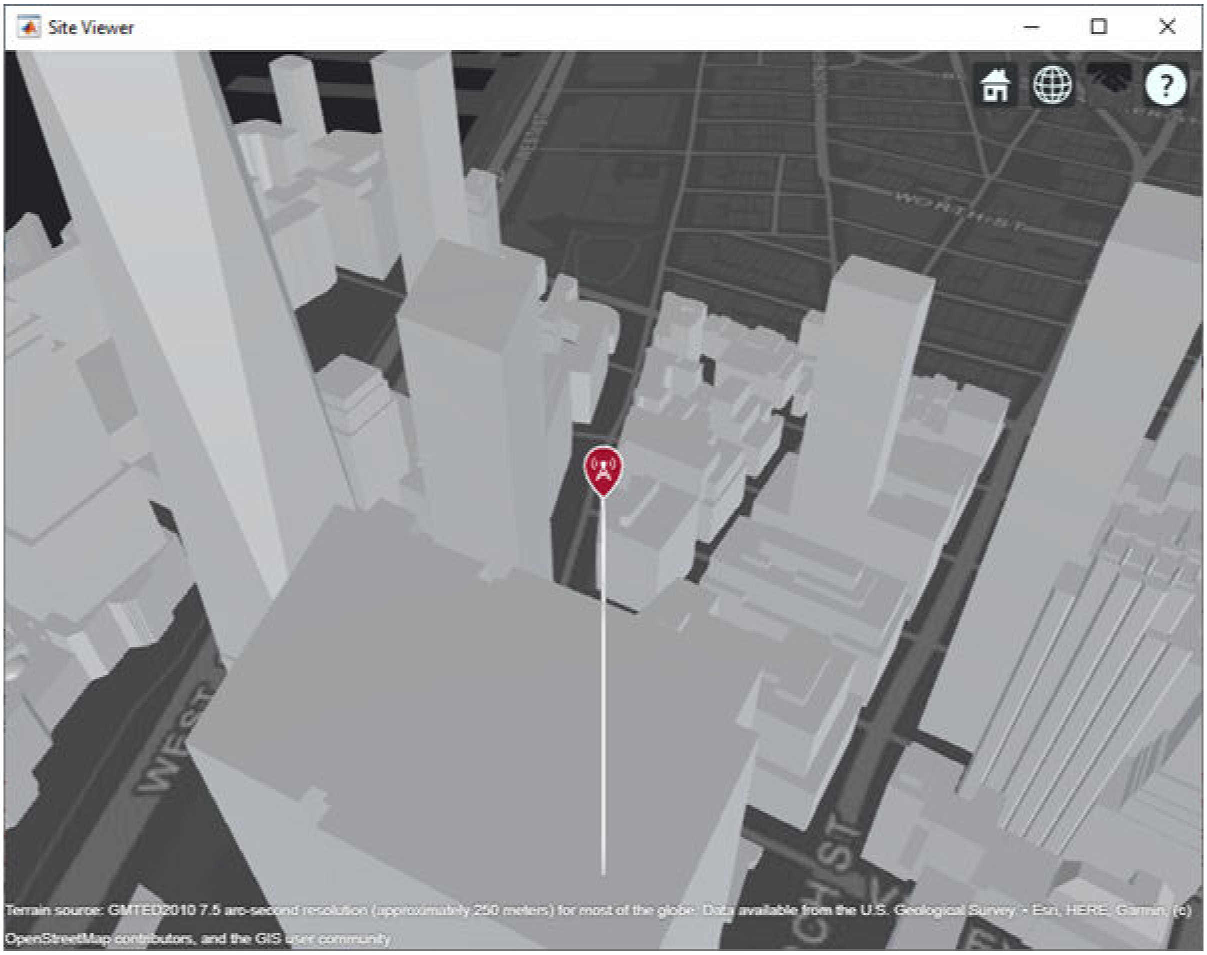
```
viewer = siteviewer("Basemap","streets_dark",...
        "Buildings","manhattan.osm");
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Show a transmitter site on a building.

```
tx = txsite("Latitude",40.7107,...
        "Longitude",-74.0114,...
        "AntennaHeight",80);
show(tx)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Create receiver sites along nearby streets.

```
latitude = [linspace(40.7088, 40.71416, 50), ...
        linspace(40.71416, 40.715505, 25), ...
        linspace(40.715505, 40.7133, 25), ...
        linspace(40.7133, 40.7143, 25)]';
longitude = [linspace(-74.0108, -74.00627, 50), ...
        linspace(-74.00627 ,-74.0092, 25), ...
        linspace(-74.0092, -74.0110, 25), ...
        linspace(-74.0110, -74.0132, 25)]';
rxs = rxsite("Latitude", latitude, "Longitude", longitude);
```

Compute signal strength at each receiver location.

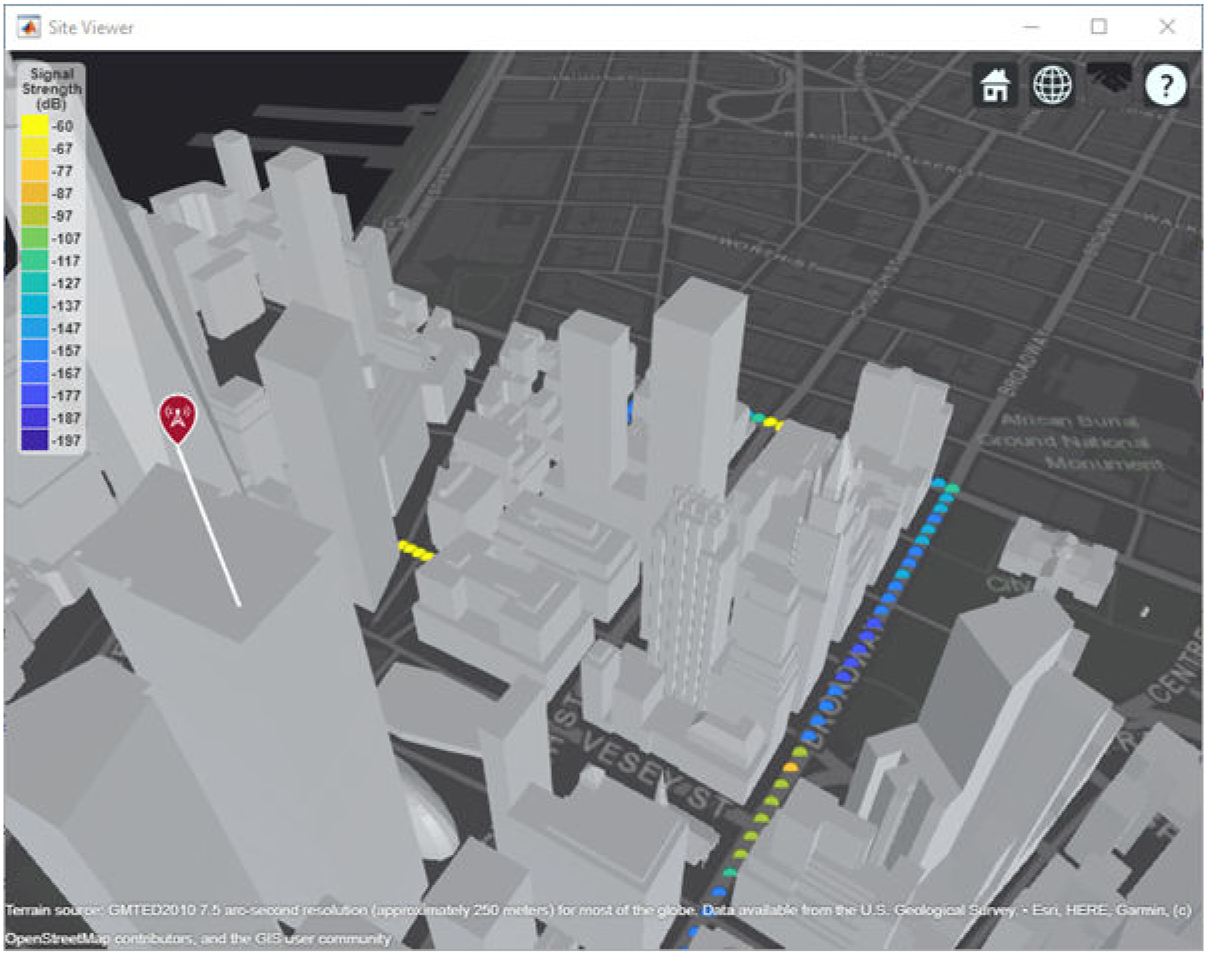
```
signalStrength = sigstrength(rxs, tx)';
```

Create a `propagationData` object to hold computed signal strength data.

```
tbl = table(latitude, longitude, signalStrength);
pd = propagationData(tbl);
```

Plot the signal strength data on a map as colored points.

```
legendTitle = "Signal" + newline + "Strength" + newline + "(dB)";
plot(pd, "LegendTitle", legendTitle, "Colormap", parula);
```


Site Viewer
Signal Strength (dB)
-60
-67
-77
-87
-97
-107
-117
-127
-137
-147
-157
-167
-177
-187
-197
VESEY ST
BROADWAY
African Burial Ground National Monument
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Esri, HERE, Garmin, (c) OpenStreetMap contributors, and the GIS user community

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

pd — Propagation data

`propagationData` object (default)

Propagation data, specified as a `propagationData` object.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','power'`

DataVariableName — Data variable to plot

pd.DataVariableName (default) | character vector | string scalar

Data variable to plot, specified as the comma-separated pair consisting of 'DataVariableName' and a character vector or a string scalar corresponding to a variable name in the data table used to create the propagation data container object pd. The default value is dynamic and corresponds to the DataVariableName property of the propagationData object.

Data Types: char | string

Type — Type of value to plot

'custom' (default) | 'power' | 'efield' | 'sinr' | 'pathloss'

Type of value to plot, specified as the comma-separated pair consisting of 'Type' and one of the values in the Type column:

Type	ColorLimits	LegendTitle
'custom'	[min(Data) max(Data)]	''
'power'	[-120 -5]	'Power (dBm)'
'efield'	[20 135]	'E-field (dBuV/m)'
'sinr'	[-5 20]	'SINR (dB)'
'pathloss'	[45 160]	'Path loss (dB)'

The default value for Levels is a linearly spaced vector bounded by ColorLimits.

Data Types: char | string

Levels — Data value levels to plot

numeric vector

Data value levels to plot, specified as the comma-separated pair consisting of 'Levels' and a numeric vector. The propagation data is binned according to Levels. The data in each bin is color coded according to the corresponding level. The colors are selected using Colors if specified, or else Colormap and ColorLimits. Data points with values below the minimum level are not included in the plot.

The default value for Levels is a linearly spaced vector bounded by ColorLimits.

Data Types: double

Colors — Colors of data points

M-by-3 array of RGB | array of strings | cell array of character vectors

Colors of the data points, specified as the comma-separated pair consisting of 'Colors' and an *M*-by-3 array of RGB (red, blue, green) or an array of strings, or a cell array of character vectors. Colors are assigned element-wise to values in Levels for coloring the corresponding points. Colors cannot be used with Colormap and ColorLimits.

Data Types: double | char | string

Colormap — Color map for coloring points

'jet(256)' (default) | predefined colormap name | *M*-by-3 array of RGB triplets

Colormap for the coloring points, specified as the comma-separated pair consisting of `'Colormap'` and a predefined colormap name or an *M*-by-3 array of RGB (red, blue, green) triplets that define *M* individual colors. `Colormap` cannot be used with `Colors`.

Data Types: `double` | `char` | `string`

`ColorLimits` — Color limits for color map

two-element vector

Color limits for the colormap, specified as the comma-separated pair consisting of `'ColorLimits'` and a two-element vector of the form [min max]. The color limits indicate the data level values that map to the first and last colors in the colormap. `ColorLimits` cannot be used with `Colors`.

Data Types: `double`

`MarkerSize` — Size of data markers

`10` (default) | positive numeric scalar

Size of data markers plotted on the map, specified as the comma-separated pair consisting of `'MarkerSize'` and a positive numeric scalar in pixels.

Data Types: `double`

`ShowLegend` — Show color legend on map

`true` (default) | `false`

Show color legend on map, specified as the comma-separated pair consisting of `'ShowLegend'` and `true` or `false`.

Data Types: `logical`

`LegendTitle` — Title of color legend

character vector | string scalar

Title of color legend, specified as the comma-separated pair consisting of `'LegendTitle'` and a character vector or a string scalar.

Data Types: `string` | `char`

`Map` — Map for surface data

`siteviewer` object

Map for surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[6] The default value is the current Site Viewer or a new Site Viewer, if none is open.

Data Types: `char` | `string`

See Also

Introduced in R2020a

[6] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

location

Data location coordinates

Syntax

```
datalocation = location(pd)
[lat,lon] = location(pd)
```

Description

`datalocation = location(pd)` returns the location coordinates of the data points in the propagation data object.

`[lat,lon] = location(pd)` returns the latitude and longitude of the propagation data object

Examples

Transmitter Site Service Areas

Define names and locations of sites around Boston.

```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

Create array of transmitter sites.

```
txs = txsite("Name", names,...
       "Latitude",lats,...
       "Longitude",lons, ...
       "TransmitterFrequency",2.5e9);
```

Compute received power data for each transmitter site.

```
maxr = 20000;
pd1 = coverage(txs(1),"MaxRange",maxr);
pd2 = coverage(txs(2),"MaxRange",maxr);
pd3 = coverage(txs(3),"MaxRange",maxr);
```

Compute rectangle containing locations of all data.

```
locs = [location(pd1); location(pd2); location(pd3)];
[minlatlon, maxlatlon] = bounds(locs);
```

Create grid of locations over rectangle.

```
gridlength = 300;
latv = linspace(minlatlon(1),maxlatlon(1),gridlength);
lonv = linspace(minlatlon(2),maxlatlon(2),gridlength);
[lons,lats] = meshgrid(lonv,latv);
lats = lats(:);
lons = lons(:);
```

Get data for each transmitter at grid locations using interpolation.

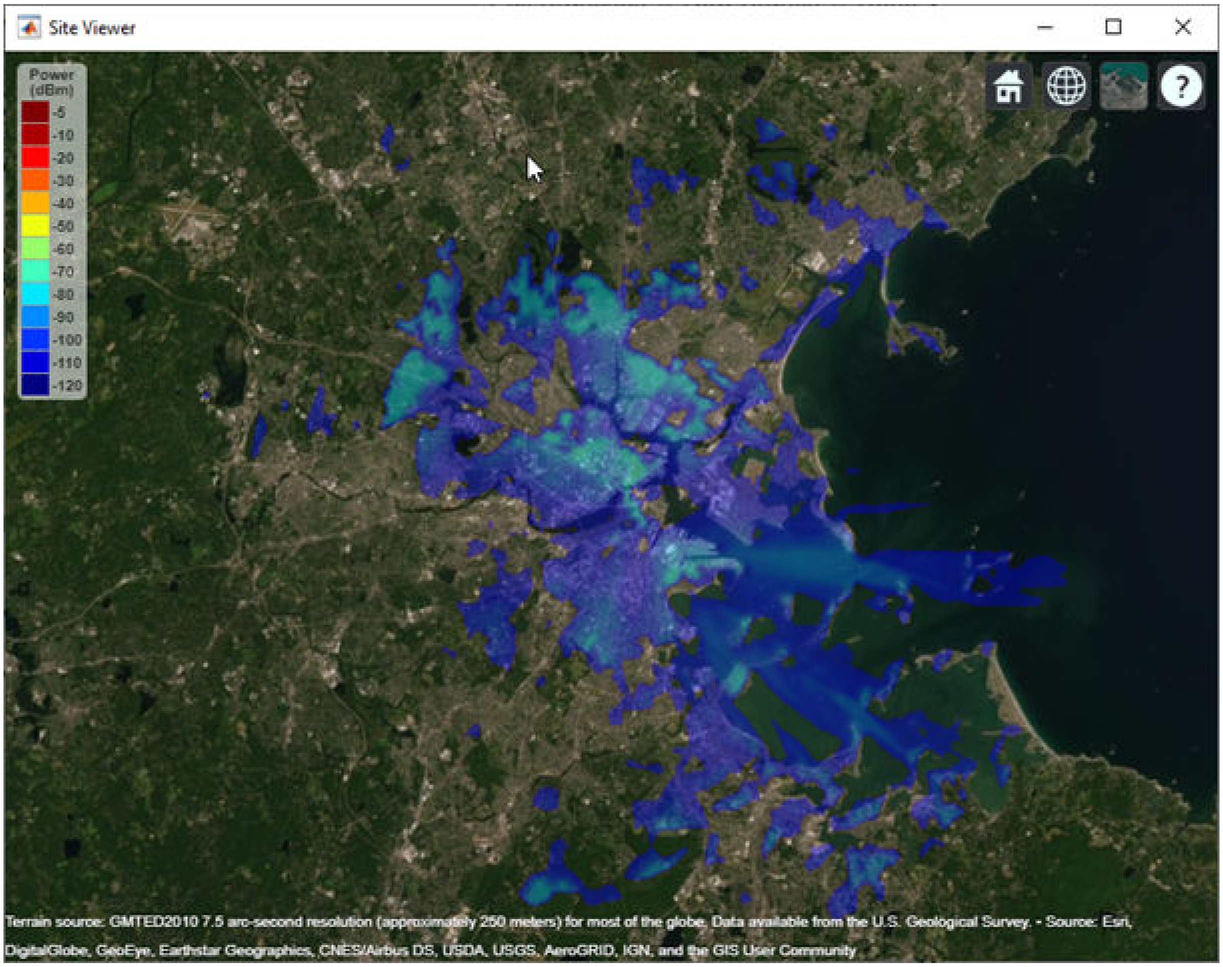
```
v1 = interp(pd1,lats,lons);
v2 = interp(pd2,lats,lons);
v3 = interp(pd3,lats,lons);
```

Create propagation data containing minimum received power values.

```
minReceivedPower = min([v1 v2 v3],[],2,"includenan");
pd = propagationData(lats,lons,"MinReceivedPower",minReceivedPower);
```

Plot minimum received power, which shows the weakest signal received from any transmitter site. The area shown may correspond to the service area of triangulation using the three transmitter sites.

```
sensitivity = -110;
contour(pd,"Levels",sensitivity:-5,"Type","power")
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

pd — Propagation data
propagationData object (default)

Propagation data, specified as a `propagationData` object.

Output Arguments

`datalocation` — Location coordinates of data points

M-by-2 matrix

Location of antenna site, returned as an *M*-by-2 matrix with each element unit in degrees. *M* is the number of rows in the data table with valid latitude and longitude values. Duplicate locations are not removed.

`lat` — Latitude of data points

M-by-1 vector

Latitude of data points, returned as an *M*-by-`1` vector with each element unit in degrees.

`lon` — Longitude of data points

M-by-1 vector

Longitude of data points, returned as an *M*-by-`1` matrix with each element unit in degrees. The output is wrapped so that the values are in the range `[-180 180]`.

See Also

Introduced in R2020a

propagationModel

Create RF propagation model

Syntax

```
pm = propagationModel(modelname)
pm = propagationModel( ___ ,Name,Value)
```

Description

`pm = propagationModel(modelname)` creates an RF propagation model for the specified model.

`pm = propagationModel( ___ ,Name,Value)` updates the model using one or more name-value pairs. For example, `pm = propagationModel('rain','RainRate',96)` creates a rain propagation model with a rain rate of 96 mm/h. Enclose each property name in quotes.

Examples

Signal Strength of Receiver in Heavy Rain

Specify transmitter and receiver sites.

```
tx = txsite('Name','MathWorks Apple Hill',...
       'Latitude',42.3001, ...
       'Longitude',-71.3504, ...
       'TransmitterFrequency', 2.5e9);

rx = rxsite('Name','Fenway Park',...
       'Latitude',42.3467, ...
       'Longitude',-71.0972);
```

Create the propagation model for a heavy rainfall rate.

```
pm = propagationModel('rain','RainRate',50)
```

```
pm = 
  Rain with properties:

    RainRate: 50
        Tilt: 0
```

Calculate the signal strength at the receiver using the rain propagation model.

```
ss = sigstrength(rx,tx,pm)
```

```
ss = -87.1559
```

Longley-Rice Propagation Model

Create a transmitter site.

```
tx = txsite
```

```
tx = 
  txsite with properties:

                   Name: 'Site 1'
               Latitude: 42.3001
              Longitude: -71.3504
                Antenna: 'isotropic'
           AntennaAngle: 0
          AntennaHeight: 10
             SystemLoss: 0
    TransmitterFrequency: 1.9000e+09
        TransmitterPower: 10
```

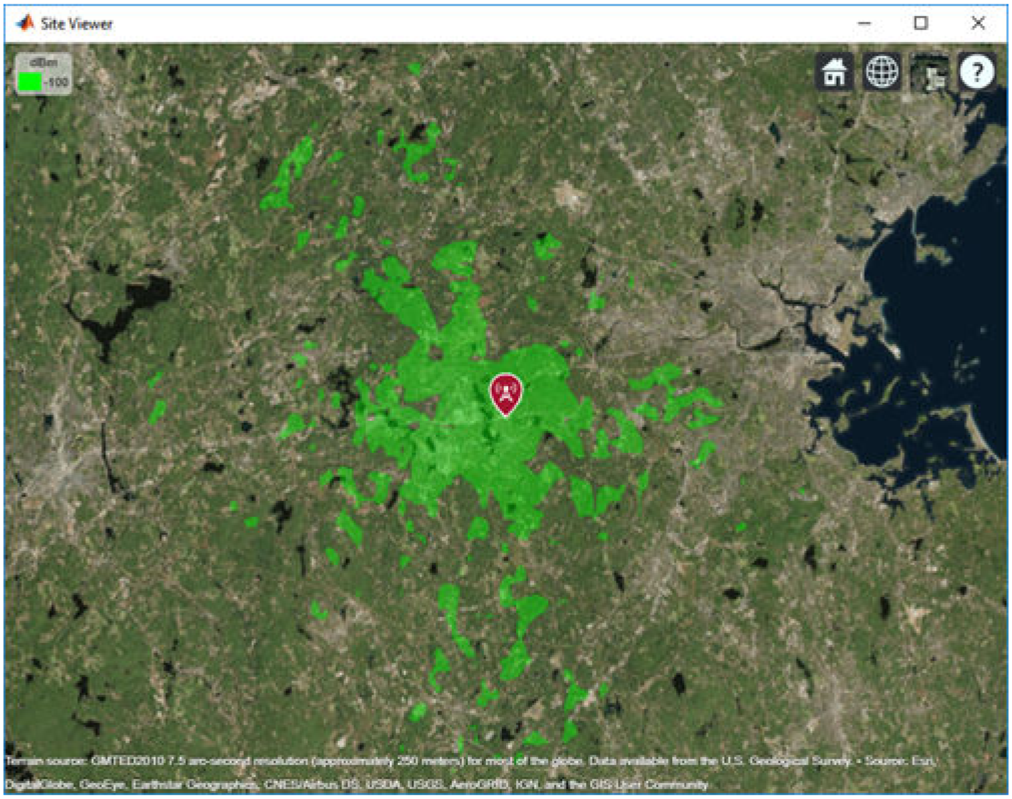
Create a Longley-Rice propagation model using the `propagationModel` function.

```
pm = propagationModel('longley-rice','TimeVariabilityTolerance',0.7)
```

```
pm = 
  LongleyRice with properties:

              AntennaPolarization: 'horizontal'
               GroundConductivity: 0.0050
               GroundPermittivity: 15
          AtmosphericRefractivity: 301
                      ClimateZone: 'continental-temperate'
         TimeVariabilityTolerance: 0.7000
    SituationVariabilityTolerance: 0.5000
```

Find the coverage of the transmitter site using the defined propagation model.

```
coverage(tx,'PropagationModel',pm)
```


Site Viewer
dBm
-100
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

`modelname` — Type of propagation model
`'freespace'` | `'rain'` | `'gas'` | `'fog'` | `'close-in'` |`'longley-rice'` | `'tirem'` | `'raytracing-image-method'`

Type of propagation model specified as one of these:

- `'freespace'` — Free space propagation model.
- `'rain'` — Rain propagation model. For more information, see [3].
- `'gas'` — Gas propagation model.
- `'fog'` — Fog propagation model. For more information, see [2] (Antenna Toolbox).
- `'close-in'` — Close-in propagation model typically used in urban macro-cell scenarios. For more information, see [1].

  Note The close-in model implements a statistical path loss model and can be configured for different scenarios. The default values correspond to an urban macro-cell scenario in a non-line-of-sight (NLOS) environment.
- `'longley-rice'` — Longley-Rice propagation model. This model is also known as Irregular Terrain Model (ITM). You can use this model to calculate point-to-point path loss between sites over an irregular terrain, including buildings. Path loss is calculated from free-space loss, terrain diffraction, ground reflection, refraction through atmosphere, tropospheric scatter, and atmospheric absorption. For more information and list of limitations, see [4].

Note The Longley-Rice model implements the point-to-point mode of the model, which uses terrain data to predict the loss between two points.

- `'tirem'` — Terrain Integrated Rough Earth Model™ (TIREM™). You can use this model to calculate point-to-point path loss between sites over an irregular terrain, including buildings. Path loss is calculated from free-space loss, terrain diffraction, ground reflection, refraction through atmosphere, tropospheric scatter, and atmospheric absorption. This model needs access to an external TIREM library. The actual model is valid from 1 MHZ to 1000 GHz. But with Antenna Toolbox elements and arrays the frequency range is limited to 200 GHz.
- `'raytracing-image-method'` — The ray tracing propagation model is a multipath propagation model that uses ray tracing analysis to compute propagation paths and their corresponding path losses. Path loss is calculated from free-space loss, reflection loss due to material, and antenna polarization loss. The ray tracing analysis uses the method of images, which includes surface reflections but does not include effects from refraction, diffraction, or scattering. This model is valid for a frequency range of 100 MHz to 100 GHz.

You can use these functions on RF propagation models:

- `range` — Calculate the range of the radio wave under different propagation scenarios. The`range` function does not support Longley-Rice, TIREM or `'raytracing-image-method'` propagation models.
- `pathloss` — Calculate the path loss of radio wave propagation between the transmitter and receiver sites under different propagation scenarios.
- `add` — Add propagation models.

Dependencies

To specify `'tirem'`, requires the Antenna Toolbox.

Data Types: `char`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RainRate',50`. Sets the rate of rainfall in the rain propagation model to 40.

Rain

`RainRate` — Rain rate

`16` (default) | nonnegative scalar

Rain rate, specified as a nonnegative scalar in millimeters per hour (mm/h).

Dependencies

To specify `'RainRate'`, you must specify `'rain'` propagation model.

Data Types: `double`

`Tilt` — Polarization tilt angle of signal

`0` (default) | scalar

Polarization tilt angle of the signal, specified as a scalar in degrees.

Dependencies

To specify `'Tilt'`, you must specify `'rain'` propagation model.

Data Types: `double`

Gas

`Temperature` — Air temperature

`15` (default) | scalar

Air temperature, specified as a scalar in Celsius (C).

Dependencies

To specify `'Temperature'`, you must specify `'gas'` propagation model.

Data Types: `double`

`AirPressure` — Dry air pressure

`101300` (default) | scalar

Dry air pressure, specified as a scalar in pascals (Pa).

Dependencies

To specify `'AirPressure'`, you must specify `'gas'` propagation model.

Data Types: `double`

`WaterDensity` — Water vapor density

`7.5` (default) | scalar

Water vapor density, specified as a scalar in grams per cubic meter (g/m^3).

Dependencies

To specify `'WaterDensity'`, you must specify `'gas'` propagation model.

Data Types: `double`

Fog

`Temperature` — Air temperature

`15` (default) | scalar

Air temperature, specified as a scalar in Celsius (C).

Dependencies

To specify `'Temperature'`, you must specify `'fog'` propagation model.

Data Types: `double`

`WaterDensity` — Liquid water density

`0.5` (default) | scalar

Liquid water density, specified as a scalar in grams per cubic meter (g/m^3).

Dependencies

To specify `'WaterDensity'`, you must specify `'fog'` propagation model.

Data Types: `double`

Close-In

`ReferenceDistance` — Free-space reference distance

`1` (default) | scalar

Free-space reference distance, specified as a scalar in meters.

Dependencies

To specify `'ReferenceDistance'`, you must specify the `'close-in'` propagation model.

Data Types: `double`

`PathLossExponent` — Path loss exponent

`2.9` (default) | scalar

Path loss exponent, specified as a scalar.

Dependencies

To specify `'PathLossExponent'`, you must specify `'close-in'` propagation model.

Data Types: `double`

`Sigma` — Standard deviation

`5.7` (default) | scalar

Standard deviation of the zero-mean Gaussian random variable, specified as a scalar in decibels (dB).

Dependencies

To specify `'Sigma'`, you must specify `'close-in'` propagation model.

Data Types: `double`

`NumDataPoints` — Number of data points

`1869` (default) | integer

Number of data points of zero-mean Gaussian random variable, specified as an integer.

Dependencies

To specify `'NumDataPoints'`, you must specify `'close-in'` propagation model.

Data Types: `double`

Note The close-in model is valid for distances greater than or equal to the `'ReferenceDistance'` property. If a distance less than the `'ReferenceDistance'` is used, path loss is `0`.

Longley-Rice

`AntennaPolarization` — Polarization of transmitter and receiver antennas

`'horizontal'` (default) | `'vertical'`

Polarization of transmitter and receiver antennas, specified as `'horizontal'` or `'vertical'`. Both antennas are assumed to have the same polarization. This value is used to calculate path loss due to ground reflection.

Dependencies

To specify `'AntennaPolarization'`, you must specify `'longley-rice'` propagation model.

Data Types: `char` | `string`

`GroundConductivity` — Conductivity of ground

`0.005` (default) | scalar

Conductivity of the ground, specified as a scalar in Siemens per meter (S/m). This value is used to calculate path loss due to ground reflection. The default value corresponds to average ground.

Dependencies

To specify `'GroundConductivity'`, you must specify `'longley-rice'` propagation model.

Data Types: `double`

`GroundPermittivity` — Relative permittivity of ground

`15` (default) | scalar

Relative permittivity of the ground, specified as a scalar. Relative permittivity is expressed as a ratio of absolute material permittivity to the permittivity of vacuum. This value is used to calculate the path loss due to ground reflection. The default value corresponds to average ground.

Dependencies

To specify `'GroundPermittivity'`, you must specify `'longley-rice'` propagation model.

Data Types: `double`

`AtmosphericRefractivity` — Atmospheric refractivity near ground

`301` (default) | scalar

Atmospheric refractivity near the ground, specified as a scalar in N-units. This value is used to calculate the path loss due to refraction through the atmosphere and tropospheric scatter. The default value corresponds to average atmospheric conditions.

Dependencies

To specify `'AtmosphericRefractivity'`, you must specify `'longley-rice'` propagation model.

Data Types: `double`

`ClimateZone` — Radio climate zone

`'continental-temperate'` (default) | `'equatorial'` | `'continental-subtropical'` | `'maritime-subtropical'` | `'desert'` | `'maritime-over-land'` | `'maritime-over-sea'`

Radio climate zone. This value is used to calculate the variability due to changing atmospheric conditions. The default value corresponds to average atmospheric conditions in a particular climate zone.

Dependencies

To specify `'ClimateZone'`, you must specify `'longley-rice'` propagation model.

Data Types: `char` | `string`

`TimeVariabilityTolerance` — Time variability tolerance level

`0.5` (default) | scalar

Time variability tolerance level of the path loss, specified as a scalar between [0.001, 0.999]. Time variability occurs due to changing atmospheric conditions. This value gives the required system reliability or the fraction of time during which the actual path loss is expected to be less than or equal to model prediction. For more information, see [5].

Dependencies

To specify `'TimeVariabilityTolerance'`, you must specify `'longley-rice'` propagation model.

Data Types: `double`

`SituationVariabilityTolerance` — Situation variability tolerance level

`0.5` (default) | scalar

Situation variability tolerance level of the path loss, specified as a scalar in between [0.001, 0.999]. Situation variability occurs due to uncontrolled or hidden random variables. This value gives the required system confidence or the fraction of similar situations for which the actual path loss is expected to be less than or equal to the model prediction. For more information, see [5].

Dependencies

To specify `'SituationVariabilityTolerance'`, you must specify `'longley-rice'` propagation model.

Data Types: `double`

TIREM

`AntennaPolarization` — Polarization of transmitter and receiver antennas

`'horizontal'` (default) | `'vertical'`

Polarization of transmitter and receiver antennas, specified as `'horizontal'` or `'vertical'`. Both antennas are assumed to have the same polarization. This value is used to calculate path loss due to ground reflection.

Dependencies

To specify `'AntennaPolarization'`, you must specify `'tirem'` propagation model.

Data Types: `char` | `string`

`GroundConductivity` — Conductivity of ground

`0.005` (default) | numeric scalar

Conductivity of the ground, specified as a numeric scalar in Siemens per meter (S/m) in the range of 0.0005 to 100. This value is used to calculate path loss due to ground reflection. The default value corresponds to average ground.

Dependencies

To specify `'GroundConductivity'`, you must specify `'tirem'` propagation model.

Data Types: `double`

`GroundPermittivity` — Relative permittivity of ground
`15` (default) | numeric scalar

Relative permittivity of the ground, specified as a numeric scalar in the range of 1 to 100. Relative permittivity is expressed as a ratio of absolute material permittivity to the permittivity of vacuum. This value is used to calculate the path loss due to ground reflection. The default value corresponds to average ground.

Dependencies

To specify `'GroundPermittivity'`, you must specify `'tirem'` propagation model.

Data Types: `double`

`AtmosphericRefractivity` — Atmospheric refractivity near ground
`301` (default) | scalar

Atmospheric refractivity near the ground, specified as a numeric scalar in N-units in the range of 250 to 400. This value is used to calculate the path loss due to refraction through the atmosphere and tropospheric scatter. The default value corresponds to average atmospheric conditions.

Dependencies

To specify `'AtmosphericRefractivity'`, you must specify `'tirem'` propagation model.

Data Types: `double`

`Humidity` — Absolute air humidity near ground
`'9'` (default) | numeric scalar

Absolute air humidity near ground,specified as a numeric scalar in `g/m^3` units in the range of 0 to 110. You can use this value to calculate path loss due to atmospheric absorption. The default value corresponds to the absolute humidity of air at 15 degrees Celsius and 70 percent relative humidity.

Dependencies

To specify `'Humidity'`, you must specify `'tirem'` propagation model.

Data Types: `double`

raytracing-image-method

`MaxNumReflections` — Maximum number of path reflections
`1` (default) | `0` | `2`

Maximum number of reflections in the propagation paths to search for using ray tracing, specified as `0`, `1`, or `2`. The default value results in a search for a line-of-sight propagation path along with propagation paths that each contain a single reflection.

Dependencies

To specify `'MaxNumReflections'`, you must specify `'raytracing-image-method'` propagation model.

Data Types: `double`

`CoordinateSystem` — Coordinate system of map and site location
`'geographic'` (default) | `'cartesian'`

Coordinate system of the site location, specified as `'geographic'` or `'cartesian'`. If you specify `'geographic'`, material types are defined using `'BuildingMaterial'` or `'TerrainMaterial'` properties. If you specify `'cartesian'`, material types are defined using the `'SurfaceMaterial'` properties.

Data Types: `string` | `char`

`BuildingsMaterial` — Surface material of geographic buildings

`'concrete'` (default) | `'perfect-reflector'` | `'brick'` | `'wood'` | `'glass'` | `'metal'` | `'custom'`

Surface material of geographic buildings, specified as one of these: `'perfect-reflector'`, `'concrete'`, `'brick'`, `'wood'`, `'glass'`, `'metal'`, or `'custom'`. The material type is used to calculate reflection loss where propagation paths reflect off of building surfaces. For more information, see “ITU Permittivity and Conductivity Values for Common Materials” on page 4-173.

When `'BuildingsMaterial'` is set to `'custom'`, the material permittivity and conductivity are specified in the `BuildingsMaterialPermittivity` and `BuildingsMaterialConductivity` properties.

Dependencies

To specify `'BuildingsMaterials'`, you must set `'CoordinateSystem'` to `'geographic'`.

Data Types: `char` | `string`

`BuildingsMaterialPermittivity` — Relative permittivity of buildings surface materials

`5.31` (default) | nonnegative scalar

Relative permittivity of the buildings surface material, specified as a nonnegative scalar. Relative permittivity is expressed as a ratio of absolute material permittivity to the permittivity of vacuum. This value is used to calculate path loss due to reflection. The default value corresponds to concrete at 1.9 GHz.

Dependencies

To specify `'BuildingsMaterialPermittivity'`, you must set `'CoordinateSystem'` to `'geographic'` and `'BuildingsMaterial'` to `'custom'`.

Data Types: `double`

`BuildingsMaterialConductivity` — Conductivity of buildings surface materials

`0.0548` (default) | nonnegative scalar

Conductivity of the buildings surface material, specified as a nonnegative scalar in Siemens per meter (S/m). This value is used to calculate path loss due to reflection. The default value corresponds to concrete at 1.9 GHz.

Dependencies

To specify `'BuildingsMaterialConductivity'`, you must set `'CoordinateSystem'` to `'geographic'` and `'BuildingsMaterial'` to `'custom'`.

Data Types: `double`

`TerrainMaterial` — Surface material of geographic terrain

`'concrete'` (default) | `'perfect-reflector'` | `'brick'` | `'water'` | `'vegetation'` | `'loam'` | `'custom'`

Surface material of terrain, specified as one of these: `'perfect-reflector'`, `'concrete'`, `'brick'`, `'water'`, `'vegetation'`, `'loam'`, or `'custom'`. The material type is used to calculate reflection loss where propagation paths reflect off of terrain surfaces. For more information, see “ITU Permittivity and Conductivity Values for Common Materials” on page 4-173.

When `'TerrainMaterial'` is set to `'custom'`, the material permittivity and conductivity are specified in the `'TerrainMaterialPermittivity'` and `'TerrainMaterialConductivity'` properties.

Dependencies

To specify `'TerrainMaterial'`, you must set `'CoordinateSystem'` to `'geographic'`.

Data Types: `char` | `string`

`TerrainMaterialPermittivity` — Relative permittivity of terrain materials

`5.31` (default) | nonnegative scalar

Relative permittivity of the terrain material, specified as a nonnegative scalar. Relative permittivity is expressed as a ratio of absolute material permittivity to the permittivity of vacuum. This value is used to calculate path loss due to reflection. The default value corresponds to concrete at 1.9 GHz.

Dependencies

To specify `'TerrainMaterialPermittivity'`, you must set `'CoordinateSystem'` to `'geographic'` and `'TerrainMaterial'` to `'custom'`.

Data Types: `double`

`TerrainMaterialConductivity` — Conductivity of terrain materials

`0.0548` (default) | nonnegative scalar

Conductivity of the terrain material, specified as a nonnegative scalar in Siemens per meter (S/m). This value is used to calculate path loss due to reflection. The default value corresponds to concrete at 1.9 GHz.

Dependencies

To specify `'TerrainMaterialConductivity '`, you must set `'CoordinateSystem'` to `'geographic'` and set `'TerrainMaterial'` to `'custom'`.

Data Types: `double`

`SurfaceMaterial` — Surface material of Cartesian map surface

`'plasterboard'` (default) | `'perfect-reflector'` | `'ceilingboard'` | `'chipboard'` | `'floorboard'` | `'concrete'` | `'brick'` | `'wood'` | `'glass'` | `'metal'` | `'water'` | `'vegetation'` | `'loam'` | `'custom'`

Surface material of Cartesian map surface, specified as one of these: `'plasterboard'`,`'perfect-reflector'`, `'ceilingboard'`, `'chipboard'`, `'floorboard'`, `'concrete'`, `'brick'`, wood, `'glass'`, `'metal'`, `'water'`, `'vegetation'`, `'loam'`, or `'custom'`. The material type is used to calculate reflection loss where propagation paths reflect off of surfaces. For more information, see “ITU Permittivity and Conductivity Values for Common Materials” on page 4-173.

When `'SurfaceMaterial'` is set to `'custom'`, the material permittivity and conductivity are specified in the `'SurfaceMaterialPermittivity'` and `'SurfaceMaterialConductivity'` properties.

Dependencies

To specify `'SurfaceMaterial'`, you must set `'CoordinateSystem'` to `'cartesian'`.

Data Types: `char` | `string`

`SurfaceMaterialPermittivity` — Relative permittivity of surface materials

`2.94` (default) | nonnegative scalar

Relative permittivity of the surface material, specified as a nonnegative scalar. Relative permittivity is expressed as a ratio of absolute material permittivity to the permittivity of vacuum. This value is used to calculate path loss due to reflection. The default value corresponds to plaster board at 1.9 GHz.

Dependencies

To specify `'SurfaceMaterialPermittivity'`, you must set `'CoordinateSystem'` to `'cartesian'` and `'SurfaceMaterial'` to `'custom'`.

Data Types: `double`

`SurfaceMaterialConductivity` — Conductivity of surface materials

`0.0183` (default) | nonnegative scalar

Conductivity of the surface material, specified as a nonnegative scalar in Siemens per meter (S/m). This value is used to calculate path loss due to reflection. The default value corresponds to plaster board at 1.9 GHz.

Dependencies

To specify `'SurfaceMaterialConductivity '`, you must set `'CoordinateSystem'` to `'cartesian'` and set `'SurfaceMaterial'` to `'custom'`.

Data Types: `double`

More About

N-Units

The refractive index of air *n* is related to the dielectric constants of the gas constituents of an air mixture. The numerical value of *n* is only slightly larger than one. To make the calculation more convenient, you can use *N* units, which are given by the formula: $N = (n - 1) \times 10^6$

ITU Permittivity and Conductivity Values for Common Materials

ITU-R P.2040-1 [8] and ITU-R P.527-5 [9] present methods, equations, and values used to calculate real relative permittivity, conductivity, and complex relative permittivity for the common materials.

- For information about the values computed for building materials specified in ITU-R P.2040-1, see `buildingMaterialPermittivity`.
- For information about the values computed for terrain materials specified in ITU-R P.527-5, see `earthSurfacePermittivity`.

References

[1] Sun, S.,Rapport, T.S., Thomas, T., Ghosh, A., Nguyen, H., Kovacs, I., Rodriguez, I., Koymen, O.,and Prartyka, A. "Investigation of prediction accuracy, sensitivity, and parameter stability of large-

scale propagation path loss models for 5G wireless communications." *IEEE Transactions on Vehicular Technology*, Vol.65, No 5, pp 2843-2860, May 2016.

[2] ITU-R P.840-6. "Attenuation due to cloud and fog." *Radiocommunication Sector of ITU*

[3] ITU-R P.838-3. "Specific attenuation model for rain for use in prediction methods." *Radiocommunication Sector of ITU*

[4] Hufford, George A., Anita G. Longley, and William A.Kissick. "A Guide to the Use of the ITS Irregular Terrain Model in the Area Prediction Mode." *NTIA Report 82-100*. Pg-7.

[5] *SoftWright Homepage* https://www.softwright.com/faq/support/longley_rice_variability.html

[6] Seybold, John. *Introduction to RF Propagation*. Wiley, 2005

[7] ITU-R P.676-11. "Attenuation by atmospheric gases." *Radiocommunication Sector of ITU*

[8] ITU-R P.2040-1. "Effects of Building Materials and Structures on Radiowave Propagation Above 100MHz." *International Telecommunications Union - Radiocommunications Sector (ITU-R)*. July 2015.

[9] ITU-R P.527-5. "Electrical characteristics of the surface of the Earth." *International Telecommunications Union - Radiocommunications Sector (ITU-R)*. August 2019.

See Also

`coverage` | `link` | `los` | `pathloss` | `range` | `rangeangle` | `sigstrength` | `sinr`

Topics

“Access TIREM Software” (Antenna Toolbox)

Introduced in R2019b

range

Range of radio wave propagation

Syntax

```
r = range(propmodel,tx,pl)
```

Description

`r = range(propmodel,tx,pl)`returns the range of radio wave propagation from the transmitter site.

Examples

Range of Transmitter In Heavy Rain

Specify transmitter and receiver sites.

```
tx = txsite('Name','MathWorks Apple Hill',...
       'Latitude',42.3001, ...
       'Longitude',-71.3504, ...
       'TransmitterFrequency', 2.5e9);

rx = rxsite('Name','Fenway Park',...
       'Latitude',42.3467, ...
       'Longitude',-71.0972);
```

Create the propagation model for heavy rainfall rate.

```
pm = propagationModel('rain','RainRate',50)
```

```
pm = 
  Rain with properties:

    RainRate: 50
        Tilt: 0
```

Calculate the range of transmitter using the rain propagation model and a path loss of 127 dB.

```
r = range(pm,tx,127)
```

```
r = 2.0747e+04
```

Input Arguments

`propmodel` — Propagation model

character vector or string

Propagation model, specified as a character vector or string.

Data Types: `char`

`tx` — Transmitter site
`txsite` object

Transmitter site, specified as a `txsite` object. You can use array inputs to specify multiple sites.

Data Types: `char`

`pl` — Path loss
scalar

Path loss, specified as a scalar in decibels.

Data Types: `double`

Output Arguments

`r` — range
scalar | *M*-by-1 arrays

Range, returned as a scalar or *M*-by-1 array with each element in meters. *M* is the number of TX sites.

Range is the maximum distance for which the path loss does not exceed the value of specified `pl`.

See Also

`pathloss` | `propagationModel`

Introduced in R2019b

raytrace

Plot propagation paths between sites

Syntax

```
raytrace(tx,rx)
raytrace(tx,rx,propmodel)
raytrace( ___ ,Name,Value)
rays = raytrace( ___ )
```

Description

`raytrace(tx,rx)` plots the propagation paths from the transmitter site (`tx`) to the receiver site (`rx`). The propagation paths are found using ray tracing with surface geometry defined by the `Map` property. Each propagation path is color-coded according to the received power (dBm) or path loss (dB) along the path, assuming unpolarized rays.

Note

- The ray tracing analysis includes surface reflections but does not include effects from refraction, diffraction, or scattering.
- Operational frequency for this function is from 100 MHz to 100 GHz.

`raytrace(tx,rx,propmodel)` plots the propagation paths from the transmitter site (`tx`) to the receiver site (`rx`) based on the specified propagation model. To input building and terrain materials to calculate path loss, please use the `'raytracing-image-method'` propagation model and set the properties to specify building materials.

`raytrace( ___ ,Name,Value)` plots propagation paths with additional options specified by one or more name-value pairs.

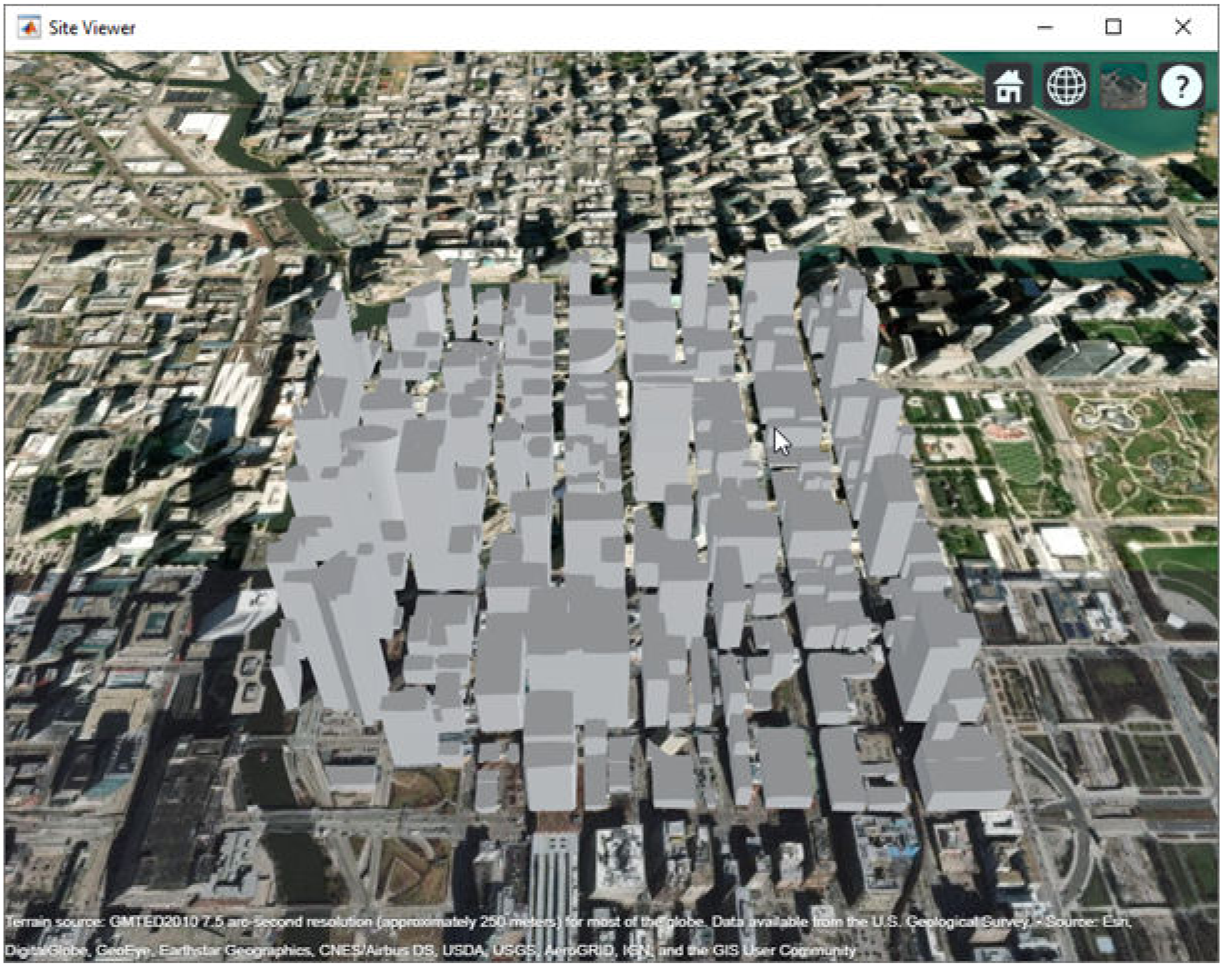
`rays = raytrace( ___ )` returns the propagation paths in `rays`.

Examples

Signal Strength Using Ray Tracing Image Method Propagation Model

Launch Site Viewer with buildings in Chicago. For more information about the osm file, see [1] on page 4-0 .

```
viewer = siteviewer("Buildings","chicago.osm");
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create transmitter site on a building.

```
tx = txsite('Latitude',41.8800, ...
    'Longitude',-87.6295, ...
    'TransmitterFrequency',2.5e9);
```

Create receiver site near another building.

```
rx = rxsite('Latitude',41.881352, ...
    'Longitude',-87.629771, ...
    'AntennaHeight',30);
```

Compute signal strength using ray tracing propagation model and default single-reflection analysis.

```
pm = propagationModel("raytracing-image-method");
ssOneReflection = sigstrength(rx,tx,pm)
```

```
ssOneReflection = -54.0915
```

Compute signal strength with analysis up to two reflections, where total received power is the cumulative power of all propagation paths

```
pm.MaxNumReflections = 2;
ssTwoReflections = sigstrength(rx,tx,pm)
ssTwoReflections = -52.3890
```

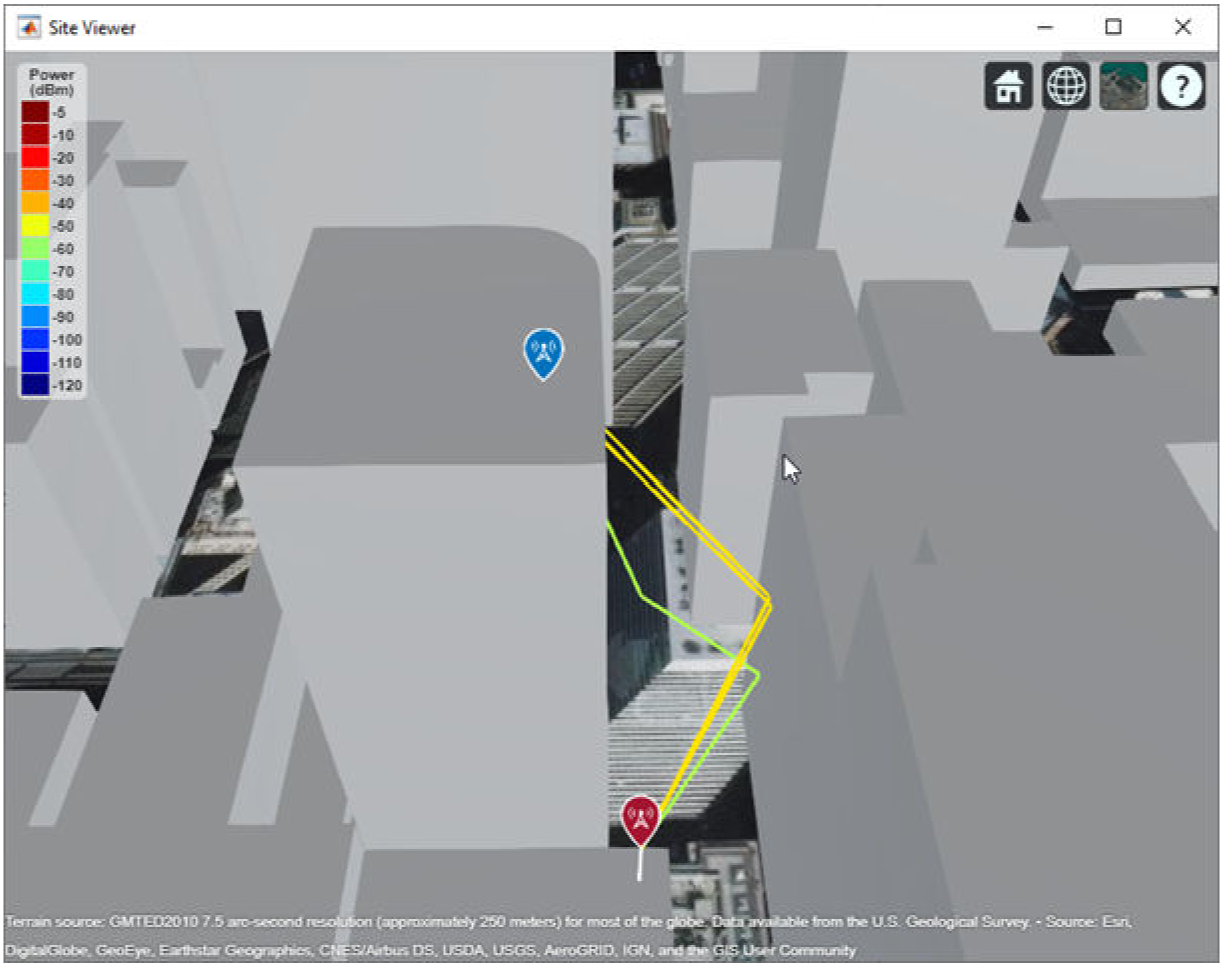
Observe effect of material by replacing default concrete material with perfect reflector.

```
pm.BuildingsMaterial = 'perfect-reflector';
ssPerfect = sigstrength(rx,tx,pm)
```

```
ssPerfect = -41.9927
```

Plot propagation paths.

```
raytrace(tx, rx, pm)
```


Site Viewer
Power (dBm)
-5
-10
-20
-30
-40
-50
-60
-70
-80
-90
-100
-110
-120
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

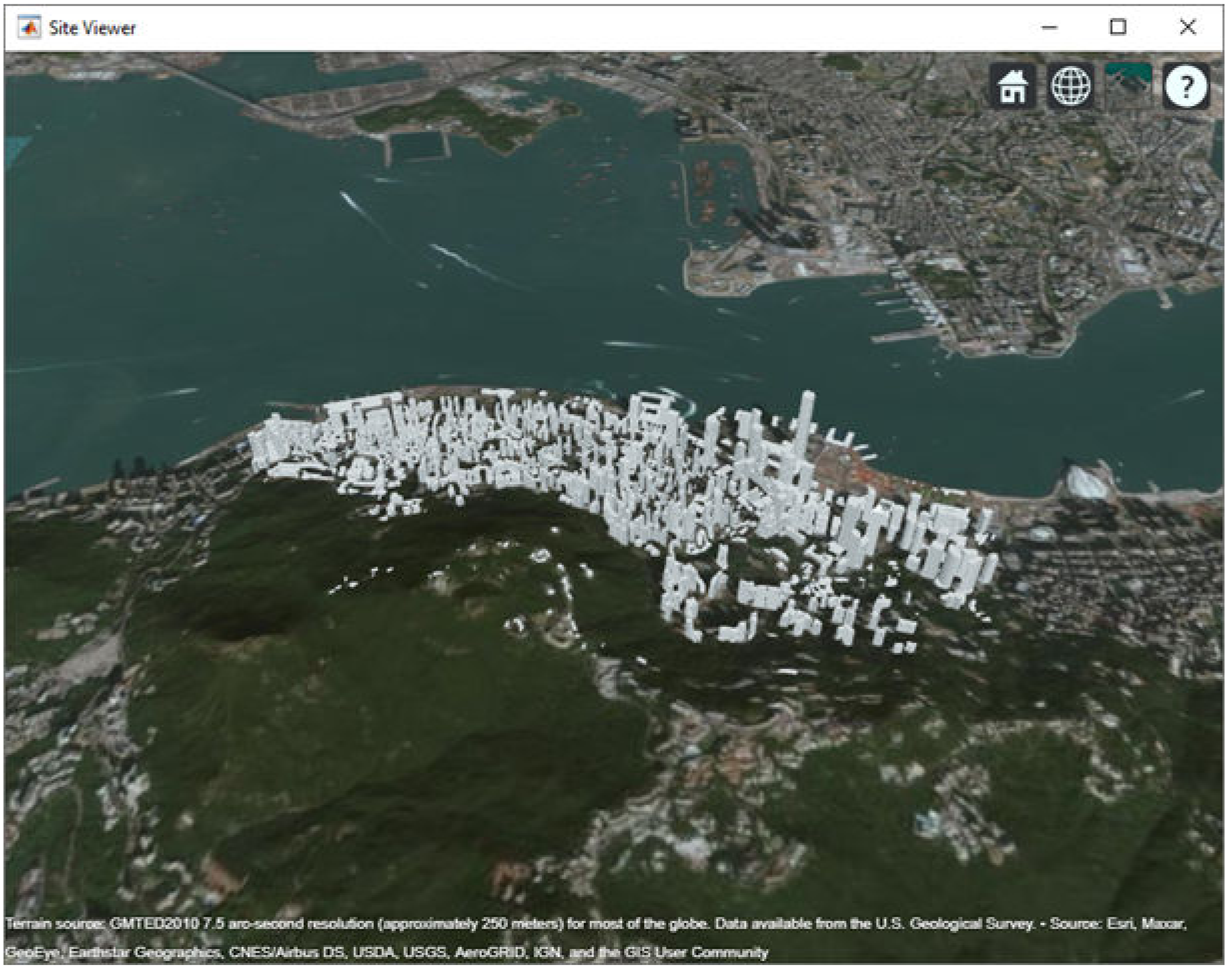
Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Path Loss Due to Material Reflection and Atmosphere

Launch Site Viewer with buildings in Hong Kong. For more information about the osm file, see [1] on page 4-0 .

```
viewer = siteviewer("Buildings","hongkong.osm");
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Define transmitter and receiver sites to model a small cell scenario in a dense urban environment.

```
tx = txsite("Name","Small cell transmitter", ...
       "Latitude",22.2789, ...
       "Longitude",114.1625, ...
       "AntennaHeight",10, ...
       "TransmitterPower",5, ...
       "TransmitterFrequency",28e9);
rx = rxsite("Name","Small cell receiver", ...
       "Latitude",22.2799, ...
       "Longitude",114.1617, ...
       "AntennaHeight",1);
```

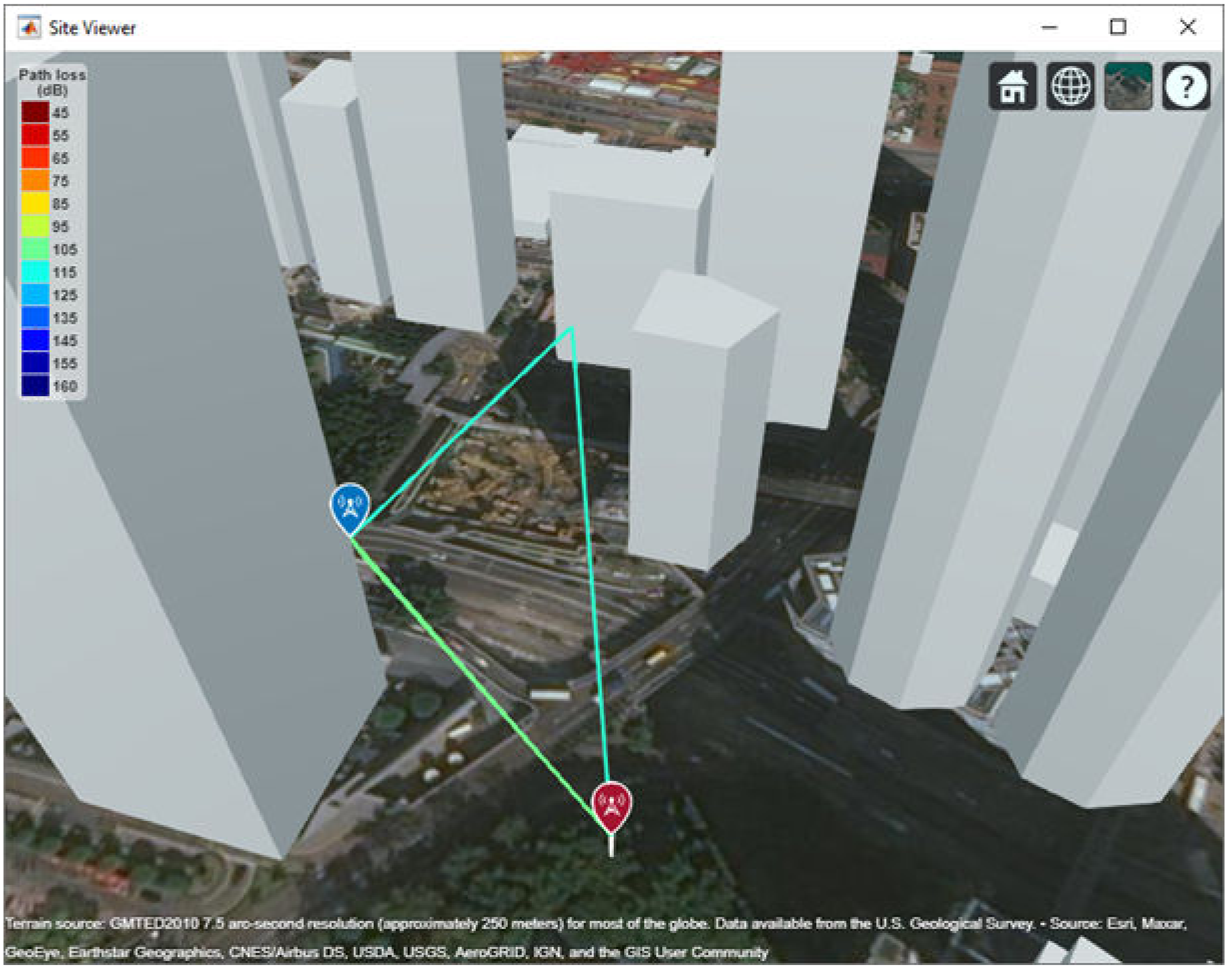
Create ray tracing propagation model for perfect reflection.

```
pm = propagationModel("raytracing-image-method", ...
       "BuildingsMaterial","perfect-reflector", ...
       "TerrainMaterial","perfect-reflector");
```

Visualize propagation paths and compute corresponding path losses.

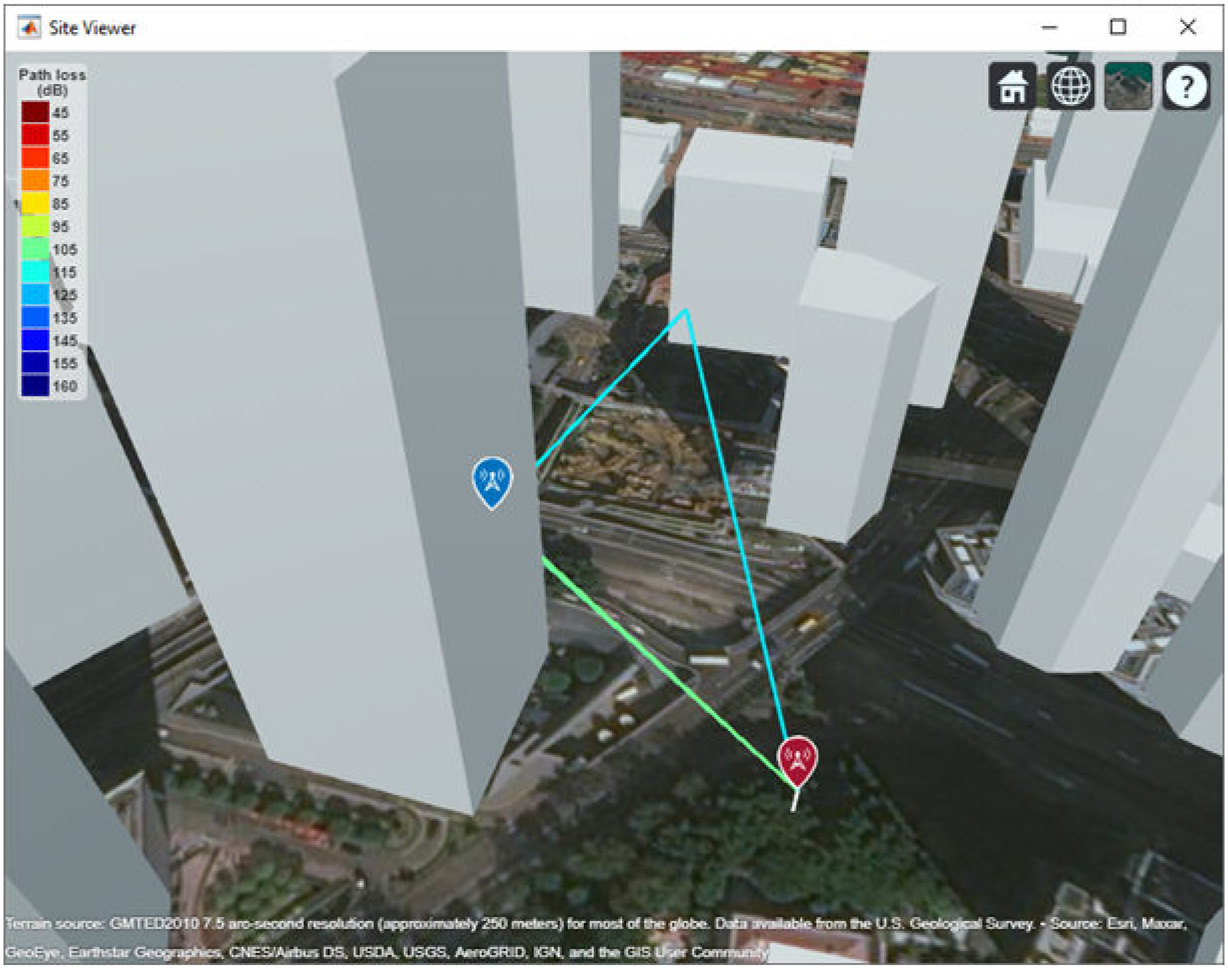
```
raytrace(tx,rx,pm,"Type","pathloss")
raysPerfect = raytrace(tx,rx,pm,"Type","pathloss");
plPerfect = [raysPerfect{1}.PathLoss]
```

```
plPerfect = 1×3

  104.2656  104.2745  112.0095
```


Site Viewer
Path loss (dB)
45
55
65
75
85
95
105
115
125
135
145
155
160
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Re-compute with material reflection loss by setting material type on the propagation model. The first value is unchanged because it corresponds to the line-of-sight propagation path.

```
pm.BuildingsMaterial = "glass";
pm.TerrainMaterial = "concrete";
raytrace(tx,rx,pm,"Type","pathloss")
raysMtrls = raytrace(tx,rx,pm,"Type","pathloss");
plMtrls = [raysMtrls{1}.PathLoss]
plMtrls = 1×3

  104.2656  106.2545  119.3577
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, Maxar, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Ray Tracing In Conference Room

Define a 3-D map for a conference room with one table and four chairs.

```
mapFileName = "conferenceroom.stl";
```

Visualize the 3-D map.

```
figure; view(3);
trisurf(stlread(mapFileName), 'FaceAlpha', 0.3, 'EdgeColor', 'none');
```

```
hold on; axis equal; grid off;
xlabel('x'); ylabel('y'); zlabel('z');
```

Define a transmitter site close to the wall and a receiver site under the table.

```
    tx = txsite("cartesian",  "AntennaPosition", [-1.45; -1.45; 1.5],"TransmitterFrequency", 2.8e
    rx = rxsite("cartesian","AntennaPosition", [.3; .2; .5]);
```

Plot the transmitter site in red and receiver site in blue.

```
scatter3(tx.AntennaPosition(1,:), tx.AntennaPosition(2,:), tx.AntennaPosition(3,:), 'sr', 'filled
scatter3(rx.AntennaPosition(1,:), rx.AntennaPosition(2,:),rx.AntennaPosition(3,:), 'sb', 'filled
```

Create a ray tracing propagation model for Cartesian coordinates and set the surface material to wood.

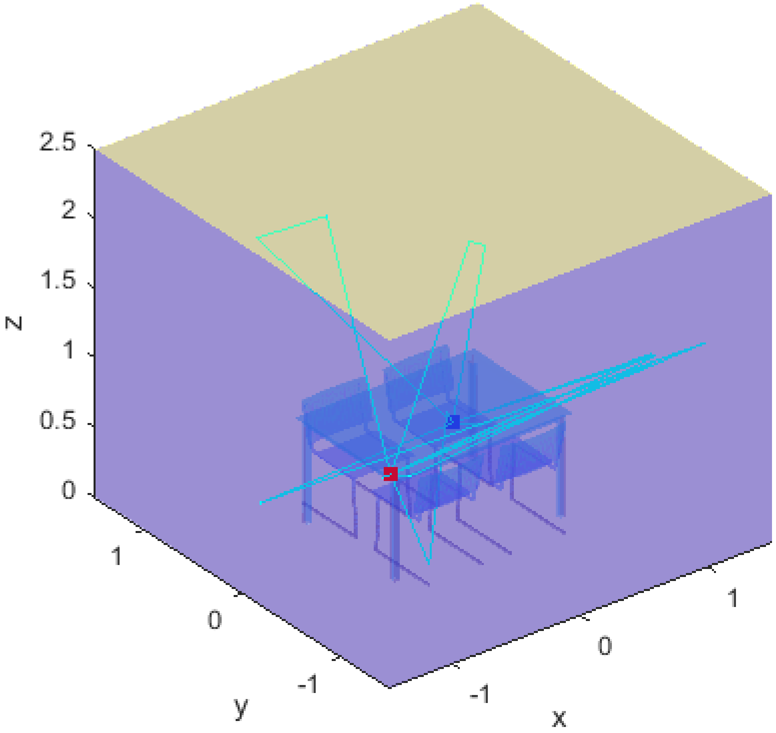
```
pm = propagationModel("raytracing-image-method", "CoordinateSystem", "cartesian", ...
        "SurfaceMaterial", "wood", "MaxNumReflections", 2);
```

Perform ray tracing and save the computed rays using comm.Ray object

```
rays = raytrace(tx, rx, pm, 'Map', mapFileName);
rays = rays{1};
```

Visualize rays in the 3D map.

```
    for i = 1:length(rays)
        if rays(i).LineOfSight
            propPath = [rays(i).TransmitterLocation, ...
                        rays(i).ReceiverLocation];
        else
            propPath = [rays(i).TransmitterLocation, ...
                    rays(i).ReflectionLocations, ...
                        rays(i).ReceiverLocation];
        end

        line(propPath(1,:), propPath(2,:), propPath(3,:), 'Color', 'cyan');
    end
```



Input Arguments

rx — Receiver site

rxsite object | array of rxsite objects

Receiver site, specified as a rxsite object or an array of rxsite objects. If the transmitter sites are specified as arrays, then the propagation paths are plotted from each transmitter to each receiver site.

tx — Transmitter site

txsite object | array of txsite objects

Transmitter site, specified as a txsite object or an array of txsite objects. If the receiver sites are specified as arrays, then the propagation paths are plotted from each transmitter to each receiver site.

propmodel — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can use the propagationModel function to define this input. The default propagation model is 'raytracing-image-method'.

You can also use the name-value pair 'PropagationModel' to specify this parameter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Type','power'`

Type — Type of quantity to plot

`'power'` (default) | `'pathloss'`

Type of quantity to plot, specified as the comma-separated pair consisting of `'Type'` and `'power'` in dBm or `'pathloss'` in dB.

When you specify `'power'`, each path is color-coded according to the received power along the path. When you specify `'pathloss'`, each path is color-coded according to the path loss along the path.

Friis equation is used to calculate the received power:

$$P_{rx} = P_{tx} + G_{tx} + G_{rx} - L - L_{tx} - L_{rx}$$

where:

- P_{rx} is the received power along the path.
- P_{tx} is the transmit power defined in tx.TransmitterPower.
- G_{tx} is the antenna gain of tx in the direction of the angle-of-departure (AoD).
- G_{rx} is the antenna gain of rx in the direction of the angle-of-arrival (AoA).
- L is the path loss calculated along the path.
- L_{tx} is the system loss of the transmitter defined in tx.SystemLoss.
- L_{rx} is the system loss of the receiver defined in rx.SystemLoss.

Data Types: `char`

`PropagationModel` — Type of propagation model for ray tracing analysis

`'raytracing-image-method'` (default) | ray tracing propagation model object

Type of propagation model for ray tracing analysis, specified as the comma-separated pair consisting of `'PropagationModel'` and `'raytracing-image-method'` or a ray tracing propagation model object created using `propagationModel`.

Data Types: `char`

`NumReflections` — Number of reflections to search for in propagation paths

`[0 1]` (default) | numeric row vector

Number of reflections to search for in propagation paths using ray tracing, specified as the comma-separated pair consisting of `'NumReflections'` and a numeric row vector whose elements are `0`, `1`, or `2`.

The default value results in the search for a line-of-sight propagation path along with propagation paths that each contain a single reflection.

Data Types: `double`

`Colormap` — Color map for coloring propagation paths

`'jet'` (default) | predefined color map name | *M*-by-3 array of RGB

Color map for coloring propagation paths, specified as the comma-separated pair consisting of `'Colormap'` and a predefined color map name or an *M*-by-3 array of RGB (red, blue, green) triplets that define *M* individual colors.

Data Types: `char` | `double`

`ColorLimits` — Color limits for colormap

two-element numeric row vector

Color limits for colormap, specified as the comma-separated pair consisting of `'ColorLimits'` and a two-element numeric row vector of the form [min max]. The units and default values of the color limits depend on the value of the `'Type'` parameter:

- `'power'`– Units are in dBm, and the default value is `[-120 -5]`.
- `'pathloss'`– Units are in dB, and the default value is `[45 160]`.

The color limits indicate the values that map to the first and last colors in the colormap. Propagation paths with values below the minimum color limit are not plotted.

Data Types: `double`

`ShowLegend` — Show color legend on map

`true` (default) | `false`

Show color legend on map, specified as the comma-separated pair consisting of `'ShowLegend'` and `true` or `false`.

Data Types: `logical`

`Map` — Map for visualization or surface data

`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
`'geographic'`	• `siteviewer`[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are `'none'`, `'gmted2010'`, or the name of the custom terrain data added using `addCustomTerrain`	• current siteviewer or new siteviewer if none are open. • `'gmted2010'` if called with an output.
`'cartesian'`	`'none'`, triangulation object or name of an STL file.	`'none'`

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: `char` | `string`

Output Arguments

`rays` — Ray configuration object

M-by-*N* cell array

Ray configuration, returned as a *M*-by-*N* cell array where *M* is the number of transmitter sites and *N* is the number of receiver sites. Each cell element is a row vector of objects representing all the rays found between the corresponding transmitter site and receiver site. array. Within each row vector, the `comm.Ray` objects are ordered by increasing number of reflections, and where number of reflections are equal they are ordered by increasing propagation distance.

See Also

`los` | `siteviewer`

Introduced in R2019b

removeCustomTerrain

Remove custom terrain data

Syntax

```
removeCustomTerrain(terrainName)
```

Description

`removeCustomTerrain(terrainName)` removes the custom terrain data specified by the user-defined `terrainName`. You can use this function to remove terrain data that is no longer needed. The terrain data to be removed must have been previously added using `addCustomTerrain`.

Examples

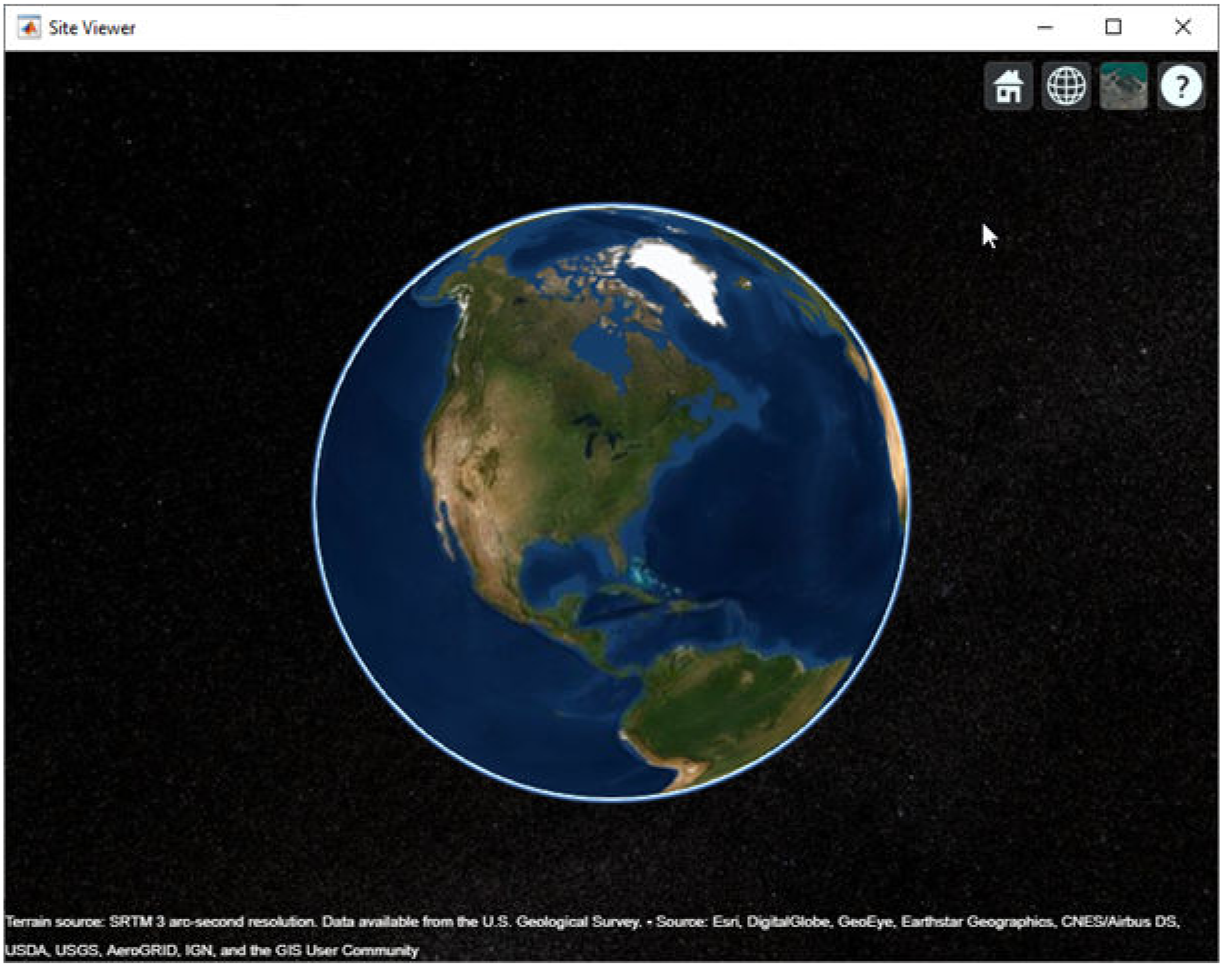
Site Viewer Maps Using Custom Terrain

Add terrain for a region around Boulder, CO. The DTED file was downloaded from the "SRTM Void Filled" data set available from the U.S. Geological Survey.

```
dtedfile = "n39_w106_3arc_v2.dt1";
attribution = "SRTM 3 arc-second resolution. Data available " + ...
   "from the U.S. Geological Survey.";
addCustomTerrain("southboulder",dtedfile,"Attribution",attribution)
```

Use the custom terrain name in Site Viewer.

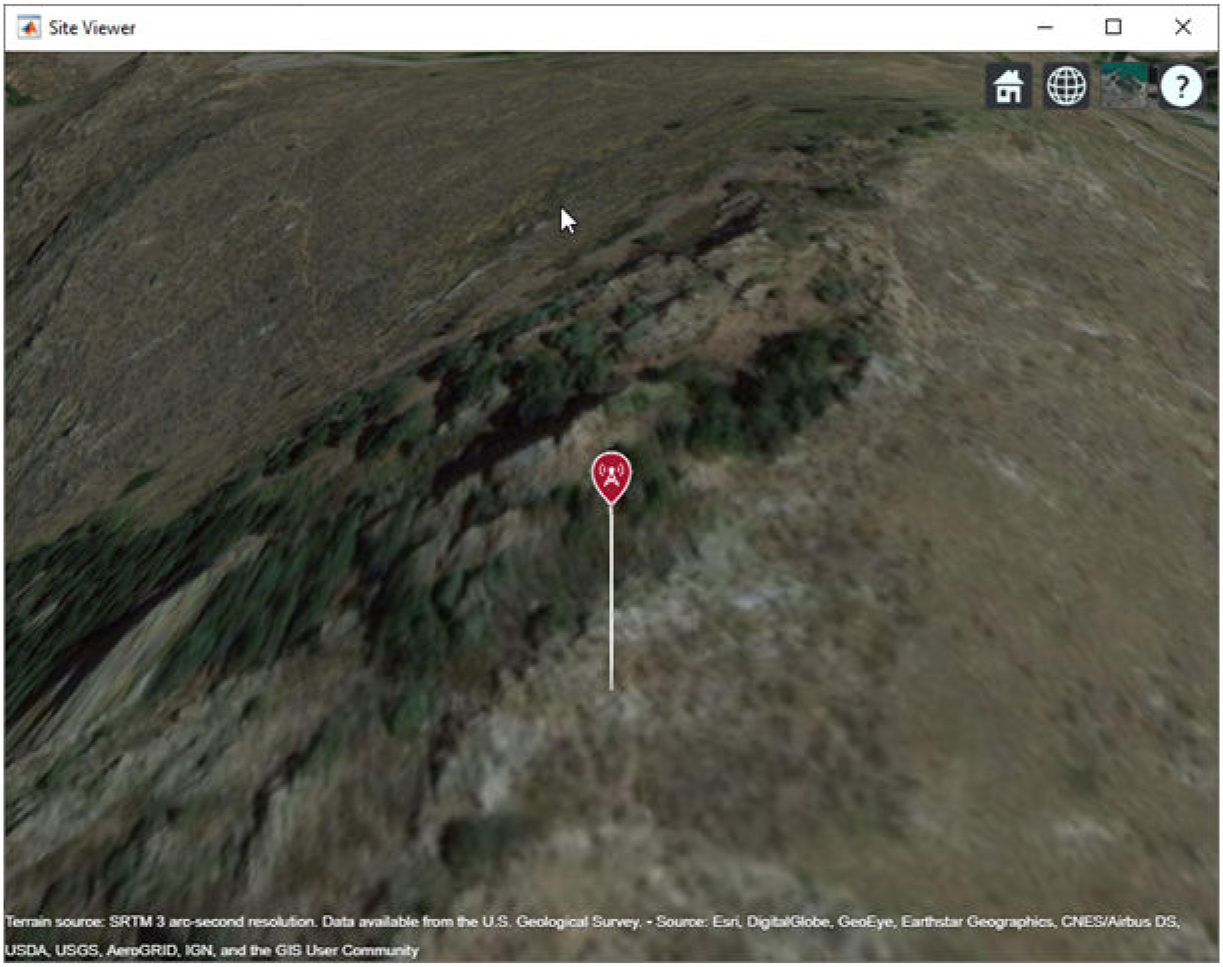
```
viewer = siteviewer("Terrain","southboulder");
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

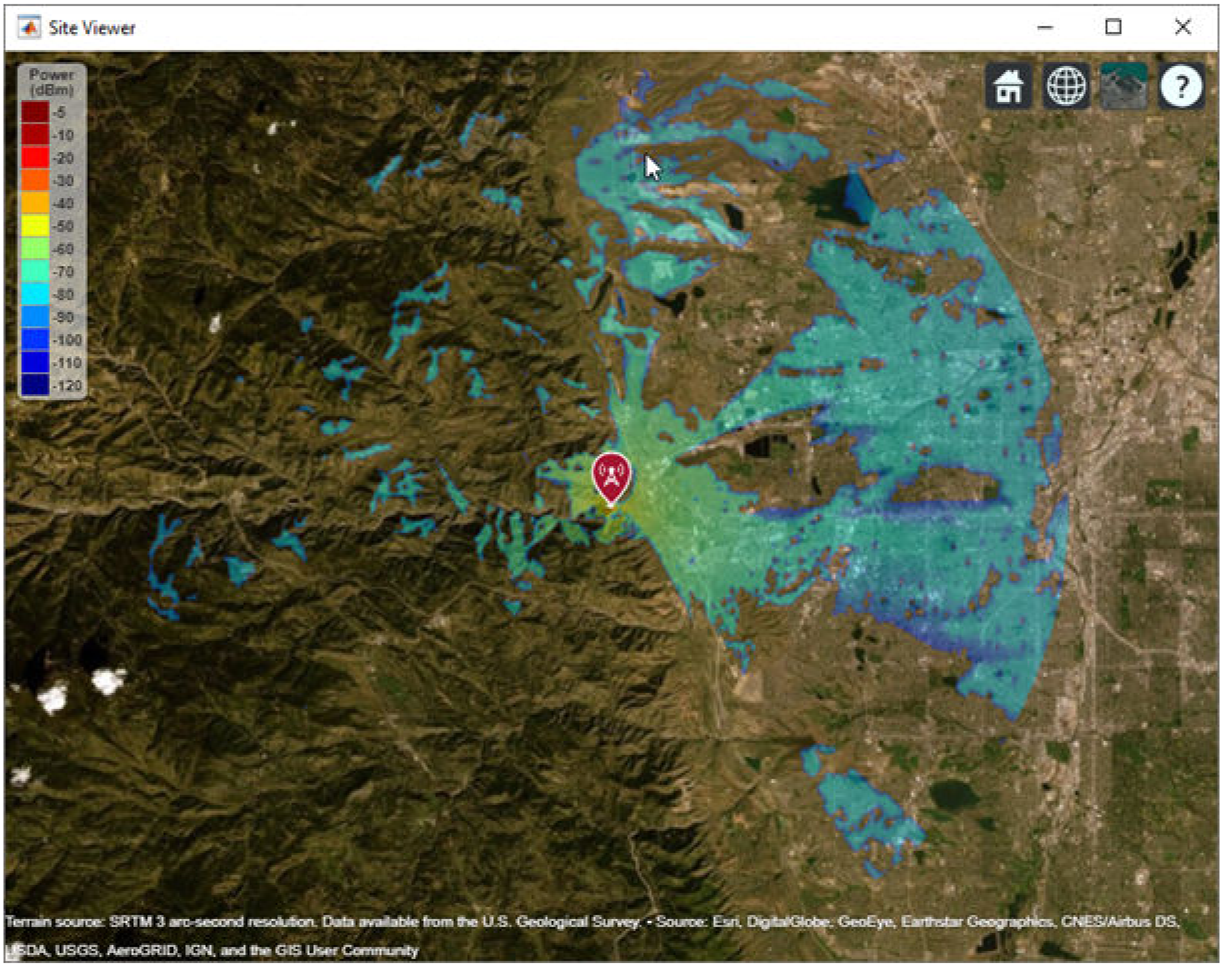
Create a site with the terrain region.

```
mtzion = txsite("Name","Mount Zion", ...
   "Latitude",39.74356, ...
   "Longitude",-105.24193, ...
   "AntennaHeight", 30);
show(mtzion)
```


Site Viewer
Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create a coverage map of the area within 20 km of the transmitter site.

```
coverage(mtzion, ...
   "MaxRange",20000, ...
   "SignalStrengths",-100:-5)
```



Terrain source: SRTM 3 arc-second resolution. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Remove the custom terrain.

```
close(viewer)
removeCustomTerrain("southboulder")
```

Input Arguments

terrainName — User-defined identifier for terrain data

string scalar | character vector

User-defined identifier for terrain data previously added using `addCustomTerrain`, specified as a string scalar or a character vector.

Data Types: `char` | `string`

See Also

`addCustomTerrain` | `siteviewer`

Introduced in R2019b

add

Add propagation models

Syntax

`pmc = add(propmodel1,propmodel2)`

Description

`pmc = add(propmodel1,propmodel2)` adds propagation model objects `propmodel1` and `propmodel2` and returns a composite propagation model object `pmc` which contains `propmodel1` and `propmodel2`.

Note

- The syntax `propmodel1+propmodel2` can be used in place of add.
- A composite propagation model cannot contain more than one propagation model object of the same class.
- A composite propagation model cannot contain more than one propagation model object which includes effects of free-space loss.

Examples

Signal Strength Over Terrain Using Composite Propagation Model

Specify the transmitter and the receiver sites.

```
tx = txsite("Name","Fenway Park", ...
      "Latitude",42.3467, ...
        "Longitude",-71.0972, ...
        "TransmitterFrequency",6e9);
rx = rxsite("Name","Bunker Hill Monument", ...
        "Latitude",42.3763, ...
        "Longitude",-71.0611);
```

Calculate signal strength using default Longley-Rice model.

```
 ss1 = sigstrength(rx,tx)
```

```
ss1 = -80.9353
```

Create composite propagation model with Longley-Rice and specific atmospheric propagation models.

```
pm = propagationModel("longley-rice") + ...
       propagationModel("gas") + propagationModel("rain");
```

Calculate signal strength using composite propagation model.

```
ss2 = sigstrength(rx,tx,pm)
```

```
ss2 = -81.2259
```

Input Arguments

`propmodel1` — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can also use the `propagationModel` function to define this input.

Data Types: `char` | `string`

`propmodel2` — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can also use the `propagationModel` function to define this input.

Data Types: `char` | `string`

Output Arguments

`pmc` — Composite propagation model

composite `propagationModel` function object

Composite propagation model, composite `propagationModel` function object

The path loss computed by `pmc` is the sum of path losses computed by `propmodel1` and `propmodel2`. If either `propmodel1` or `propmodel2` is a multipath propagation model, then `pmc` is also a multipath propagation model where path losses from rain, gas, or fog models in the composite are added to the path loss computed for each propagation path.

See Also

`propagationModel` | `range`

Introduced in R2020a

pattern

Plot antenna radiation pattern on map

Syntax

```
pattern(tx)
pattern(rx,frequency)
pattern(___,Name,Value)
```

Description

`pattern(tx)` plots the 3-D antenna radiation pattern for the transmitter site, `txsite`. Signal gain value (dBi) in a particular direction determines the color of the pattern.

Note This function only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`pattern(rx,frequency)` plots the 3-D radiation pattern for the receiver site, `rxsite` for the specified `frequency`.

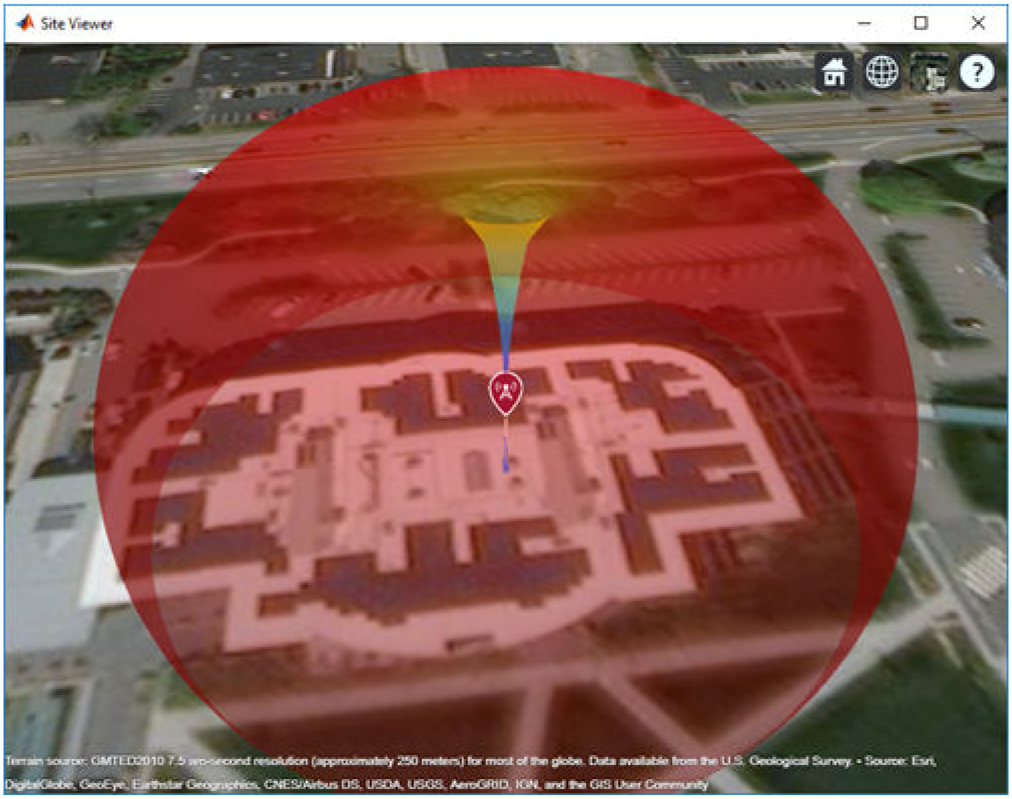
`pattern(___,Name,Value)` plots the 3-D radiation pattern with additional options specified by name-value pair arguments.

Examples

Single Transmitter Site Pattern

Define and visualize the radiation pattern of a single transmitter site.

```
tx = txsite;
pattern(tx)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

tx — Transmitter site

txsite object

Transmitter site, specified as a txsite object.

rx — Receiver site

rxsite object

Receiver site, specified as a rxsite object.

frequency — Frequency to calculate radiation pattern

positive scalar

Frequency to calculate radiation pattern, specified as a positive scalar.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Size',2

Size — Size of pattern plot

50 (default) | numerical scalar

Size of the pattern plot, specified as the comma-separated pair consisting of `'Size'` and a numerical scalar in meters. This parameter represents the distance between the antenna position and the point on the plot with the highest gain.

Data Types: `double`

`Transparency` — Transparency of pattern plot
`0.4` (default) | real number in the range of [0,1]

Transparency of the pattern plot, specified as the comma-separated pair consisting of `'Transparency'` and a real number in the range of [0,1], where `0` is completely transparent and `1` is completely opaque.

Data Types: `double`

`Colormap` — Colormap for coloring of pattern plot
`'jet(256)'` (default) | predefined colormap name | *M*-by-3 array of RGB triplets

Colormap for coloring of the pattern plot, specified as the comma-separated pair consisting of `'Colormap'` and predefined colormap name or an *M*-by-3 array of RGB (red, blue, green) triplets that define *M* individual colors.

Data Types: `double`

`Resolution` — Resolution of 3-D pattern
`'high'` (default) | `'low'` | `'`

Resolution of 3-D map, specified as the comma-separated pair consisting of `'Resolution'` and `'low'`, `'medium'`, or `'high'`. This property controls the visual quality and the time taken to plot the pattern where the value of `'low'` corresponds to the fastest and the least detailed pattern.

Data Types: `double`

`Map` — Map for visualization of surface data
`siteviewer` object

Map for visualization of surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[7]

Data Types: `char` | `string`

See Also

`coverage`

Introduced in R2019b

[7] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

show

Show site location on map

Syntax

```
show(site)
show(site,Name,Value)
```

Description

`show(site)` displays the location of transmitter or receiver site on a map using a marker.

Note This function only supports antenna sites with `CoordinateSystem` property set to `'geographic'`.

`show(site,Name,Value)` displays `site` on a map with additional options specified by one or more Name-Value pairs.

Examples

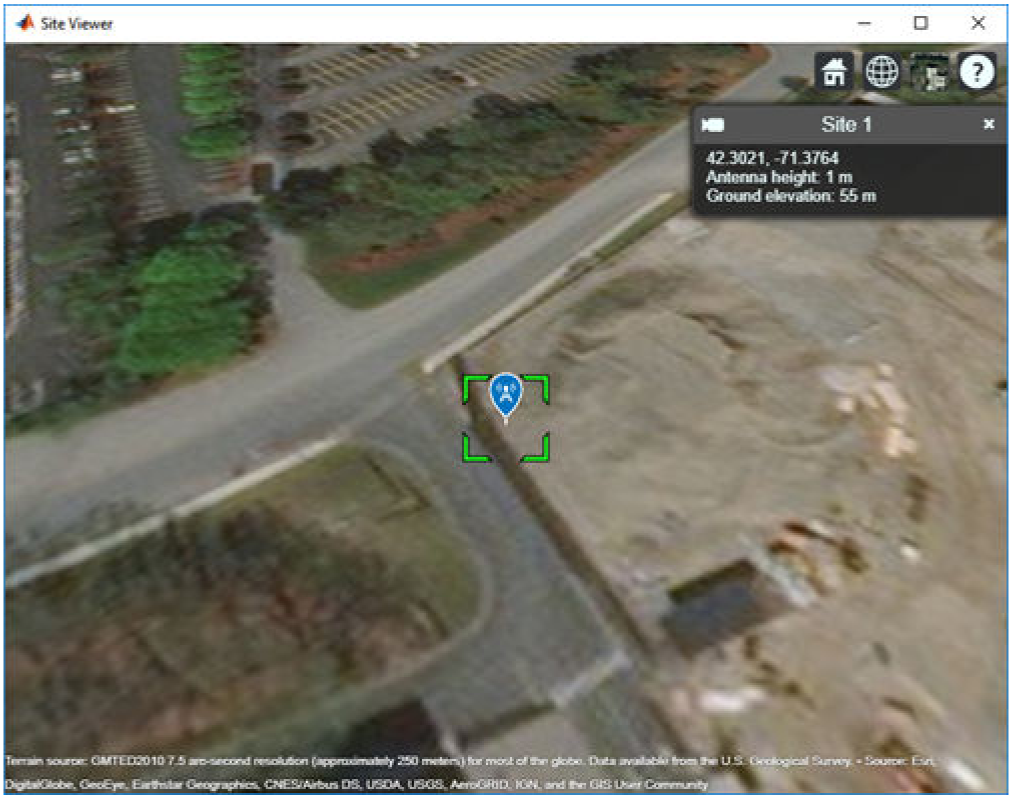
Default Receiver Site

Create and show the default receiver site.

```
rx = rxsite
```

```
rx = 
  rxsite with properties:

                   Name: 'Site 2'
               Latitude: 42.3021
              Longitude: -71.3764
                Antenna: 'isotropic'
           AntennaAngle: 0
          AntennaHeight: 1
             SystemLoss: 0
    ReceiverSensitivity: -100
```

```
show(rx)
```



Site 1

42.3021, -71.3764
Antenna height: 1 m
Ground elevation: 55 m

Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

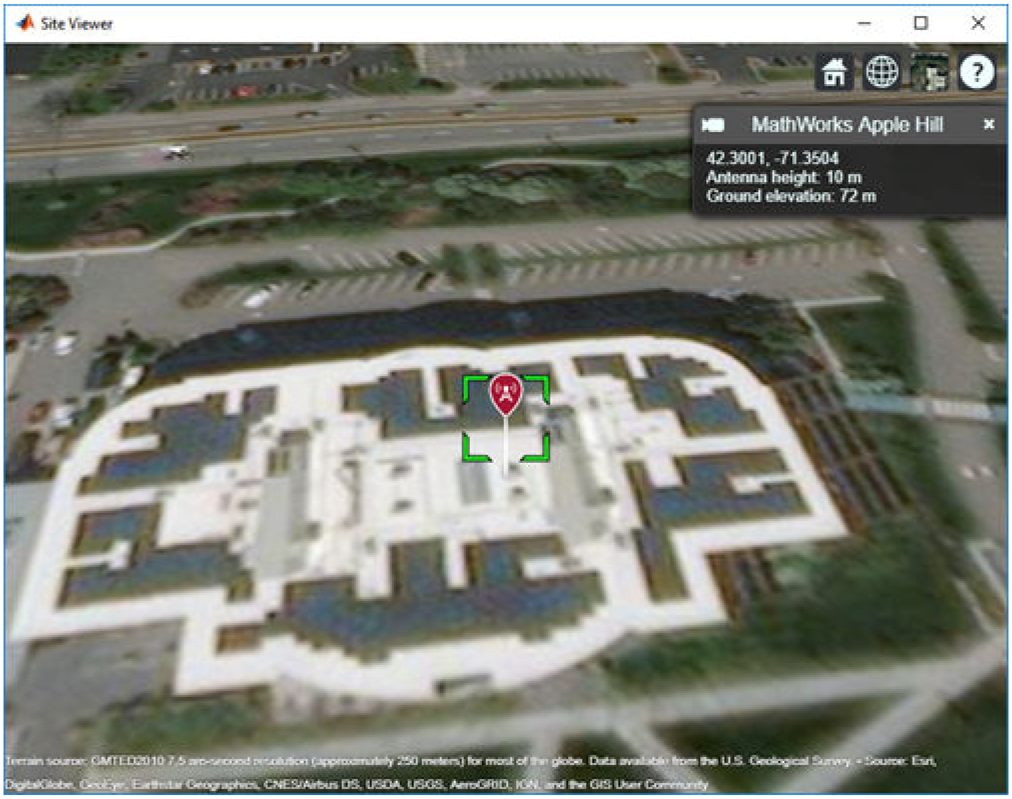
Show and Hide Transmitter Site

Create a transmitter site.

```
tx = txsite('Name','MathWorks Apple Hill',...
       'Latitude',42.3001, ...
       'Longitude',-71.3504);
```

Show the transmitter site.

```
show(tx)
```



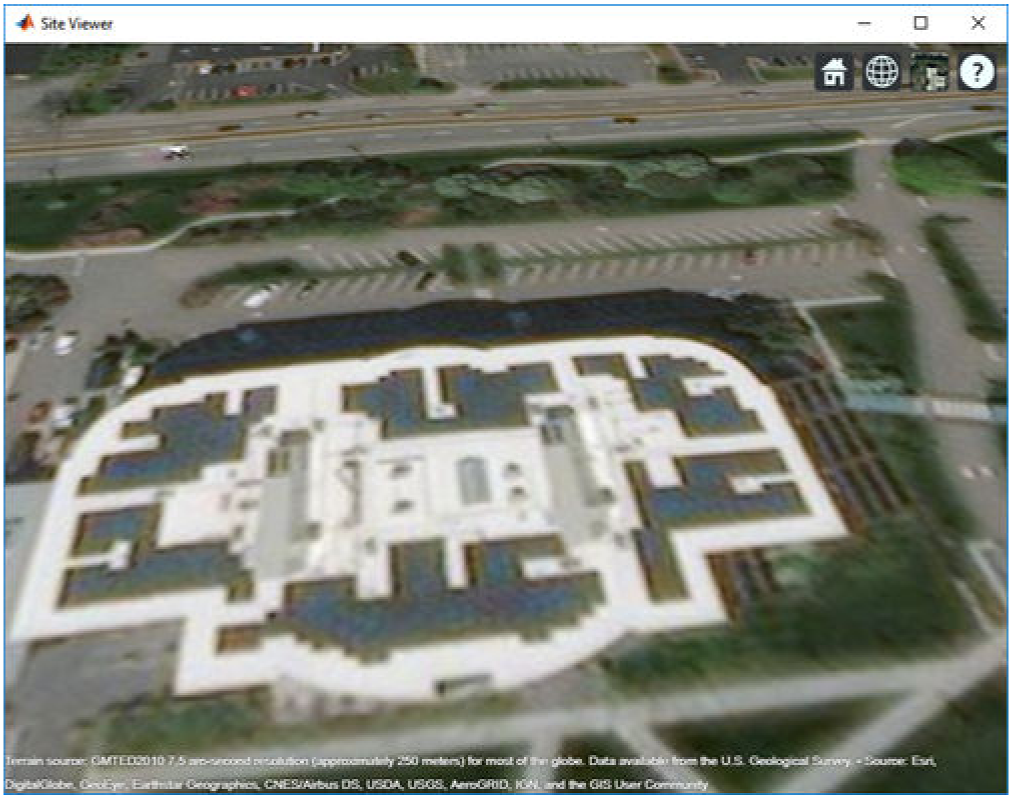
MathWorks Apple Hill

42.3001, -71.3504
Antenna height: 10 m
Ground elevation: 72 m

Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Hide the transmitter site.

```
hide(tx)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

`site` — Transmitter or receiver site
`txsite` or `rxsite` object | array of `txsite` or `rxsite` objects

Transmitter or receiver site, specified as a `txsite` or `rxsite` object or an array of `txsite` or `rxsite` objects.

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'ClusterMarkers',true`

`Icon` — Image file
character vector

Image file, specified as a character vector.

Data Types: `char`

`IconSize` — Width and height of icon
`36-by-36` (default) | 1-by-2 vector of positive numeric values

Width and height of the icon, specified as a 1-by-2 vector of positive numeric values in pixels.

`IconAlignment` — Vertical position of icon relative to site

'top' (default) | `'center'` | `'bottom'`

Vertical position of icon relative to site, specified as:

- `'bottom` - Aligns the icon below the site antenna position.
- `'center'` - Aligns the center of the icon to the site antenna position.
- `'top'` - Aligns the icon above the site antenna position.

`ClusterMarkers` — Combine nearby markers into groups or clusters

true | false

Combine nearby markers into groups or clusters, specified as true or false.

Data Types: `char`

`Map` — Map for visualization of surface data

`siteviewer` object

Map for visualization of surface data, specified as the comma-separated pair consisting of `'Map'` and a `siteviewer` object.[8]

Data Types: `char` | `string`

See Also

`hide`

Introduced in R2019b

[8] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

sigstrength

Signal strength due to transmitter

Syntax

```
ss = sigstrength(rx,tx)
ss = sigstrength(rx,tx,propmodel)
ss = sigstrength( ___ ,Name,Value)
```

Description

`ss = sigstrength(rx,tx)` returns the signal strength at the receiver site due to the transmitter site.

`ss = sigstrength(rx,tx,propmodel)` returns the signal strength at the receiver site using the specified propagation model. Specifying propagation model is same as specifying the `'PropagationModel'` name-value pair.

`ss = sigstrength( ___ ,Name,Value)` returns the signal strength using additional options specified by `Name,Value` pairs and either of the previous syntaxes.

Examples

Received Power and Link Margin at Receiver

Create a transmitter site.

```
tx = txsite('Name','Fenway Park', ...
        'Latitude', 42.3467, ...
        'Longitude', -71.0972);
```

Create a receiver site with sensitivity defined (in dBm).

```
 rx = rxsite('Name','Bunker Hill Monument', ...
        'Latitude', 42.3763, ...
        'Longitude', -71.0611, ...
        'ReceiverSensitivity', -90);
```

Calculate the received power and link margin. Link margin is the difference between the receiver's sensitivity and the received power.

```
ss = sigstrength(rx,tx)
```

```
ss = -71.1414
```

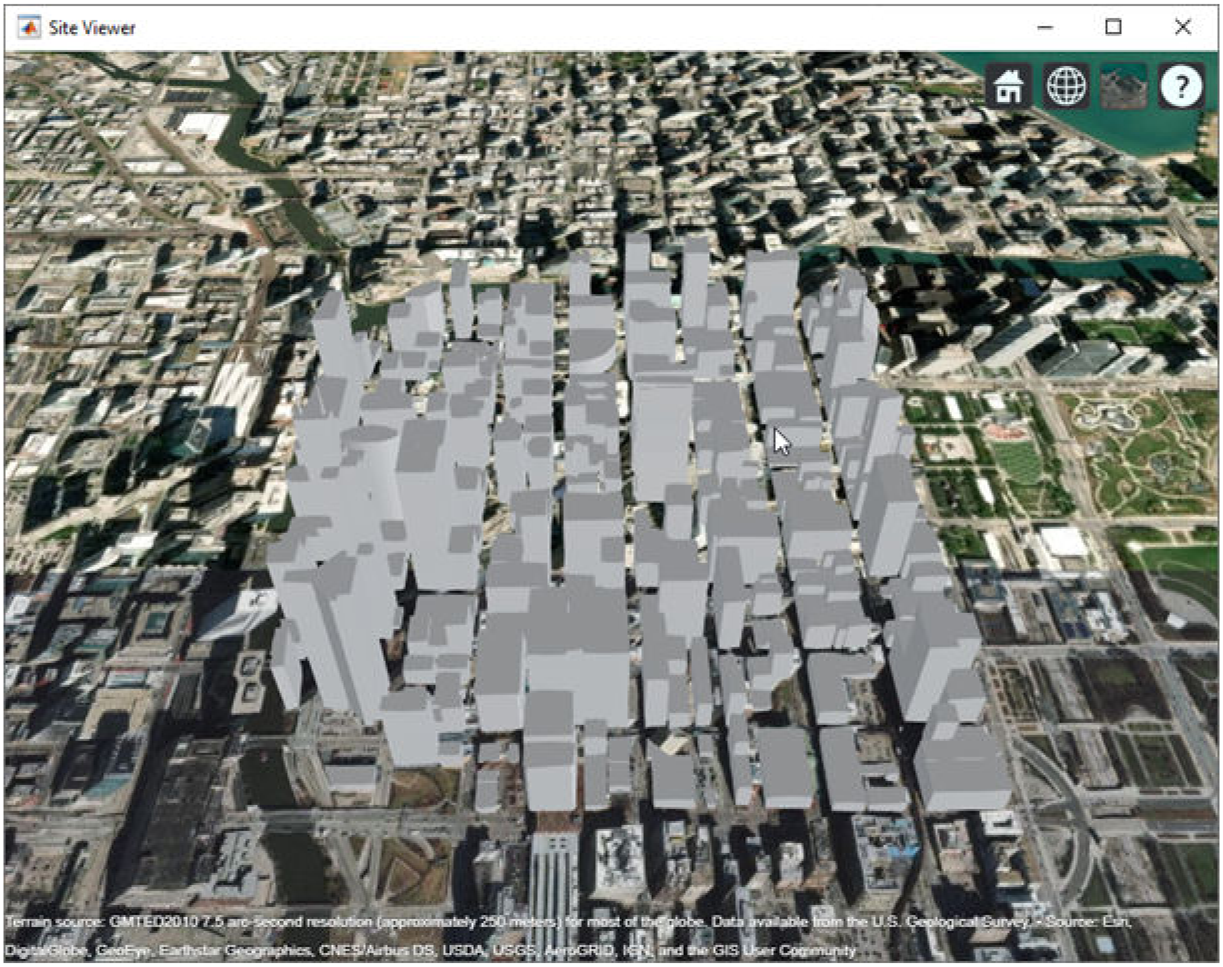
```
margin = abs(rx.ReceiverSensitivity - ss)
```

```
margin = 18.8586
```

Signal Strength Using Ray Tracing Image Method Propagation Model

Launch Site Viewer with buildings in Chicago. For more information about the osm file, see [1] on page 4-0 .

```
viewer = siteviewer("Buildings","chicago.osm");
```


Site Viewer
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. • Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Create transmitter site on a building.

```
tx = txsite('Latitude',41.8800, ...
    'Longitude',-87.6295, ...
    'TransmitterFrequency',2.5e9);
```

Create receiver site near another building.

```
rx = rxsite('Latitude',41.881352, ...
    'Longitude',-87.629771, ...
    'AntennaHeight',30);
```

Compute signal strength using ray tracing propagation model and default single-reflection analysis.

```
pm = propagationModel("raytracing-image-method");
ssOneReflection = sigstrength(rx,tx,pm)
ssOneReflection = -54.0915
```

Compute signal strength with analysis up to two reflections, where total received power is the cumulative power of all propagation paths

```
pm.MaxNumReflections = 2;
ssTwoReflections = sigstrength(rx,tx,pm)
```

```
ssTwoReflections = -52.3890
```

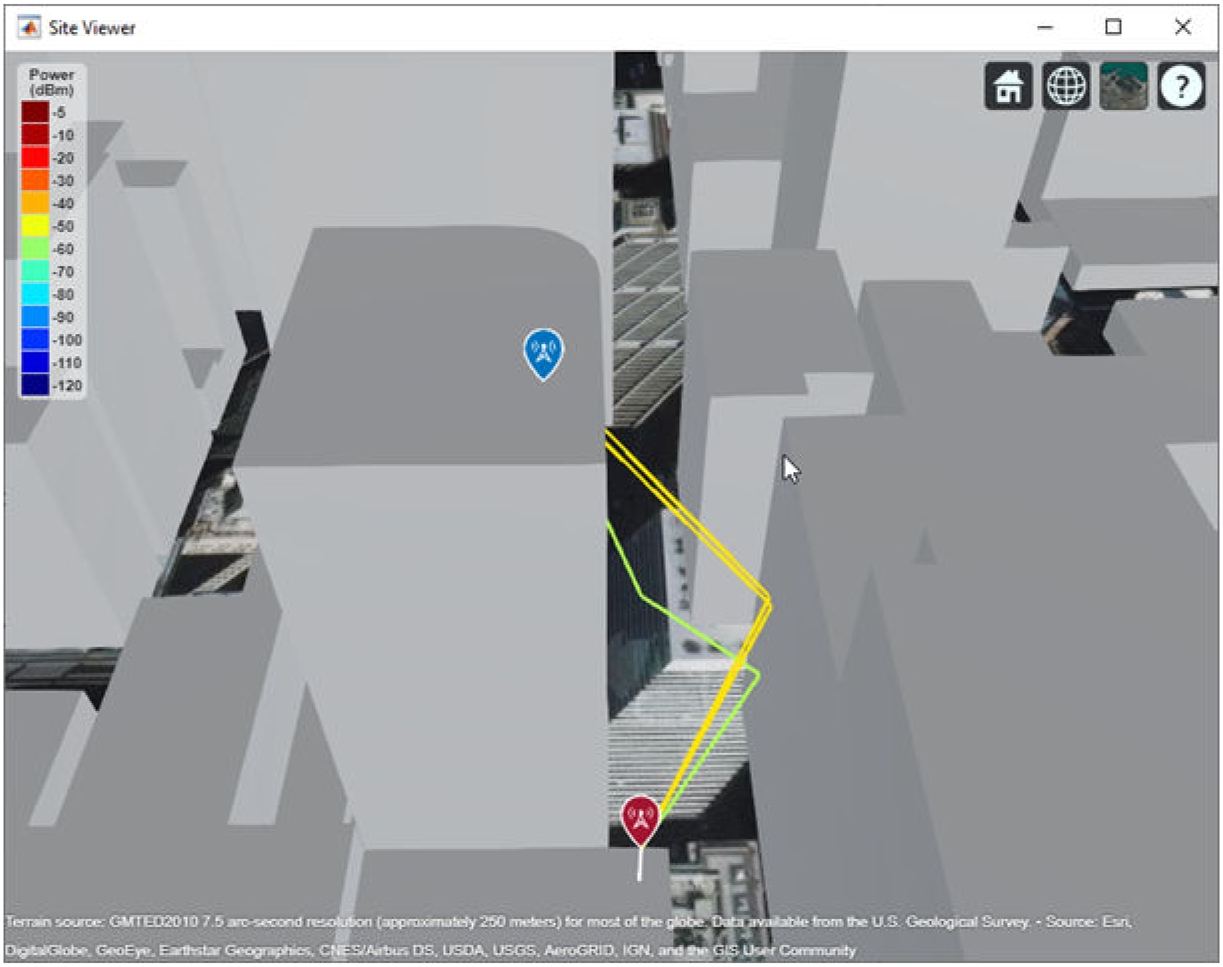
Observe effect of material by replacing default concrete material with perfect reflector.

```
pm.BuildingsMaterial = 'perfect-reflector';
ssPerfect = sigstrength(rx,tx,pm)
```

```
ssPerfect = -41.9927
```

Plot propagation paths.

```
raytrace(tx, rx, pm)
```



Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

rx — Receiver site

rxsite object | array of rxsite objects

Receiver site, specified as a rxsite object. You can use array inputs to specify multiple sites.

tx — Transmitter site

txsite object | array of txsite objects

Transmitter site, specified as a txsite object. You can use array inputs to specify multiple sites.

propmodel — Propagation model

character vector | string

Propagation model, specified as a character vector or string. You can use the propagationModel function to define this input. The default value depends on the coordinate system used by the input sites:

Coordinate System	Default propagation model value
'geographic'	• 'longley-rice' when you use a terrain. • 'freespace' when you do not use a terrain.
'cartesian'	• 'freespace' when Map is set to none. • 'raytracing-image-method' when Map is set to the name of an STL file or a triangulation object.

You can also use the name-value pair 'PropagationModel' to specify this parameter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'Type','power'

Type — Type of signal strength to compute

'power' (default) | 'efield'

Type of signal strength to compute, specified as the comma-separated pair consisting of 'Type and 'power' or 'efield'.

When type is 'power', signal strength is expressed in power units (dBm) of the signal at the mobile receiver input. When type is 'efield', signal strength is expressed in electric field strength units (dBµV/m) of signal wave incident on the antenna.

Data Types: char | string

PropagationModel — Propagation model to use for path loss calculations

'longley-rice' (default) | 'freespace' | 'close-in' | 'rain' | 'gas' | 'fog' | 'raytracing-image-method' | propagation model object

Propagation model to use for the path loss calculations, specified as the comma-separated pair consisting of 'PropagationModel' and one of the following:

- 'freespace' - Free space propagation model
- 'rain' - Rain propagation model
- 'gas' - Gas propagation model
- 'fog' - Fog propagation model
- 'close-in' - Close-in propagation model
- 'longley-rice' - Longley-Rice propagation model
- 'tirem' - Tirem propagation model
- 'raytracing-image-method' - -Raytracing propagation model using method of images.

The default propagation model is 'longley-rice' when terrain is enabled and 'freespace' when terrain is disabled.

Terrain propagation models including 'longley-rice' and 'tirem' are only supported for sites with CoordinateSystem property set to 'geographic'.

Data Types: char

Map — Map for visualization or surface data

siteviewer object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of 'Map and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
'geographic'	• siteviewer[a] • A terrain name may be specified if the function is called with an output argument. Valid terrain names are 'none', 'gmted2010', or the name of the custom terrain data added using addCustomTerrain	• current siteviewer or new siteviewer if none are open. • 'gmted2010' if called with an output.
'cartesian'	'none', triangulation object or name of an STL file.	'none'

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: char | string

Output Arguments

`ss` — Signal strength

M-by-*N* array

Signal strength, returned as *M*-by-*N* array in dBm. *M* is the number of TX sites and *N* is the number of RX sites.

See Also

`link` | `propagationModel` | `sinr`

Introduced in R2019b

sinr

Display signal-to-interference-plus-noise ratio (SINR) map

Syntax

```
sinr(txs)
sinr(txs,propmodel)
sinr( ___ ,Name,Value)
pd = sinr(txs, ___ )
r = sinr(rxs,txs, ___ )
```

Description

`sinr(txs)` displays the signal-to-interference-plus-noise ratio (SINR) for transmitter sites, `txs`. The map contours are generated using SINR values computed for receiver site locations on the map. For each location, the signal source is the transmitter site in TXS with the greatest signal strength. The remaining transmitter sites in `txs` with the same transmitter frequency act as sources of interference. If `txs` is scalar or there are no sources of interference the resultant map displays signal-to-noise ratio (SNR).

`sinr(txs,propmodel)` displays the SINR map with the propagation model set to the value in `propmodel`.

`sinr( ___ ,Name,Value)` sets properties using one or more name-value pairs, in addition to the input arguments in previous syntaxes. For example, `sinr(txs,'MaxRange',8000)` sets the range from the site location at 8000 meters to include in the SINR map region.

`pd = sinr(txs, ___ )` returns computed SINR data in the propagation data object, `pd`. No plot is displayed and any graphical only name-value pairs are ignored.

`r = sinr(rxs,txs, ___ )` returns the `sinr` in dB computed at the receiver sites due to the transmitter sites.

Examples

SINR Map for Multiple Transmitters

Define names and location of sites in Boston.

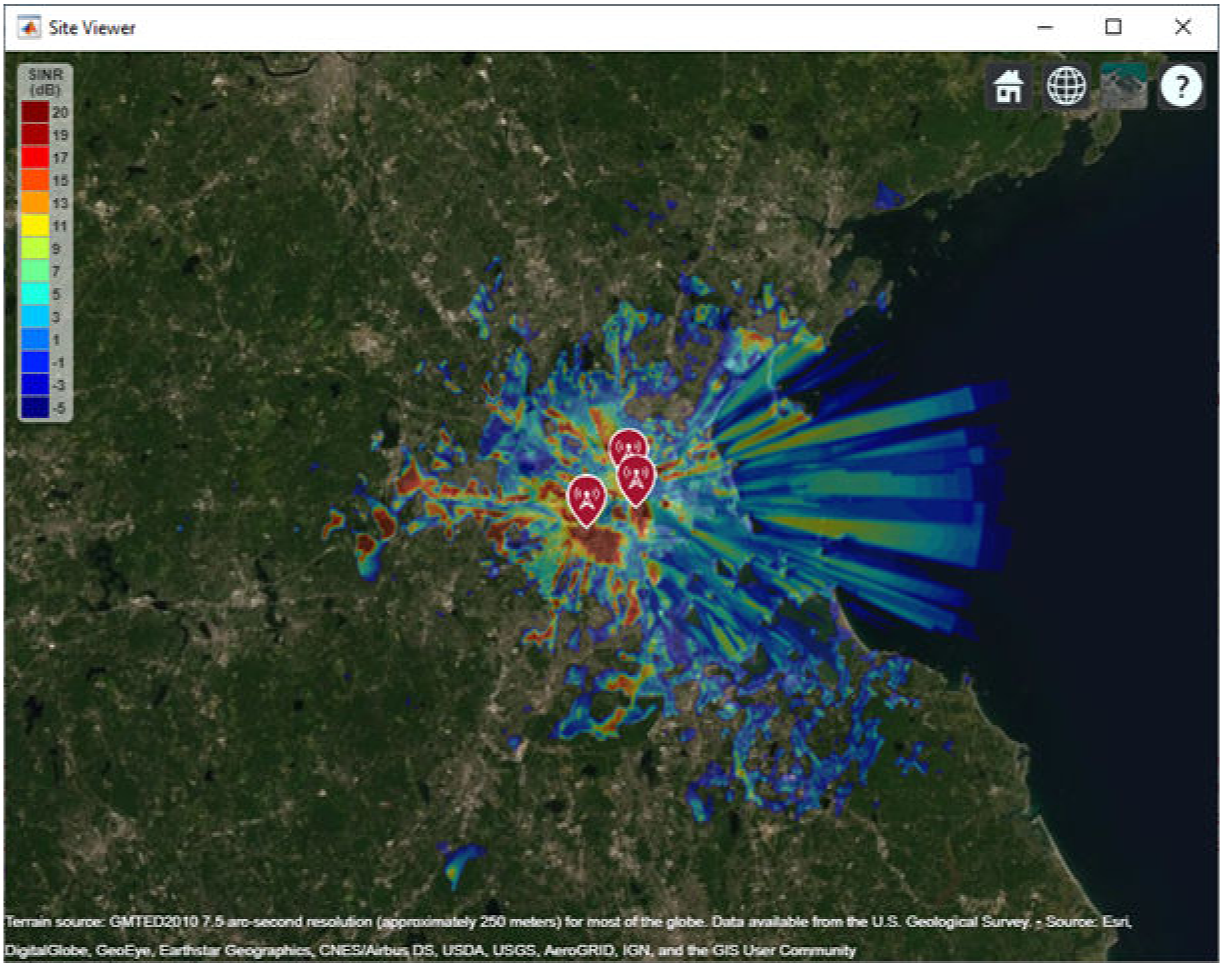
```
names = ["Fenway Park","Faneuil Hall","Bunker Hill Monument"];
lats = [42.3467,42.3598,42.3763];
lons = [-71.0972,-71.0545,-71.0611];
```

Create a transmitter site array.

```
txs = txsite('Name', names,...
       'Latitude',lats,...
       'Longitude',lons, ...
       'TransmitterFrequency',2.5e9);
```

Display the SINR map, where signal source for each location is selected as the transmitter site with the strongest signal.

```
sinr(txs)
```


Site Viewer
SINR (dB)
20
19
17
15
13
11
9
7
5
3
1
-1
-3
-5
?
Terrain source: GMTED2010 7.5 arc-second resolution (approximately 250 meters) for most of the globe. Data available from the U.S. Geological Survey. - Source: Esri, DigitalGlobe, GeoEye, Earthstar Geographics, CNES/Airbus DS, USDA, USGS, AeroGRID, IGN, and the GIS User Community

Input Arguments

txs — Transmitter sites

`txsite` object | array of `txsite` objects

Transmitter site, specified as a `txsite` object. Use array inputs to specify multiple sites.

This function only supports plotting antenna sites when `CoordinateSystem` property is set to `'geographic'`.

rxs — Receiver sites

`rxsite` object | array of `rxsite` objects

Receiver site, specified as a `rxsite` object. Use array inputs to specify multiple sites.

This function only supports plotting antenna sites when `CoordinateSystem` property is set to `'geographic'`.

propmodel — Propagation model
character vector | string

Propagation model, specified as a character vector or string. You can use the propagationModel function to define this input. The default value depends on the coordinate system used by the input sites:

Coordinate System	Default propagation model value
'geographic'	• 'longley-rice' when you use a terrain. • 'freespace' when you do not use a terrain.
'cartesian'	• 'freespace' when Map is set to none. • 'raytracing-image-method' when Map is set to the name of an STL file or a triangulation object.

You can also use the name-value pair 'PropagationModel' to specify this parameter.

Name-Value Pair Arguments

Specify optional comma-separated pairs of Name,Value arguments. Name is the argument name and Value is the corresponding value. Name must appear inside quotes. You can specify several name and value pair arguments in any order as Name1,Value1,...,NameN,ValueN.

Example: 'MaxRange',8000

General

SignalSource — Signal source of interest
'strongest' (default) | transmitter site object

Signal source of interest, specified as the comma-separated pair consisting of SignalSource and 'strongest' or as a transmitter site object. When the signal source of interest is 'strongest', the transmitter with the greatest signal strength is chosen as the signal source of interest for that location. When computing sinr, SignalSource can be a txsite array with equal number of elements rxs where each transmitter site element defines the signal source for the corresponding receiver site.

PropagationModel — Propagation model to use for path loss calculations
'longley-rice' (default) | 'freespace' | 'close-in' | 'rain' | 'gas' | 'fog' | 'raytracing-image-method' | propagation model object

Propagation model to use for the path loss calculations, specified as the comma-separated pair consisting of 'PropagationModel' and one of the following:

- 'freespace' - Free space propagation model
- 'rain' - Rain propagation model
- 'gas' - Gas propagation model
- 'fog' - Fog propagation model
- 'close-in' - Close-in propagation model
- 'longley-rice' - Longley-Rice propagation model
- 'tirem' - Tirem propagation model

- `'raytracing-image-method'`--Raytracing propagation model using method of images.

The default propagation model is `'longley-rice'` when terrain is enabled and `'freespace'` when terrain is disabled. If `'raytracing-image-method'` is specified, the value of `'MaxNumReflections'` property must be lesser than 1.

Terrain propagation models including `'longley-rice'` and `'tirem'` are only supported for sites with `CoordinateSystem` property set to `'geographic'`.

Data Types: `char`

`ReceiverNoisePower` — Total noise power at receiver

`-107` (default) | scalar

Total noise power at receiver, specified as the comma-separated pair consisting of `'ReceiverNoisePower'` and a scalar in dBm. The default value assumes that the receiver bandwidth is 1 MHz and receiver noise figure is 7 dB.

$$N = -174 + 10 * \log(B) + F$$

where,

- N = Receiver noise in dBm
- B = Receiver bandwidth in Hz
- F = Noise figure in dB

`ReceiverGain` — Receiver gain

`2.1` (default) | scalar

Mobile receiver gain, specified as the comma-separated pair consisting of `'ReceiverGain'` and a scalar in dB. The receiver gain values include the antenna gain and the system loss. If you call the function using an output argument, the default value is computed using `rxs`.

`ReceiverAntennaHeight` — Receiver antenna height

`1` (default) | scalar

Receiver antenna height above the ground, specified as the comma-separated pair consisting of `'ReceiverAntennaHeight'` and a scalar in meters. If you call the function using an output argument, the default value is computed using `rxs`.

`Map` — Map for visualization or surface data

`siteviewer` object | terrain name

Map for visualization or surface data, specified as the comma-separated pair consisting of `'Map` and one of the following depending on the coordinate system:

Coordinate System	Valid map values	Default map value
'geographic'	- siteviewer[a] - A terrain name may be specified if the function is called with an output argument. Valid terrain names are 'none', 'gmted2010', or the name of the custom terrain data added using addCustomTerrain	- current siteviewer or new siteviewer if none are open. - 'gmted2010' if called with an output.
'cartesian'	'none', triangulation object or name of an STL file.	'none'

[a] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Data Types: char | string

For Plotting SINR

Values — Values of SINR for display

[-5:20] (default) | numeric vector

Values of SINR for display, specified as the comma-separated pair consisting of 'Values' and a numeric vector. Each value is displayed as a different colored, filled on the contour map. The contour colors are derived using Colormap and ColorLimits.

MaxRange — Maximum range of coverage map from each transmitter site

numeric scalar

Maximum range of coverage map from each transmitter site, specified as a positive numeric scalar in meters representing great circle distance. MaxRange defines the region of interest on the map to plot. The default value is automatically computed based on the propagation model type as shown:

Propagation Model	MaxRange
Basic or urban	30 km
Terrain	30 km or distance to the furthest building.
Multipath	500 m

Data Types: double

Resolution — Resolution of receiver site locations used to compute SINR values

'auto' (default) | numeric scalar

Resolution of receiver site locations used to compute SINR values, specified as the comma-separated pair consisting of 'Resolution' and 'auto' or a numeric scalar in meters. The resolution defines the maximum distance between the locations. If the resolution is 'auto', sinr computes a value scaled to MaxRange. Decreasing the resolution increases the quality of the SINR map and the time required to create it.

Colormap — Colormap for coloring filled contours

'jet' (default) | *M*-by-3 array of RGB triplets

Colormap for coloring filled contours, specified as the comma-separated pair consisting of `'ColorMap'` and an *M*-by-3 array of RGB triplets, where *M* is the number of individual colors.

`ColorLimits` — Color limits for color maps
`[-5 20]` (default) | two-element vector

Color limits for color maps, specified as the comma-separated pair consisting of `'ColorLimits'` and a two-element vector of the form [min max]. The color limits indicate the SINR values that map to the first and last colors in the colormap.

`ShowLegend` — Show signal strength color legend on map
`'true'` (default) | `'false'`

Show signal strength color legend on map, specified as the comma-separated pair consisting of `'ShowLegend'` and `'true'` or `'false'`.

`Transparency` — Transparency of SINR map
`0.4` (default) | numeric scalar

Transparency of SINR map, specified as the comma-separated pair consisting of `'Transparency'` and a numeric scalar in the range 0–1. If the value is zero, the map is completely transparent. If the value is one, the map is completely opaque.

Output Arguments

r — Signal to interference plus noise ratio at the receiver
numeric vector (default)

Signal to interference plus noise ratio at the receiver due to the transmitter sites, returned as a numeric vector. The vector length is equal to the number of receiver sites.

Data Types: `double`

pd — SINR data
`propagationData` object

SINR data, returned as a `propagationData` object consisting of *Latitude* and *Longitude*, and a signal strength variable corresponding to the plot type. Name of the `propagationData` is `"SINR Data"`.

Note This function only supports plotting for antenna sites with `CoordinateSystem` property set to `'geographic'`.

See Also

`coverage` | `propagationModel`

Introduced in R2019b

plot (rays), plot

Package: comm

Plot rays in Site Viewer map

Syntax

```
plot(rays)
plot(rays,Name,Value)
```

Description

plot(rays) plots the propagation paths for ray objects in the Site Viewer map.

plot(rays,Name,Value) plots the propagation paths for ray objects in the Site Viewer map with additional options specified by one or more name-value pair arguments.

Examples

Plot Propagation Rays Between Sites in Chicago

Return ray tracing results in comm.Ray objects and plot the ray propagation path after relaunching the Site Viewer map.

Create a Site Viewer map, loading building data for Chicago. For more information about the osm file, see [1] on page 4-0 .

```
viewer = siteviewer('Buildings','chicago.osm');
```

Create and show a transmitter site on one building and a receiver site on another building.

```
tx = txsite('Latitude',41.8800,'Longitude',-87.6295, ...
    'TransmitterFrequency',2.5e9);
show(tx);
rx = rxsite('Latitude',41.881352,'Longitude',-87.629771, ...
    'AntennaHeight',30);
show(rx);
```

Perform ray tracing, returning the ray object results. For the configuration defined, ray tracing returns a cell array containing one ray object. Display the ray object properties. Then, close the Site Viewer map.

```
rays = raytrace(tx,rx)
```

```
rays = 1×1 cell array
    {1×1 comm.Ray}
```

```
rays{1}
```

```
ans = 
  Ray with properties:
      PathSpecification: 'Locations'
       CoordinateSystem: 'Geographic'
    TransmitterLocation: [3×1 double]
       ReceiverLocation: [3×1 double]
            LineOfSight: 0
    ReflectionLocations: [3×1 double]
              Frequency: 2.5000e+09
         PathLossSource: 'Custom'
               PathLoss: 94.0915
             PhaseShift: 1.2939

   Read-only properties:
       PropagationDelay: 5.7088e-07
    PropagationDistance: 171.1462
       AngleOfDeparture: [2×1 double]
         AngleOfArrival: [2×1 double]
         NumReflections: 1
```

```
close(viewer);
```

You can plot the rays without performing ray tracing again. Create another Site Viewer map with the same buildings. Show the transmitter and receiver sites. Using the previously returned cell array of ray objects, plot the reflected rays between the transmitter site and the receiver site. The plot function can plot the path for one ray object at a time.

```
siteviewer('Buildings','chicago.osm');
show(tx);
show(rx);
plot(rays{1},'Type','power', ...
    'TransmitterSite',tx,'ReceiverSite',rx);
```

Appendix

[1] The osm file is downloaded from https://www.openstreetmap.org, which provides access to crowd-sourced map data all over the world. The data is licensed under the Open Data Commons Open Database License (ODbL), https://opendatacommons.org/licenses/odbl/.

Input Arguments

rays — Ray configuration object

comm.Ray object

Ray configuration, specified as one `comm.Ray` object or a vector of `comm.Ray` objects. Each object must have the `PathSpecification` property set to `"Locations"` and the `CoordinateSystem` property set to `"Geographic"`.

Data Types: `comm.Ray`

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `plot(rays,"Type","pathloss","ColorLimits",[-100 0])` adds the propagation path specified in `rays` to the current Site Viewer and adjusts the default color limits.

Type — Quantity type to plot
`"pathloss"` (default) | `"power"`

Quantity type to plot, specified as `"pathloss"` or `"power"`. Based on the value specified for `Type`, the color applied along the path maps to the path loss in dB or the power in dBm of the signal along the path.

Data Types: `char` | `string`

`TransmitterSite` — Transmitter site
`txsite` object

Transmitter site, specified as a `txsite` object.

Dependencies

Applies only when `Type` is set to `"power"`.

Data Types: `char`

`ReceiverSite` — Receiver site
`rxsite` object

Receiver site, specified as an `rxsite` object.

Dependencies

Applies only when `Type` is set to `"power"`.

Data Types: `char`

`ColorLimits` — Colormap color limits
[-120 -5] or [45 160] (default) | 1-by-2 numeric vector

Color limits for colormap, specified as a 1-by-2 numeric vector, [*min*, *max*], where *min* represents the lower saturation limit and *max* represents the upper saturation limit. The default is [-120 -5] when `Type` is set to `'power'` and [45 160] when `Type` is set to `'pathloss'`.

Data Types: `double`

`Colormap` — Colormap applied to propagation path
`'jet'` (default) | *M*-by-3 numeric array

Colormap applied to propagation path, specified as an *M*-by-3 numeric array of RGB (red,green,blue) triplets that define *M* individual colors.

Data Types: `double` | `char` | `string`

`ShowLegend` — Show color legend on map
`true` (default) | `false`

Show color legend on map, specified as `true` or `false`.

Data Types: `logical`

Map — Map for visualization and surface data
siteviewer object

Map for visualization and surface data, specified as a siteviewer object.[9] The default is the current siteviewer object, or if no Site Viewer is open a new siteviewer object opens.

Data Types: siteviewer object

See Also

Functions
raytrace

Objects
comm.Ray | siteviewer

Introduced in R2020a

[9] Alignment of boundaries and region labels are a presentation of the feature provided by the data vendors and do not imply endorsement by MathWorks.

Blocks

A-Law Compressor

Implement A-law compressor for source coding

Library

Source Coding

Description

The A-Law Compressor block implements an A-law compressor for the input signal. The formula for the A-law compressor is

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

where A is the A-law parameter of the compressor, V is the peak signal magnitude for x, log is the natural logarithm, and *sgn* is the `sign` function.

The most commonly used A value is 87.6.

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

A value

The A-law parameter of the compressor.

Peak signal magnitude

The peak value of the input signal. This is also the peak value of the output signal.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

A-Law Expander

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

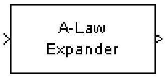
Blocks

A-Law Expander | Mu-Law Compressor

Introduced before R2006a

A-Law Expander

Implement A-law expander for source coding



Library

Source Coding

Description

The A-Law Expander block recovers data that the A-Law Compressor block compressed. The formula for the A-law expander, shown below, is the inverse of the compressor function.

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

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

A value

The A-law parameter of the compressor.

Peak signal magnitude

The peak value of the input signal. This is also the peak value of the output signal.

Match these parameters to the ones in the corresponding A-Law Compressor block.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

A-Law Compressor

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

A-Law Compressor | Mu-Law Expander

Introduced before R2006a

Algebraic Deinterleaver

Restore ordering of input symbols using algebraically derived permutation

Library

Block sublibrary of Interleaving

Description

The Algebraic Deinterleaver block restores the original ordering of a sequence that was interleaved using the Algebraic Interleaver block. In typical usage, the parameters in the two blocks have the same values.

The **Number of elements** parameter, *N*, indicates how many numbers are in the input vector. This block accepts a column vector input signal.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Type** parameter indicates the algebraic method that the block uses to generate the appropriate permutation table. Choices are `Takeshita-Costello` and `Welch-Costas`. Each of these methods has parameters and restrictions that are specific to it; these are described on the reference page for the Algebraic Interleaver block.

Parameters

Type

The type of permutation table that the block uses for deinterleaving. Choices are `Takeshita-Costello` and `Welch-Costas`.

Number of elements

The number of elements, *N*, in the input vector.

Multiplicative factor

The factor the block uses to compute the corresponding interleaver's cycle vector. This field appears only when you set **Type** to `Takeshita-Costello`.

Cyclic shift

The amount by which the block shifts indices when creating the corresponding interleaver's permutation table. This field appears only when you set **Type** to `Takeshita-Costello`.

Primitive element

An element of order *N* in the finite field GF(N+1). This field appears only if **Type** is set to `Welch-Costas`.

Pair Block

Algebraic Interleaver

References

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

[2] Takeshita, O. Y. and D. J. Costello, Jr. "New Classes Of Algebraic Interleavers for Turbo-Codes." *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16-21, 1998. 419.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Algebraic Interleaver | General Block Deinterleaver

Introduced before R2006a

Algebraic Interleaver

Reorder input symbols using algebraically derived permutation table

Library

Block sublibrary of Interleaving

Description

The Algebraic Interleaver block rearranges the elements of its input vector using a permutation that is algebraically derived. The **Number of elements** parameter, *N*, indicates how many numbers are in the input vector. This block accepts a column vector input signal.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Type** parameter indicates the algebraic method that the block uses to generate the appropriate permutation table. Choices are `Takeshita-Costello` and `Welch-Costas`. Each of these methods has parameters and restrictions that are specific to it:

- If you set **Type** to `Welch-Costas`, then $N + 1$ must be prime. The **Primitive element** parameter is an integer, *A*, between 1 and *N* that represents a primitive element of the finite field GF($N + 1$). This means that every nonzero element of GF($N + 1$) can be expressed as *A* raised to some integer power.

  In a Welch-Costas interleaver, the permutation maps the integer *k* to mod(A^k,$N + 1$) - 1.
- If you set **Type** to `Takeshita-Costello`, then *N* must be 2^m for some integer *m*. The **Multiplicative factor** parameter, *k*, must be an odd integer less than *N*. The **Cyclic shift** parameter, *h*, must be a nonnegative integer less than *N*.

  A Takeshita-Costello interleaver uses a length-*N* *cycle vector* whose *n*th element is

  $$c(n) = \mathrm{mod}\left(k \cdot \frac{n \cdot (n-1)}{2}, N\right) + 1,\ n$$

  for integers n between 1 and N. The intermediate permutation function is obtained by using the following relationship:

  $$\Pi(c(n)) = c(n+1)$$

  where

  $$n = 1{:}N$$

  The interleaver's actual permutation vector is the result of cyclically shifting the elements of the permutation vector, π, by the **Cyclic shift** parameter, *h*.

Parameters

Type

The type of permutation table that the block uses for interleaving.

Number of elements

The number of elements, *N*, in the input vector.

Multiplicative factor

The factor used to compute the interleaver's cycle vector. This field appears only if **Type** is set to `Takeshita-Costello`.

Cyclic shift

The amount by which the block shifts indices when creating the permutation table. This field appears only if **Type** is set to `Takeshita-Costello`.

Primitive element

An element of order *N* in the finite field GF(N+1). This field appears only if **Type** is set to `Welch-Costas`.

Pair Block

Algebraic Deinterleaver

References

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

[2] Takeshita, O. Y. and D. J. Costello, Jr. "New Classes Of Algebraic Interleavers for Turbo-Codes." *Proc. 1998 IEEE International Symposium on Information Theory*, Boston, Aug. 16-21, 1998. 419.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

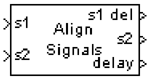
See Also

Blocks
Algebraic Deinterleaver | General Block Interleaver

Introduced before R2006a

Align Signals

(To be removed) Align two signals by finding delay between them



Align Signals

Compatibility

Align Signals Block will be removed in a future release. Use the Find Delay block to find delays and the Delay block to apply delays instead. For more information, see “Compatibility Considerations” on page 5-11.

Library

Utility Blocks

Description

The Align Signals block aligns two signals by finding the delay between them. This is useful when you want to compare a transmitted and received signal to determine the bit error rate, but do not know the delay in the received signal. This block accepts a column vector or matrix input signal. For a matrix input, the block aligns each channel independently.

The `s1` input port receives the original signal, while the `s2` input port receives a delayed version. The two input signals must have the same dimensions and sample times. The block calculates the delay between the two signals, and then

- Delays the first signal, `s1`, by the calculated value, and outputs it through the port labeled `s1 del`.
- Outputs the second signal `s2` without change through the port labeled `s2`.
- Outputs the delay value through the port labeled `delay`.

See “Delays” for more information about signal delays.

The block's **Correlation window length** parameter specifies how many samples of the signals the block uses to calculate the cross-correlation. The delay output is a nonnegative integer less than the **Correlation window length**.

As the **Correlation window length** is increased, the reliability of the computed delay also increases. However, the processing time to compute the delay increases as well.

You can make the Align Signals block stop updating the delay after it computes the same delay value for a specified number of samples. To do so, select **Disable recurring updates**, and enter a positive integer in the **Number of constant delay outputs to disable updates** field. For example, if you set **Number of constant delay outputs to disable updates** to `20`, the block will stop recalculating and updating the delay after it calculates the same value 20 times in succession. Disabling recurring updates causes the simulation to run faster after the target number of constant delays occurs.

Tips for Using the Block Effectively

- Set the **Correlation window length** parameter sufficiently large so that the computed delay eventually stabilizes at a constant value. If the computed delay is not constant, you should increase **Correlation window length**. If the increased value of **Correlation window length** exceeds the duration of the simulation, then you should also increase the duration of the simulation accordingly.
- If the cross-correlation between the two signals is broad, then **Correlation window length** should be much larger than the expected delay, or else the algorithm might stabilize at an incorrect value. For example, a CPM signal has a broad autocorrelation, so it has a broad cross-correlation with a delayed version of itself. In this case, the **Correlation window length** value should be much larger than the expected delay.
- If the block calculates a delay that is greater than 75 percent of **Correlation window length**, the signal `s1` is probably delayed relative to the signal `s2`. In this case, you should switch the signal lines leading into the two input ports.
- If you use the Align Signals block with the Error Rate Calculation block, you should set the **Receive delay** parameter of the Error Rate Calculation block to `0` because the Align Signals block compensates for the delay. Also, you might want to set the Error Rate Calculation block's **Computation delay** parameter to a nonzero value to account for the possibility that the Align Signals block takes a nonzero amount of time to stabilize on the correct amount by which to delay one of the signals.

Parameters

Correlation window length

The number of samples the block uses to calculate the cross-correlations of the two signals.

Disable recurring updates

Selecting this option causes the block to stop computing the delay after it computes the same delay value for a specified number of samples.

Number of constant delay outputs to disable updates

A positive integer specifying how many times the block must compute the same delay before ceasing to update. This field appears only if **Disable recurring updates** is selected.

Algorithm

The Align Signals block finds the delay by calculating the cross-correlations of the first signal with time-shifted versions of the second signal, and then finding the index at which the cross-correlation is maximized.

Compatibility Considerations

Align Signals Block will be removed

Warns starting in R2019b

Align Signals Block will be removed in a future release. Use the Find Delay block to find delays and the Delay block to apply delays instead.

See “Delays” for an example that uses the Find Delay block in conjunction with the Error Rate Calculation block.

For an example that illustrates how to set the correlation window length properly, see the section **Setting the Correlation Window Length** on the Find Delay block reference page.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

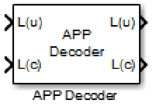
Blocks
Delay | Error Rate Calculation | Find Delay

Topics
“Delays”
“Align Signals Replacement for Single Rate Signals in Simulink”
“Align Signals Replacement for Multirate Signals in Simulink”

Introduced in R2012a

APP Decoder

Decode convolutional code using a posteriori probability (APP) method



Library

Convolutional sublibrary of Error Detection and Correction

Description

The APP Decoder block performs a posteriori probability (APP) decoding of a convolutional code.

Input Signals and Output Signals

The input $L(u)$ represents the sequence of log-likelihoods of encoder input bits, while the input $L(c)$ represents the sequence of log-likelihoods of code bits. The outputs $L(u)$ and $L(c)$ are updated versions of these sequences, based on information about the encoder.

If the convolutional code uses an alphabet of 2^n possible symbols, this block's $L(c)$ vectors have length $Q*n$ for some positive integer Q. Similarly, if the decoded data uses an alphabet of 2^k possible output symbols, then this block's L(u) vectors have length $Q*k$.

This block accepts a column vector input signal with any positive integer for Q.

If you only need the input $L(c)$ and output $L(u)$, you can attach a Simulink Ground block to the input $L(u)$ and a Simulink Terminator block to the output $L(c)$.

This block accepts `single` and `double` data types. Both inputs, however, must be of the same type. The output data type is the same as the input data type.

Specifying the Encoder

To define the convolutional encoder that produced the coded input, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code”. You can use this parameter field in two ways:

- If you have a variable in the MATLAB workspace that contains the trellis structure, enter its name as the **Trellis structure** parameter. This way is preferable because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage described next.
- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```

To indicate how the encoder treats the trellis at the beginning and end of each frame, set the **Termination method** parameter to either `Truncated` or `Terminated`. The `Truncated` option indicates that the encoder resets to the all-zeros state at the beginning of each frame. The `Terminated` option indicates that the encoder forces the trellis to end each frame in the all-zeros state. If you use the Convolutional Encoder block with the **Operation mode** parameter set to `Truncated (reset every frame)`, use the `Truncated` option in this block. If you use the Convolutional Encoder block with the **Operation mode** parameter set to `Terminate trellis by appending bits`, use the `Terminated` option in this block.

Specifying Details of the Algorithm

You can control part of the decoding algorithm using the **Algorithm** parameter. The `True APP` option implements a posteriori probability decoding as per equations 20–23 in section V of [1]. To gain speed, both the `Max*` and `Max` options approximate expressions like

$$\log\sum_i \exp(a_i)$$

by other quantities. The `Max` option uses $\max(a_\mathrm{i})$ as the approximation, while the `Max*` option uses $\max(a_\mathrm{i})$ plus a correction term given by $\ln(1 + \exp(-|a_{i-1} - a_i|))$ [3].

The `Max*` option enables the **Scaling bits** parameter in the dialog box. This parameter is the number of bits by which the block scales the data it processes internally (multiplies the input by (2^`numScalingBits`) and divides the pre-output by the same factor). Use this parameter to avoid losing precision during the computations.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Termination method

Either `Truncated` or `Terminated`. This parameter indicates how the convolutional encoder treats the trellis at the beginning and end of frames.

Algorithm

Either `True APP`, `Max*`, or `Max`.

Number of scaling bits

An integer between 0 and 8 that indicates by how many bits the decoder scales data in order to avoid losing precision. This field is active only when **Algorithm** is set to `Max*`.

Disable L(c) output port

Select this check box to disable the secondary block output, L(c).

References

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

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

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

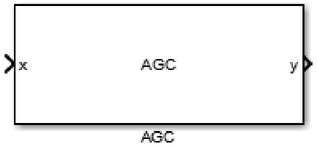
Blocks
Convolutional Encoder | Viterbi Decoder

Functions
`poly2trellis`

Introduced before R2006a

AGC Block

Adaptively adjust gain for constant signal-level output



Library

RF Impairments Correction

Description

The automatic gain controller (AGC) block adaptively adjusts its gain to achieve a constant signal level at the output.

Parameters

Step size

Specify the step size for gain updates as a double-precision or single-precision real positive scalar. The default is `0.01`.

If you increase **Step size**, the AGC responds faster to changes in the input signal level. However, gain pumping also increases.

Desired output power (W)

Specify the desired output power level as a real positive scalar. The power level is specified in Watts referenced to 1 ohm. The default is `1`.

Averaging length

Specify the length of the averaging window in samples as a positive integer scalar. The default is `100`.

Note If you use the AGC with higher order QAM signals, you might need to reduce the variation in the gain during steady-state operation. Inspect the constellation diagram at the output of the AGC and increase the averaging length as needed. An increase in **Averaging length** reduces execution speed.

Maximum power gain (dB)

Specify the maximum gain of the AGC in decibels as a positive scalar. The default is `60`.

If the AGC input signal power is very small, the AGC gain will be very large. This can cause problems when the input signal power suddenly increases. Use **Maximum power gain (dB)** to avoid this by limiting the gain that the AGC applies to the input signal.

Enable output of estimated input power

Select this check box to provide an input signal power estimate to an output port. By default, the check box is not selected.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.AGC` reference page. The object properties correspond to the block parameters.

Examples

To open these examples, enter the example names at the MATLAB command prompt:

- doc_agc_received_signal_amplitude — Adaptively adjusts the received signal power to approximately 1 Watt.
- doc_agc_plot_step_size — Plots the effect of step size on AGC performance.
- doc_agc_plot_max_gain — Shows how the maximum gain affects the ability of the AGC to reach its target output power.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

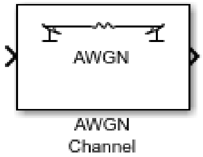
Objects

`comm.AGC`

Introduced in R2013a

AWGN Channel

Add white Gaussian noise to input signal
Library: Communications Toolbox / Channels



Description

The AWGN Channel block adds white Gaussian noise to the input signal. It inherits the sample time from the input signal.

Ports

Input

`In` — Input data signal
vector | matrix

Input data signal, specified as an N_S-by-1 vector or an N_S-by-N_C matrix.

N_S represents the number of samples in the input signal. N_C represents the number of channels, as determined by the number of columns in the input signal matrix. Both N_S and N_C can be equal to `1`.

The block adds frames of length-N_S Gaussian noise to each of the N_C channels, using a distinct random distribution per channel.

Data Types: `double` | `single`
Complex Number Support: Yes

`Var` — Variance of additive white Gaussian noise
positive scalar | vector

Variance of additive white Gaussian noise, specified as a positive scalar or a 1-by-N_C vector. N_C represents the number of channels, as determined by the number of columns in the input signal matrix. For more information, see “Specifying the Variance Directly or Indirectly” on page 5-22.

Dependencies

To enable this port, set Mode to `Variance from port`.

Data Types: `double`

Output

`Out` — Output data signal
vector | matrix

Output data signal for the AWGN channel, returned as a vector or matrix. The datatype and dimensions of `Out` match those of the input signal, In.

Parameters

`Initial seed` — Noise generator initial seed

`67` (default) | positive scalar | vector

Noise generator initial seed, specified as a positive scalar or a 1-by-N_C vector.

This block uses the Random Source block to generate noise. Random numbers are generated using the Ziggurat method (V5 RANDN algorithm). The block reuses the same initial seeds every time you rerun the simulation, so that this block outputs the same signal each time you run a simulation.

When the input signal is complex, the block creates random data as:

```
randData = randn(2*NS,NC)
noise = randData(1:2:end) + 1i(randData(2:2:end))
```

N_S is the number of samples and N_C is the number of channels.

You can specify different seed values for each DLL build.

Tunable: Yes

`Mode` — Variance mode

`Signal to noise ratio (Eb/No)` (default) | `Signal to noise ratio (Es/No)` | `Signal to noise ratio (SNR)` | `Variance from mask` | `Variance from port`

Variance mode, specified as `Signal to noise ratio (Eb/No)`, `Signal to noise ratio (Es/No)`, `Signal to noise ratio (SNR)`, `Variance from mask`, or `Variance from port`. For more information, see "Relationship Among Eb/No, Es/No, and SNR Modes" on page 5-21 and "Specifying the Variance Directly or Indirectly" on page 5-22.

`Eb/No (dB)` — Ratio of information bit energy per symbol to noise power spectral density

`10` (default) | scalar | vector

Ratio of information bit energy per symbol to noise power spectral density in decibels, specified as a scalar or vector. The information bit energy is the magnitude without channel coding.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Eb/No`.

`Es/No (dB)` — Ratio of information symbol energy per symbol to noise power spectral density

`10` (default) | scalar | vector

Ratio of information symbol energy per symbol to noise power spectral density in decibels, specified as a scalar or vector. The information bit energy is the magnitude without channel coding.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Es/No`.

`SNR (dB)` — Ratio of signal power to noise power

`10` (default) | scalar | vector

Ratio of signal power to noise power in decibels, specified as a scalar or vector.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `SNR`.

`Number of bits per symbol` — Number of bits in each input symbol

1 (default) | scalar | vector

Number of bits in each input symbol, specified as a scalar or vector.

Dependencies

To enable this parameter, set Mode to `Eb/No`.

`Input signal power, referenced to 1 ohm (watts)` — Mean square power of input

`1` (default) | scalar | vector

Mean square power of the input in watts, specified as a scalar or vector.

- When Mode is `Eb/No` or `Es/No`, the parameter is the mean square power of the input symbols.
- When `Mode` is `SNR`, this parameter is the mean square power of the input samples.

Tunable: Yes

Dependencies

To enable this parameter, set `Mode` to `Eb/No`, `Es/No`, or `SNR`.

`Symbol period (s)` — Duration of an information channel

`1` (default) | positive scalar | vector

Duration of an information channel symbol in seconds, specified as a positive scalar or vector. The duration of the information channel is measured without channel coding.

Dependencies

To enable this parameter, set Mode to `Eb/No` or `Es/No`.

`Variance` — Variance of white Gaussian noise

`1` (default) | scalar | vector

Variance of the white Gaussian noise, specified as a scalar or vector. For more information, see “Specifying the Variance Directly or Indirectly” on page 5-22.

Tunable: Yes

Dependencies

To enable this parameter, set Mode to `Variance from mask`.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Tips

- You can tune parameters in normal mode, accelerator mode, or rapid accelerator mode.
- Unless otherwise indicated, parameters are *nontunable*.
  - For nontunable parameters, when you use the Simulink Coder™ rapid simulation (RSIM) target to build an RSIM executable, you cannot change their values without recompiling the model.
  - If a parameter is *tunable*, you can change its value at any time. This is useful for Monte Carlo simulations in which you run the simulation multiple times (such as on multiple computers) with different amounts of noise.

Algorithms

Relationship Among Eb/No, Es/No, and SNR Modes

For uncoded complex input signals, the AWGN Channel block relates $E_b/N_{0,}$ E_s/N_0, and SNR according to these equations:

$$E_s/N_0 = (T_{sym}/T_{samp}) \cdot SNR$$

$$E_s/N_0 = E_b/N_0 + 10\log_{10}(k) \text{ in dB}$$

- E_s represents the signal energy in joules.
- E_b represents the bit energy in joules.
- N_0 represents the noise power spectral density in watts/Hz.
- T_{sym} represents the Symbol period (s) parameter of the block in `Es/No` mode.
- k represents the number of information bits per input symbol, Number of bits per symbol.
- T_{samp} represents the inherited sample time of the block, in seconds.

For real signal inputs, the AWGN Channel block relates E_s/N_0 and SNR according to this equation:

$$E_s/N_0 = 0.5\ (T_{sym}/T_{samp}) \cdot SNR$$

Note

- All values of power assume a nominal impedance of 1 ohm.
- The equation for the real case differs from the corresponding equation for the complex case by a factor of 2. Specifically, the object uses a noise power spectral density of $N_0/2$ watts/Hz for real input signals, versus N_0 watts/Hz for complex signals.

For more information, see “AWGN Channel Noise Level”.

Specifying the Variance Directly or Indirectly

To directly specify the variance of the noise generated by AWGN Channel, specify the Mode as:

- `Variance from mask`, where you specify the variance in the dialog box. The value must be positive.
- `Variance from port`, where you provide the variance as an input to the block. The variance input must be positive, and its sampling rate must equal that of the input signal.

For `Variance from mask` and `Variance from port` mode:

- If the variance is a scalar, then all signal channels are uncorrelated but share the same variance.
- If the variance is a vector whose length is the number of channels in the input signal, then each element represents the variance of the corresponding signal channel.

> **Note** If you apply complex input signals to the AWGN Channel block, then it adds complex zero-mean Gaussian noise with the calculated or specified variance. The variance for each quadrature component of the complex noise is half of the calculated or specified value.

To specify the variance indirectly, that is, to have the block calculate the variance, specify the Mode as:

- `Signal to noise ratio (Eb/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - Eb/No (dB), the ratio of bit energy to noise power spectral density
  - Number of bits per symbol
  - Input signal power, referenced to 1 ohm (watts), the actual power of the symbols at the input of the block
  - Symbol period (s)
- `Signal to noise ratio (Es/No)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - Es/No (dB), the ratio of signal energy to noise power spectral density
  - Input signal power, referenced to 1 ohm (watts), the actual power of the symbols at the input of the block
  - Symbol period (s)
- `Signal to noise ratio (SNR)`, where the block calculates the variance from these quantities that you specify in the dialog box:
  - SNR (dB), the ratio of signal power to noise power
  - Input signal power, referenced to 1 ohm (watts), the actual power of the samples at the input of the block

Changing the symbol period in the AWGN Channel block affects the variance of the noise added per sample, which also causes a change in the final error rate.

$$NoiseVariance = \frac{SignalPower \times SymbolPeriod}{SampleTime \times 10^{\frac{Es/No}{10}}}$$

Tip Select the symbol period equal to the symbol period of the model. The value depends on what constitutes a symbol and what the oversampling applied to it is. For example, a symbol could have 3 bits and be oversampled by 4. For more information, see “AWGN Channel Noise Level”.

References

[1] Proakis, John G. *Digital Communications*. 4th Ed. McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

MIMO Fading Channel | Random Source

Objects

`comm.AWGNChannel`

Topics

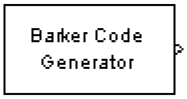
“Gray Coded 8-PSK”
“Filter Using Simulink Raised Cosine Filter Blocks”
“Reed Solomon Examples with Shortening, Puncturing, and Erasures”

Introduced before R2006a

Barker Code Generator

Generate bipolar Barker Code

Library: Communications Toolbox
Communications Toolbox / Comm Sources / Sequence Generators



Description

The Barker Code Generator block generates a bipolar Barker code. The short length and low correlation sidelobes make Barker codes useful for frame synchronization in digital communications systems. For more information, see “Barker Codes” on page 5-25.

Ports

Output

`output` — Barker code frame
column vector

Barker code frame, returned as a column vector. If the frame length exceeds the Barker code length, the block fills the frame by repeating the Barker code.

Dependencies

Set the data type of the output with the **Output data type** parameter.

Parameters

`Code length` — Length of generated code
7 (default) | 1 | 2 | 3 | 4 | 5 | 11 | 13

Length of the generated code, specified as 1, 2, 3, 4, 5, 7, 11, or 13. For more information, see “Barker Codes” on page 3-73.

Example: 2 outputs the Barker code [–1;1].

Data Types: `double`

`Sample time` — Output sample time
1 (default) | -1 | positive scalar

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-26.

`Samples per frame` — Samples per output frame
1 (default) | positive integer

Samples per output frame, specified as a positive integer. If **Samples per frame** is *M*, the block outputs a frame containing *M* samples comprised of length *N* Barker code sequences. *N* is the length of the generated code, which is set by the **Code length** parameter. When *M* is not an integer multiple of *N*, consecutive frames maintain continuity of the Barker code across frame boundaries.

For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-26.

Data Types: `double`

`Output data type` — Output data type

`double` (default) | `int8`

Output data type, specified as `double` or `int8`.

Data Types: `char` | `string`

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `integer`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Barker Codes

Barker codes have a maximum autocorrelation sequence, which has off-peak autocorrelations no larger than 1.

A correlation sidelobe is the correlation of a codeword with a time-shifted version of itself. The correlation sidelobe, C_k, for a *k*-symbol shift of an *N*-bit code sequence, $\{X_j\}$, is

$$C_k = \sum_{j=1}^{N-k} X_j X_{j+k}$$

For j=1, 2, 3,..., N, X_j is an individual code symbol that is equal to +1 or –1. The adjacent symbols are assumed to be 0.

The output code is in a bipolar format with 0 and 1 mapped to 1 and –1. The maximum known Barker code length is 13. The short length and low correlation sidelobes make Barker codes useful for frame synchronization in digital communications systems. The Barker code generator outputs the Barker codes listed in this table.

Barker Code Length	Barker Code	Sidelobe Level
1	[–1]	0 dB
2	[–1; 1]	–6 dB
3	[–1; –1; 1]	–9.5 dB
4	[–1; –1; 1; –1]	–12 dB
5	[–1; –1; –1; 1; –1]	–14 dB
7	[–1; –1; –1; 1; 1; –1; 1]	–16.9 dB
11	[–1; –1; –1; 1; 1; 1; –1; 1; 1; –1; 1]	–20.8 dB
13	[–1; –1; –1; –1; –1; 1; 1; –1; –1; 1; –1; 1; –1]	–22.3 dB

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Barker Code Generator block version available before R2015b.

Existing models automatically update to load the Barker Code Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Does not support integer only code generation.

See Also

Blocks

OVSF Code Generator | PN Sequence Generator | Walsh Code Generator

Objects

`comm.BarkerCode`

Introduced before R2006a

Baseband File Reader

Read baseband signals from file
Library: Communications Toolbox / Comm Sources

Description

The Baseband File Reader block reads a signal from a baseband file. A baseband file is a specific type of binary file written by the Baseband File Writer block. Baseband signals are typically downconverted from a nonzero center frequency to 0 Hz. The block automatically reads the sample rate, center frequency, number of channels, and any descriptive data.

Input/Output Ports

Output

`Data` — Baseband signal
scalar | vector | matrix

Baseband signal, returned as a scalar, vector, or matrix. The signal is read from the file specified by the **Baseband file name** parameter. The sample time is either inherited from the file or can be set by the **Sample Time (s)** parameter.

Data Types: `double`

`EOF` — End-of-file indicator
logical scalar

End-of-file indicator, returned as a logical scalar. The output is `true` when the **Repeatedly read the file** parameter is `false` and the entire file has been read. To enable this port, select the **Output end-of-file indicator** parameter.

Parameters

`Baseband file name` — Name of file from which data is read
`example.bb` (default) | character vector

Specify the name of the baseband file as a character vector.

Click **Browse** to locate the baseband file you want to read. Click **File Info** to display this information:

- File name
- Sample rate
- Center frequency
- Number of samples

- Number of channels
- Data type
- Any metadata fields

Data Types: `char`

`Inherit sample time from file` — Select source of sample time

`on` (default) | `off`

Select this check box to inherit the sample time from the file specified by **Baseband file name**.

`Sample time (s)` — Block sample time

`1` (default) | positive scalar

Specify the block sample time in seconds as a positive scalar. To enable this parameter, clear the **Inherit sample time from file** check box.

`Samples per frame` — Number of samples in one frame

`100` (default) | positive integer scalar

Specify the samples per frame as a positive integer scalar.

`Repeatedly read the file` — Continuously loop data from file

`off` (default) | `on`

Select this check box to repeatedly read the contents of the baseband file. When the end of the file is reached:

- The block outputs zeros, if the **Repeatedly read the file** parameter is not selected (`off`).
- The block outputs samples from the beginning of the file, if the **Repeatedly read the file** parameter is selected (`on`).

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Baseband File Writer

Introduced in R2016b

Baseband File Writer

Write baseband signals to file
Library: Communications Toolbox / Comm Sinks

Description

The Baseband File Writer block writes a baseband signal to a specific type of binary file. Baseband signals are typically down-converted from a nonzero center frequency to 0 Hz. Sample rate, which is determined by the input signal sample time and frame size, and center frequency are saved when the signal is written to a file.

Input/Output Ports

Input

`Port_1` — Baseband signal

scalar | vector | matrix

This port accepts a baseband signal to be saved under the filename specified by the **Baseband file name** parameter. The saved signal is always complex.

Data Types: `single` | `double`

Parameters

`Baseband file name` — Name of file in which data is saved

`untitled.bb` (default) | character vector

Specify the name of the baseband file as a character vector.

To specify the location where the file is saved, click **Browse**.

`Center frequency (Hz)` — Center frequency of the baseband signal

`1e8` (default) | nonnegative scalar

Specify the center frequency in Hz as a nonnegative scalar.

`Metadata in a structure` — Data describing the baseband signal

`struct()` (default) | structure

Specify data describing the baseband signal as a structure. If the signal has no descriptive data, this parameter is an empty structure. The structure can contain any number of fields. Field names have no restrictions, but the field values must be numeric, logical, or character data types having any dimension.

`Number of latest samples to write` — Number of samples to write
`inf` (default) | positive scalar

Specify the number to write. If this parameter is `inf`, all samples are saved. Otherwise, only the last *N* samples are saved, where *N* is specified by this parameter.

`Simulate using` — Select simulation mode
`Code generation` (default) | `Interpreted execution`

`Code generation`

> On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

`Interpreted execution`

> Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Block Characteristics

Data Types	`double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Tips

- The Baseband File Writer block writes baseband signals to uncompressed binary files. To share these files, you can compress them to a zip file using the `zip` function. For more information, see “Create and Extract from Zip Archives”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Baseband File Reader

Introduced in R2016b

Baseband PLL

Implement baseband phase-locked loop

Library

Components sublibrary of Synchronization

Description

The Baseband PLL (phase-locked loop) block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. Unlike the Phase-Locked Loop block, this block uses a baseband method and does not depend on a carrier frequency.

This PLL has these three components:

- An integrator used as a phase detector.
- A filter. You specify the filter's transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, you can use the Signal Processing Toolbox functions `cheby1`, and `cheby2`. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```
- A voltage-controlled oscillator (VCO). You specify the sensitivity of the VCO signal to its input using the **VCO input sensitivity** parameter. This parameter, measured in Hertz per volt, is a scale factor that determines how much the VCO shifts from its quiescent frequency.

This block accepts a sample-based scalar signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

This model is nonlinear; for a linearized version, use the Linearized Baseband PLL block.

Parameters

Lowpass filter numerator

The numerator of the lowpass filter's transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

Lowpass filter denominator

The denominator of the lowpass filter's transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

VCO input sensitivity (Hz/V)

This value scales the input to the VCO and, consequently, the shift from the VCO's quiescent frequency.

References

For more information about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization” in *Communications Toolbox User*'*s Guide*.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

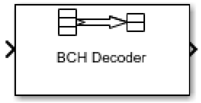
Linearized Baseband PLL | Phase-Locked Loop

Introduced before R2006a

BCH Decoder

Decode BCH code to recover binary vector data

Library: Communications Toolbox / Error Detection and Correction / Block



Description

The BCH Decoder block recovers a binary message vector from a binary BCH codeword vector. For proper decoding, the **Codeword length, N** and **Message length, K** parameter values in this block must match the parameters in the corresponding BCH Encoder block. The full-length values of *N* and *K* must produce a valid narrow-sense BCH code.

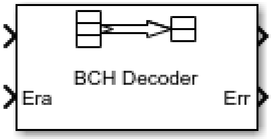
If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 5-38.

See “Tips” on page 5-40 for information about valid *N* values, valid (*N*,*K*) pairs, and error-correcting capabilities for a given BCH code.

If decoding fails, the message portion of the decoder input is returned unchanged as the decoder output.

The sample times of all input and output signals are equal.

This icon shows optional ports.



Ports

Input

In — Encoded message
binary column vector

Encoded message, specified as a binary column vector. The encoded message is a BCH code with message length *K* and codeword length (*N* – number of punctures).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Era — Erasure vector
binary column vector

Erasure vector, specified as a binary column vector that is the same length as `In`. Erasure values of 1 correspond to erased bits in the same position in the codeword. Values of 0 correspond to bits that are not erased.

Dependencies

To enable this port, select **Enable erasures input port**.

Data Types: `double` | `Boolean`

Output

`Out` — Decoded message

binary column vector

Decoded message, returned as a binary column vector input signal with an integer multiple of **Message length, K** elements or **Shortened message length, S** elements if the code is shortened. Each group of input elements represents one codeword to decode. The input and output signal lengths are listed in the "Input and Output Signal Length in BCH Blocks" on page 5-38 table.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

`Err` — Decoding errors

integer vector

Decoding errors, returned as an integer vector that indicates the number of errors detected during decoding of the codeword. A negative integer indicates that the block detected more errors than it could correct by using the coding scheme.

Dependencies

To enable this port, select **Output number of corrected errors**.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

For more information, see "Supported Data Types" on page 5-40.

Parameters

`Codeword length, N` — Codeword length

15 (default) | integer

Codeword length, specified as an integer of the form $N = 2^M–1$, where M is an integer from 3 through 16. For more information, see "Tips" on page 5-40.

`Message length, K` — Message length

5 (default) | integer

Message length, specified as an integer. The (`N`, `K`) pair must produce a narrow-sense BCH code.

`Shortened message length, S` — Shortened message length

5 (default) | integer

Shortened message length, specified as an integer. When you specify this parameter, provide full-length N and K values to specify the (N, K) code that is shortened to an ($N–K+S$, S) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial, specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.
- A binary Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'`, which is equivalent to `bchgenpoly(15,5)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^4 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. It is a polynomial of order *M* that defines the finite Galois field GF(2), specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^4 + X + 1'`, which is the primitive polynomial used for a (15,5) code, `de2bi(primpoly(4,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Disable generator polynomial checking` — Option to disable generator polynomial checking

on (default) | off

Select this parameter to disable generator polynomial check.

Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the specified generator polynomial, where *N* represents the full codeword length. For larger codes, disabling the check speeds up the simulation process.

Tip Always run the check at least once before disabling this feature.

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Puncture vector` — Puncture vector

`[ones(8,1); zeros(2,1)]` (default) | column vector

Puncture vector, specified as a binary column vector of length *N*–*K*. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 5-39.

Dependencies

To enable this parameter, select **Puncture code**.

`Enable erasures input port` — Option to enable erasures input port

off (default) | on

Selecting this check box enables the erasures port, `Era`.

Through the port, you can input a binary column vector that is 1/*M* times as long as the codeword input.

Erasure values of `1` correspond to erased symbols in the same position in the bit-packed codeword. Values of `0` correspond to nonerased symbols. For more information, see “Puncturing and Erasures” on page 5-39.

`Output number of corrected errors` — Option to enable port to output number of corrected errors

off (default) | on

Selecting this check box enables an additional output port, `Err`, which indicates the number of errors the block corrected in the input codeword.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Input and Output Signal Length in BCH Blocks

This table shows how to compute the input and output signal lengths for the BCH encoder and decoder blocks.

The notation `y = c * x` denotes that `y` is an integer multiple of `x`.

Specify Shortened Message Length, S	BCH Encoder	BCH Decoder
off	**Input Length:** *c * K* **Output Length:** *c * (N – P)*	**Input Length:** *c * (N – P)* **Output Length:** *c * K* **Erasures Length:** *c * (N – P)*
on	**Input Length:** *c * S* **Output Length:** *c * (N - K + S - P)*	**Input Length:** *c * (N - K + S - P)* **Output Length:** *c * S* **Erasures Length:** *c * (N - K + S - P)*

- *N* is the codeword length
- *K* is the message length
- *S* is the shortened message length
- *P* is the number of punctures value, and is equal to the number of zeros in the puncture vector.

Puncturing and Erasures

1s and 0s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- 1 means that the data symbol is passed through the block unaltered.
- 0 means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- 1 means that the data symbol is to be replaced with an erasure symbol.
- 0 means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Encoder — Encodes data using BCH algorithm.

Tips

- To generate the list of valid (`N`,`K`) pairs along with the corresponding values of the error-correction capability, run `bchnumerr(N)`.
- Valid values for `N` = 2^M–1, where M is an integer from 3 through 16. The maximum allowable value of `N` is 65,535.

Algorithms

This block implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

References

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

[2] Berlekamp, Elwyn R. *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BCH Encoder

Objects

`comm.BCHDecoder`

Functions

`bchdec` | `bchgenpoly` | `primpoly`

Topics

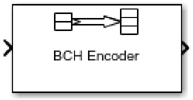
"Block Codes"

Introduced before R2006a

BCH Encoder

Create BCH code from binary vector data

Library: Communications Toolbox / Error Detection and Correction / Block



Description

The BCH Encoder block creates a BCH code with message length *K* and codeword length (*N* – number of punctures).

If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 5-44.

See “Tips” on page 5-45 for information about valid *N* values, valid (*N,K*) pairs, and error-correcting capabilities for a given BCH code.

Ports

Input

`In` — Message to encode
binary column vector

Message to encode, specified as a binary column vector input signal with an integer multiple of **Message length, K** elements or **Shortened message length, S** elements if the code is shortened. Each group of input elements represents one message word to encode. The input and output signal lengths are listed in “Input and Output Signal Length in BCH Blocks” on page 5-44.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Out` — Encoded message
binary column vector

Encoded message, returned as a binary column vector. The encoded message is a BCH code with message length *K* and codeword length (*N* – number of punctures).

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

For more information, see “Supported Data Types” on page 5-45.

Parameters

`Codeword length, N` — Codeword length
15 (default) | integer

Codeword length, specified as an integer of the form $N = 2^M–1$, where M is an integer from 3 through 16. For more information, see “Tips” on page 5-45.

`Message length, K` — Message length

5 (default) | integer

Message length, specified as an integer. The (`N`, `K`) pair must produce a narrow-sense BCH code.

`Shortened message length, S` — Shortened message length

5 (default) | integer

Shortened message length, specified as an integer. When you specify this parameter, provide full-length N and K values to specify the (N, K) code that is shortened to an ($N–K+S$, S) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial, specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.
- A binary Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^10 + X^8 + X^5 + X^4 + X^2 + X + 1'`, which is equivalent to `bchgenpoly(15,5)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^4 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. It is a polynomial of order M that defines the finite Galois field GF(2), specified as one of the following:

- A polynomial character vector — For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power.

Example: `'X^4 + X + 1'`, which is the primitive polynomial used for a (15,5) code, `de2bi(primpoly(4,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Disable generator polynomial checking` — Option to disable generator polynomial checking

on (default) | off

Select this parameter to disable generator polynomial check.

Each time a model initializes, the block performs a polynomial check. This check verifies that $X^N + 1$ is divisible by the specified generator polynomial, where N represents the full codeword length. For larger codes, disabling the check speeds up the simulation process.

Tip Always run the check at least once before disabling this feature.

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Puncture vector` — Puncture vector

`[ones(8,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as a binary column vector of length $N–K$. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Shortening, Puncturing, and Erasures”.

Note `1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors. For an erasure vector, `1` means that the data symbol is to be replaced with an erasure symbol, and `0` means that the data symbol is passed through the block unaltered. This convention applies to both the encoder and the decoder.

Dependencies

To enable this parameter, select **Puncture code**.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Input and Output Signal Length in BCH Blocks

This table shows how to compute the input and output signal lengths for the BCH encoder and decoder blocks.

The notation `y = c * x` denotes that `y` is an integer multiple of `x`.

Specify Shortened Message Length, S	BCH Encoder	BCH Decoder
off	**Input Length:** *c * K* **Output Length:** *c * (N – P)*	**Input Length:** *c * (N – P)* **Output Length:** *c * K* **Erasures Length:** *c * (N – P)*
on	**Input Length:** *c * S* **Output Length:** *c * (N - K + S - P)*	**Input Length:** *c * (N - K + S - P)* **Output Length:** *c * S* **`Erasures Length:`** *c * (N - K + S - P)*

- *N* is the codeword length
- *K* is the message length
- *S* is the shortened message length
- *P* is the number of punctures value, and is equal to the number of zeros in the puncture vector.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

BCH Decoder — Decodes BCH encoded data.

Tips

- To generate the list of valid (`N`,`K`) pairs along with the corresponding values of the error-correction capability, run `bchnumerr(N)`.

- Valid values for `N` = 2^M–1, where *M* is an integer from 3 through 16. The maximum allowable value of `N` is 65,535.

Algorithms

This block implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BCH Decoder

Objects

`comm.BCHEncoder`

Functions

`bchenc` | `bchgenpoly` | `primpoly`

Topics

“Block Codes”

Introduced before R2006a

Bernoulli Binary Generator

Generate Bernoulli-distributed random binary numbers

Library: Communications Toolbox / Comm Sources / Random Data Sources

Description

The Bernoulli Binary Generator block generates random binary numbers using a Bernoulli distribution. Use this block to generate random data bits to simulate digital communication systems and obtain performance metrics such as bit error rate. The Bernoulli distribution with parameter *p* produces zero with probability *p* and one with probability *1-p*. The Bernoulli distribution has mean value *1-p* and variance *p*(*1-p*). The **Probability of zero** parameter specifies *p* and can be any real number in range [0, 1].

The output signal can be a column or row vector, two-dimensional matrix, or scalar. The number of rows in the output signal corresponds to the number of samples in one frame and is set by the **Samples per frame** parameter. The number of columns in the output signal corresponds to the number of channels and is set by the number of elements in the **Probability of zero** parameter. For more details, see "Sources and Sinks" in *Communications Toolbox User's Guide*

Ports

Output

`Out` — Output data signal
scalar | vector | matrix

Output data signal, returned as a scalar, vector, or matrix.

Data Types: `double`

Parameters

`Probability of zero` — Probability of generating zero at output
`0.5` (default) | integer in the range [0, 1] | vector of integers in the range [0, 1]

Probability of zero must be in the range of [0, 1]. The number of elements in the **Probability of zero** parameter corresponds to the number of independent channels output from the block. The Bernoulli distribution with parameter *p* produces zero with probability *p* and one with probability *1-p*.

`Source of initial seed` — Source of initial seed for random number generator
`Auto` (default) | `Parameter`

Select `Parameter` to use the **Initial seed** parameter to specify the initial seed for the random number generator.

Note When the **Source of initial seed** parameter is set to `Auto` and the **Simulate using** parameter is set to `Code generation`, the random number generator uses an initial seed of zero. In this case, the block generates the same random numbers each time it is started. To ensure that the model uses different initial seeds, set **Simulate using** parameter to `Interpreted execution`. If you run `Interpreted execution` in `Rapid accelerator` mode, then the model behaves the same as `Code generation` mode.

Dependencies

Select `Auto` for the block to use the global random number stream as the initial seed. For more information, see “Managing the Global Stream Using RandStream” and “Random Number Generators”.

`Initial seed` — Initial seed for random number generator

nonnegative scalar

If you set the **Initial seed** parameter to a constant value, then the resulting sequence is repeatable.

Dependencies

To enable this parameter, set the **Source of initial seed** to `Parameter`.

`Sample time` — Sample time of output signal

1 (default) | -1 | positive scalar

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-49.

`Samples per frame` — Samples per frame of output signal

1 (default) | positive scalar

Samples per frame in one channel of the output signal, specified as a positive integer. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-49.

`Output data type` — Data type of output signal

`double` (default) | `boolean`

Select the data type for the output signal.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Bernoulli Binary Generator block update supported in Upgrade Advisor

Behavior changed in R2020a

Starting in R2020a, Bernoulli Binary Generator block allows you to use the Upgrade Advisor. You can update to the block version announced in R2015b or keep the block version available before R2015b.

- Use the Upgrade Advisor to update existing models that include the Bernoulli Binary Generator block.
- Behavior of the random number generator is changed. The statistics are improved. For more information, see "Source blocks output frames of contiguous time samples but do not use the frame attribute" in the R2015b Release Notes.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Binary Symmetric Channel | Random Integer Generator

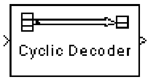
Functions

`rand` | `randi`

Introduced before R2006a

Binary Cyclic Decoder

Decode systematic cyclic code to recover binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Binary Cyclic Decoder block recovers a message vector from a codeword vector of a binary systematic cyclic code. For proper decoding, the parameter values in this block should match those in the corresponding Binary Cyclic Encoder block.

This block accepts a column vector input signal containing *N* elements, where *N* is the codeword length. The output signal is a column vector containing *K* elements, where *K* is the message length of the cyclic code.

You can determine the systematic cyclic coding scheme in one of two ways:

- To create an [N,K] code, enter *N* and *K* as the first and second dialog parameters, respectively. The block computes an appropriate generator polynomial, namely, `cyclpoly(N,K,'min')`.
- To create a code with codeword length N and a particular degree-(N-K) binary *generator polynomial*, enter *N* as the first parameter and a polynomial character vector or a binary vector as the second parameter. The vector represents the generator polynomial by listing its coefficients in order of ascending exponents. You can create cyclic generator polynomials using the Communications Toolbox `cyclpoly` function.

For information about the data types each block port supports, see the “Supported Data Type” on page 5-51 table on this page.

Parameters

Codeword length N

The codeword length *N*, which is also the input vector length.

Message length K, or generator polynomial

Either the message length, which is also the input vector length, a polynomial character vector, or a binary vector that represents the generator polynomial for the code.

Supported Data Type

Port	Supported Data Types
In	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point
Out	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point

Pair Block

Binary Cyclic Encoder

See Also

`cyclpoly`

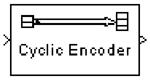
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

Binary Cyclic Encoder

Create systematic cyclic code from binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Binary Cyclic Encoder block creates a systematic cyclic code with message length *K* and codeword length *N*.

This block accepts a column vector input signal containing *K* elements. The output signal is a column vector containing *N* elements.

You can determine the systematic cyclic coding scheme in one of two ways:

- To create an [N,K] code, enter N and K as the first and second dialog parameters, respectively. The block computes an appropriate generator polynomial, namely, `cyclpoly(N,K,'min')`.
- To create a code with codeword length *N* and a particular degree-(N-K) binary *generator polynomial*, enter *N* as the first parameter and a polynomial character vector or a binary vector as the second parameter. The vector represents the generator polynomial by listing its coefficients in order of ascending exponents. You can create cyclic generator polynomials using the Communications Toolbox `cyclpoly` function.

For information about the data types each block port supports, see the “Supported Data Type” on page 5-53 table on this page.

Parameters

Codeword length N

The codeword length, which is also the output vector length.

Message length K, or generator polynomial

Either the message length, which is also the input vector length, a polynomial character vector, or a binary vector that represents the generator polynomial for the code.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Cyclic Decoder

See Also

`cyclpoly` (in the Communications Toolbox documentation)

Extended Capabilities

C/C++ Code Generation

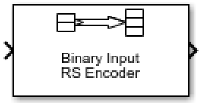
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

Binary-Input RS Encoder

Create Reed-Solomon code from binary vector data

Library: Communications Toolbox / Error Detection and Correction / Block

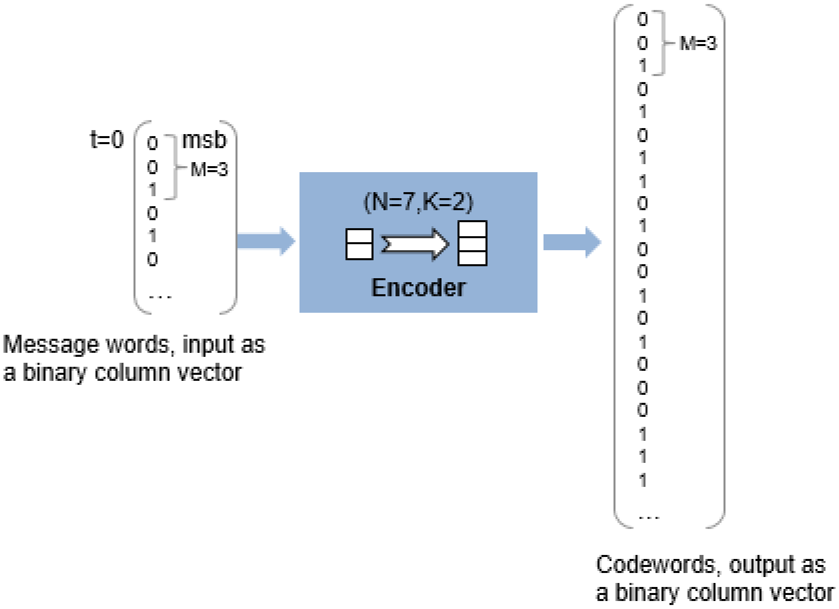


Description

The Binary-Input RS Encoder block creates a Reed-Solomon code.

The symbols for the code are binary sequences of length *M*, corresponding to elements of the Galois field *GF*(2^M). The first bit in each symbol is the most significant bit.

Suppose M = 3, N = 2^3-1 = 7, and K = 2. Then a message is a vector of length 2 whose entries are integers between 0 and 7. A corresponding codeword is a vector of length 7 whose entries are integers between 0 and 7. The following figure illustrates possible input and output signals to this block when codeword length *N*=7 and message word length *K*=2. Since *N*=2^M–1, when *N*=7, the symbol length, *M*=3.



Each input message word is a binary vector of length 6, that represents 2 three-bit integers. Each corresponding output codeword is a binary vector of length 21 that represents 7 three-bit integers. For more information, see “Input and Output Signal Length in RS Blocks” on page 5-57.

Ports

Input

`In` — Message
binary column vector

Message in bits, specified as one of the following:

- When there is no message shortening, a (N_C×K×M)-by-1 binary column vector.
- When there is message shortening, a (N_C×S×M)-by-1 binary column vector.

N_C is the number of message words, K is the **Message length K (symbols)**, M is the number of bits per symbol, and S is the **Shortened message length S (symbols)**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-57.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

Output

`Out` — Reed-Solomon codeword
binary column vector

Reed-Solomon codeword in bits, returned as an (N_C×(N – K + S – P)×M)-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-57.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

For more information, see “Supported Data Types” on page 5-59.

Parameters

`Codeword length N (symbols)` — Codeword length
7 (default) | integer

Codeword length in symbols, specified as an integer.

For more information, see “Restrictions on the M and the Codeword Length N” on page 5-58 and “Input and Output Signal Length in RS Blocks” on page 5-57.

`Message length K (symbols)` — Message word length
3 (default) | integer

Message word length in symbols, specified as an integer in the range [1, N–2], where N is the codeword length.

`Shortened message length S (symbols)` — Shortened message word length

3 (default) | integer

Shortened message word length in symbols, specified as an integer, such that $S \leq K$. When **Shortened message length S (symbols) < Message length K (symbols)**, the Reed-Solomon code is shortened.

You still specify *N* and *K* values for the full-length (*N*, *K*) code but the decoding is shortened to an (*N*–*K*+*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values from 0 to 2^M–1, in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see "Character Representation of Polynomials".
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see "Specify the Generator Polynomial" on page 5-58.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see "Character Representation of Polynomials".
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see "Restrictions on the M and the Codeword Length N" on page 5-58.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector
`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 5-59.

Dependencies

To enable this parameter, select **Puncture code**.

`Output data type` — Output type of the block
`Same as input` (default) | `boolean` | `double`

Output type of the block, specified as `Same as input`, `boolean`, or `double`.

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[a] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

[a] ufix(1) only.

More About

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, *K*, or shortened message word length, *S*. The codeword length is $N - K + S - P$, where *N* is the full codeword length and *P* is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N - P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that *y* is an integer multiple of *x*.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Input RS Encoder	**Input Length (bits):**
	$N_C \times K \times M$
	Output Length (bits):
	$N_C \times (N–P) \times M$
		Input Length (bits):
		$N_C \times S \times M$
		Output Length (bits):
		$N_C \times (N–K+S–P) \times M$

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Output RS Decoder	**Input Length (bits):**
	$N_C \times (N–P) \times M$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (bits):
	$N_C \times K \times M$
		Input Length (bits):
		$N_C \times (N–K+S–P) \times M$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (bits):
		$N_C \times S \times M$

- N is the codeword length.
- K is the message word length.
- S is the shortened message word length.
- N_C is the number of codewords (and message words).
- P is the number of punctures per codeword, and is equal to the number of zeros in the puncture vector.
- M is the degree of the primitive polynomial. Each group of M bits represents an integer between `0` and `2`^{M}`–1` that belongs to the finite Galois field `GF(2`^{M}`)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Also, see “Restrictions on the M and the Codeword Length N” on page 5-58.

Restrictions on the M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, N, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, M, are from 3 to 16. The valid values for N in this case are from 7 to 2^M–1. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 2^M-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of $GF(2^M)$ represented in integer format. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})\ldots(x+\alpha^{b+N-K-1})$$

α is the primitive element of the Galois field over which the input message is defined, and b is an integer.

If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to *b*=1, for Reed-Solomon encoding. You can display the default generator polynomial by running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))

Pair Block

Binary-Output RS Decoder

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Binary-Output RS Decoder | Integer-Input RS Encoder

Objects

`comm.RSEncoder`

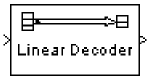
Functions

`primpoly` | `rsenc` | `rsgenpoly`

Introduced before R2006a

Binary Linear Decoder

Decode linear block code to recover binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Binary Linear Decoder block recovers a binary message vector from a binary codeword vector of a linear block code.

The **Generator matrix** parameter is the generator matrix for the block code. For proper decoding, this should match the **Generator matrix** parameter in the corresponding Binary Linear Encoder block. If *N* is the codeword length of the code, then **Generator matrix** must have *N* columns. If *K* is the message length of the code, then the **Generator matrix** parameter must have *K* rows.

This block accepts a column vector input signal containing *N* elements. This block outputs a column vector with a length of *K* elements.

The decoder tries to correct errors, using the **Decoding table** parameter. If **Decoding table** is the scalar 0, then the block defaults to the table produced by the Communications Toolbox function `syndtable`. Otherwise, **Decoding table** must be a 2^{N-K}-by-N binary matrix. The *r*th row of this matrix is the correction vector 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.

For information about the data types each block port supports, see the “Supported Data Type” on page 5-62 table on this page.

Parameters

Generator matrix

Generator matrix for the code; same as in Binary Linear Encoder block.

Decoding table

Either a 2^{N-K}-by-N matrix that lists correction vectors for each codeword's syndrome; or the scalar 0, in which case the block defaults to the table corresponding to the **Generator matrix** parameter.

Supported Data Type

Port	Supported Data Types
In	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point
Out	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point

Pair Block

Binary Linear Encoder

Extended Capabilities

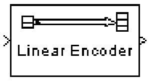
C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

Binary Linear Encoder

Create linear block code from binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Binary Linear Encoder block creates a binary linear block code using a generator matrix that you specify. If *K* is the message length of the code, then the **Generator matrix** parameter must have *K* rows. If *N* is the codeword length of the code, then **Generator matrix** must have *N* columns.

This block accepts a column vector input signal containing *K* elements. This block outputs a column vector with a length of *N* elements. For information about the data types each block port supports, see "Supported Data Type" on page 5-63.

Parameters

Generator matrix

A *K*-by-*N* matrix, where *K* is the message length and *N* is the codeword length.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Binary Linear Decoder

Extended Capabilities

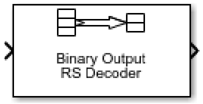
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

Binary-Output RS Decoder

Decode Reed-Solomon code to recover binary vector data

Library: Communications Toolbox / Error Detection and Correction / Block

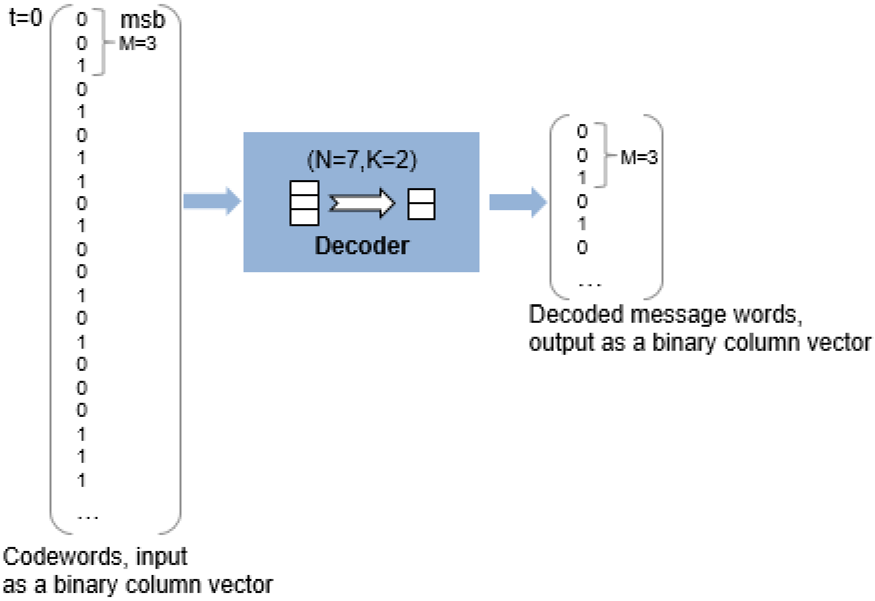


Description

The Binary-Output RS Decoder block recovers a binary message vector from a binary Reed-Solomon codeword vector. For proper decoding, the parameter values in this block must match parameter values in the corresponding Binary-Input RS Encoder block.

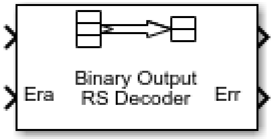
The symbols for the code are binary sequences of length *M*, corresponding to elements of the Galois field $GF(2^M)$. The first bit in each symbol is the most significant bit.

This figure shows the decoder input-output word length for codeword length $N=7$ and message word length $K=2$. Since $N=2^M-1$, when $N=7$, the symbol length, $M=3$.



Each input codeword is a binary vector of length 21 that represents 7 three-bit integers. Each corresponding output message word is a binary vector of length 6, that represents 2 three-bit integers. For more information, see “Input and Output Signal Length in RS Blocks” on page 5-69.

This icon shows all ports, including optional ones:



Ports

Input

`In` — Reed-Solomon codeword
binary column vector

Reed-Solomon codeword in bits, specified as an (N_C×(N – K + S – P)×M)-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

For more information, see "Input and Output Signal Length in RS Blocks" on page 5-69.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

`Era` — Erasure vector
binary column vector

Erasure vector in symbols, specified as an (N_C×(N – K + S – P))-by-1 binary column vector. N_C is the number of codewords, N is the **Codeword length N (symbols)**, K is the **Message length K (symbols)**, S is the **Shortened message length S (symbols)**, P is the number of punctures per codeword, and M is the number of bits per symbol.

Erasure values of 1 correspond to erased bits in the same position in the codeword. Values of 0 correspond to bits that are not erased. For more information, see "Puncturing and Erasures" on page 5-71.

Dependencies

To enable this port, select **Enable erasures input port**.

Data Types: `double` | `Boolean`

Output

`Out` — Decoded message
binary column vector

Decoded message in bits, returned as one of the following:

- When there is no message shortening, a (N_C×K×M)-by-1 binary column vector.
- When there is message shortening, a (N_C×S×M)-by-1 binary column vector.

N_C is the number of message words, K is the **Message length K (symbols)**, M is the number of bits per symbol, and S is the **Shortened message length S (symbols)**.

Note The number of decoded message words equals the number of codewords.

For more information, see "Input and Output Signal Length in RS Blocks" on page 5-69.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean` | `ufix(1)`

`Err` — Decoding errors

integer vector

Symbol decoding errors, returned as an integer vector with N_C elements, where N_C is the number of codewords. This port indicates the number of symbol errors detected during decoding of each codeword. A negative integer indicates that the block detected more errors than it could correct by using the specified coding scheme.

Note An (*N*,*K*) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (not bit errors) in each codeword. When a received codeword contains more than (*N*-*K*)/2 symbol errors, a decoding failure occurs.

Dependencies

To enable this port, select **Output number of corrected symbol errors**.

Data Types: `double`

For more information, see "Supported Data Types" on page 5-71.

Parameters

`Codeword length N (symbols)` — Codeword length

7 (default) | integer

Codeword length in symbols, specified as an integer.

For more information, see "Restrictions on M and Codeword Length N" on page 5-70 and "Input and Output Signal Length in RS Blocks" on page 5-69.

`Message length K (symbols)` — Message word length

3 (default) | integer

Message word length in symbols, specified as an integer in the range [1, *N*–2], where *N* is the codeword length.

`Shortened message length S (symbols)` — Shortened message word length

3 (default) | integer

Shortened message word length in symbols, specified as an integer, such that $S \leq K$. When **Shortened message length S (symbols) < Message length K (symbols)**, the Reed-Solomon code is shortened.

You still specify *N* and *K* values for the full-length (*N*, *K*) code but the decoding is shortened to an (*N*–*K*+*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values from 0 to 2^M–1, in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see “Specify the Generator Polynomial” on page 5-70.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Restrictions on M and Codeword Length N” on page 5-70.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 5-71.

Dependencies

To enable this parameter, select **Punctured code**.

`Enable erasures input port` — Enable erasures input port

off (default) | on

Selecting this check box enables the erasures port, **Era**. For more information, see “Puncturing and Erasures” on page 5-71.

`Output number of corrected symbol errors` — Enable port to output number of corrected symbol errors

off (default) | on

Selecting this check box enables an additional output port, **Err**, which indicates the number of symbol errors the block corrected in the input codeword.

`Output data type` — Output type of the block

`Same as input` (default) | `boolean` | `double`

Output type of the block, specified as `Same as input`, `boolean`, or `double`.

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[a] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

[a] ufix(1) only.

More About

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, *K*, or shortened message word length, *S*. The codeword length is $N – K + S – P$, where *N* is the full codeword length and *P* is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N – P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that *y* is an integer multiple of *x*.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Binary-Input RS Encoder	**Input Length (bits):**
	$N_C \times K \times M$
	Output Length (bits):
	$N_C \times (N-P) \times M$
		Input Length (bits):
		$N_C \times S \times M$
		Output Length (bits):
		$N_C \times (N-K+S-P) \times M$

Binary-Output RS Decoder	**Input Length (bits):**
	$N_C \times (N-P) \times M$
	Erasures Length (symbols):
	$N_C \times (N-P)$
	Output Length (bits):
	$N_C \times K \times M$
		Input Length (bits):
		$N_C \times (N-K+S-P) \times M$
		Erasures Length (symbols):
		$N_C \times (N-K+S-P)$
		Output Length (bits):
		$N_C \times S \times M$

- N is the codeword length.
- K is the message word length.
- S is the shortened message word length.
- N_C is the number of codewords (and message words).
- P is the number of punctures per codeword, and is equal to the number of zeros in the puncture vector.
- M is the degree of the primitive polynomial. Each group of M bits represents an integer between `0` and 2^M–`1` that belongs to the finite Galois field `GF(`2^M`)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Also, see “Restrictions on M and Codeword Length N” on page 5-70.

Restrictions on M and Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, N, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, M, are from 3 to 16. The valid values for N in this case are from 7 to 2^M–1. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 2^M-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of $GF(2^M)$ represented in integer format. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})\ldots(x+\alpha^{b+N-K-1})$$

α is the primitive element of the Galois field over which the input message is defined, and b is an integer.

If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to b=1, for Reed-Solomon encoding. You can display the default generator polynomial by running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))

Port	Supported Data Types
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • 1-bit unsigned integer (ufix(1))
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point

Pair Block

Binary-Input RS Encoder

Algorithms

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see "Algorithms for BCH and RS Errors-only Decoding".

References

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

[2] Berlekamp, Elwyn R. *Algebraic Coding Theory*. New York: McGraw-Hill, 1968.

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Binary-Input RS Encoder | Integer-Output RS Decoder

Objects

`comm.RSDecoder`

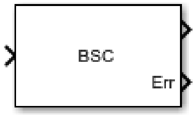
Functions

`primpoly` | `rsdec` | `rsgenpoly`

Introduced before R2006a

Binary Symmetric Channel

Introduce binary errors
Library: Communications Toolbox / Channels



Description

The Binary Symmetric Channel block introduces errors to the input signal transmitted through a binary symmetric channel. The errors are introduced based on the specified Error probability. For more information, see “Tips” on page 5-74.

Ports

Input

`Input` — Input signal
column vector | matrix

Input signal, specified as a column vector or an N_S-by-N_C matrix of `Boolean` values. N_S is the number of samples per channel. N_C is the number of independent data channels. For more information, see “Tips” on page 5-74.

Output

`Output` — Binary output signal
column vector | matrix

Binary output signal, returned as a column vector or matrix with the same dimensions as `Input`. The output signal is a version of the input signal that has been modified by introducing random errors based on the specified Error probability. To set the output data type, use Output data type.

`Err` — Error locations
column vector | matrix

Error locations, returned as a column vector or matrix with the same dimensions as `Input`. Element values in `Err` are `1` or `0`, where:

- `1` indicates that the corresponding element in `Output` has an error.
- `0` indicates that the corresponding element in `Output` does not have an error.

The data type of `Err` is the same as Output, as set by Output data type.

Dependencies

To enable this port, select Output error vector.

Parameters

Error probability — Probability of error occurrence

0.05 (default) | scalar

Probability of error occurrence for the input signal elements, specified as a scalar in the range [0,1]. The probability of error applies independently for each element.

Output error vector — Option to output error locations

on (default) | off

To enable the Err output port to the block, select this parameter.

Output data type — Output data type

double (default) | single | boolean

Select the output data type as double, single, or boolean. This parameter sets the output data type for both the **Output** and **Err** ports.

Initial seed — Initial seed

71 (default) | integer

Initial seed value for the random number generator used by the block, specified as an integer. The block uses the mt19937ar algorithm to generate uniformly distributed random numbers. For details about the mt19937ar algorithm, see “Choosing a Random Number Generator”.

Simulate using — Type of simulation to run

Code generation (default) | Interpreted execution

Type of simulation to run, specified as:

- Code generation –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- Interpreted execution –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In Interpreted execution mode, you can debug the source code of the block.

Block Characteristics

Data Types	Boolean | double | fixed point | integer | single
Multidimensional Signals	no
Variable-Size Signals	no

Tips

- When the input consists of not Boolean values, Binary Symmetric Channel converts zero-valued elements to 0 and converts nonzero-valued elements to 1.
- The Binary Symmetric Channel block creates and uses an independent RandStream to provide a random number stream for probability determination.

- To generate repeatable results, use the same Initial seed value.
- To generate independent probability statistics, set different Initial seed values for multichannel signals, multiple processing chains, or simulation runs.

Compatibility Considerations

Random Number Generation

Behavior changed in R2018b

To improve statistical properties, the Binary Symmetric Channel block uses the `mt19937ar` algorithm with `RandStream`. The Binary Symmetric Channel block accepts a single scalar value for the Initial seed parameter.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Bernoulli Binary Generator

Topics

"Design a Rate 2/3 Feedforward Encoder Using Simulink"

Introduced before R2006a

Bipolar to Unipolar Converter

Map bipolar signal into unipolar signal in range [0, M-1]

Library

Utility Blocks

Description

The Bipolar to Unipolar Converter block maps the bipolar input signal to a unipolar output signal. If the input consists of integers in the set {-*M*+1, -*M*+3, -*M*+5,..., *M*-1}, where *M* is the **M-ary number** parameter, then the output consists of integers between 0 and M-1. This block is only designed to work when the input value is within the set {-*M*+1, -*M*+3, -*M*+5,..., *M*-1}, where M is the **M-ary number** parameter. If the input value is outside of this set of integers the output may not be valid.

The table below shows how the block's mapping depends on the **Polarity** parameter.

Polarity Parameter Value	Output Corresponding to Input Value of k
`Positive`	(M-1+k)/2
`Negative`	(M-1-k)/2

Parameters

M-ary number

The number of symbols in the bipolar or unipolar alphabet.

Polarity

A value of `Positive` causes the block to maintain the relative ordering of symbols in the alphabets. A value of `Negative` causes the block to reverse the relative ordering of symbols in the alphabets.

Output Data Type

The type of bipolar signal produced at the block's output.

The block supports the following output data types:

- `Inherit via internal rule`
- `Same as input`
- `double`
- `int8`
- `uint8`
- `int16`

- `uint16`
- `int32`
- `uint32`
- `boolean`

When the parameter is set to its default setting, `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
- If the input data type is not floating-point:
  - Based on the **M-ary number** parameter, the output data type is the ideal unsigned integer output word length required to contain the range [0 M-1] and is computed as follows:

    ideal word length = ceil(log2(*M*))
  - The block sets the output data type to be an unsigned integer, based on the smallest word length (in bits) that can fit best the computed ideal word length.

Note The selections in the “Hardware Implementation Pane” (Simulink) pane pertaining to word length constraints do not affect how this block determines output data types.

Examples

If the input is [-3; -1; 1; 3], the **M-ary number** parameter is `4`, and the **Polarity** parameter is `Positive`, then the output is [0; 1; 2; 3]. Changing the **Polarity** parameter to `Negative` changes the output to [3; 2; 1; 0].

If the value for the **M-ary number** is 2^8 the block gives an output of uint8.

If the value for the **M-ary number** is 2^8+1 the block gives an output of uint16.

Pair Block

Unipolar to Bipolar Converter

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

Bit to Integer Converter

Map vector of bits to corresponding vector of integers
Library: Communications Toolbox / Utility Blocks

Description

The Bit to Integer Converter block maps groups of bits in the input vector to integers in the output vector.

If *M* is specified by the **Number of bits per integer(M)** parameter:

- For unsigned integers, the block maps each group of *M* bits to an integer in the range [0, (2^M – 1)]. As a result, the output vector length is 1/*M* times the input vector length.
- For signed integers, the block maps each group of *M* bits to an integer in the range [(–2^{M-1}), ($2^{M - 1}$ – 1)].

Ports

Input

`In` — Input signal

bit scalar | column vector of bits

Input signal, specified as a scalar or column vector of bits with a length that is a multiple of the value specified in the **Number of bits per integer(M)** parameter. The input must be bits with values of 0 or 1.

Data Types: `double`

Output

`Out` — Output signal

integer | integer column vector of integers

Output signal, returned as an integer or column vector of integers. The **After bit packing, treat resulting integer values as** parameter specifies whether input bits are treated as unsigned or signed.

- When the input bits are treated as unsigned, each integer output is in the range [0, (2^M – 1)].
- When the input bits are treated as signed, each integer output is in the range [(–2^{M-1}), ($2^{M - 1}$ – 1)].

Parameters

`Number of bits per integer(M)` — Number of bits per integer

3 (default) | integer in the range [1, 32]

Number of input bits mapped to each integer in the output, specified as an integer in the range [1, 32].

`Input bit order` — Input bit order

`MSB first` (default) | `LSB first`

Input bit order, specified as `'MSB first'` or `'LSB first'`.

- `'MSB first'` –– First bit of the input signal is the most significant bit (MSB).
- `'LSB first'` –– First bit of the input signal is the least significant bit (LSB).

`After bit packing, treat resulting integer values as` — Flag for signed integer values after bit packing

`Unsigned` (default) | `Signed`

Specify whether the resulting integer values are treated as signed or unsigned after bit packing. This parameter setting determines which **Output data type** selections are available.

`Output data type` — Output data type

`Inherit via internal rule` (default) | `Smallest integer` | `Same as input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

The **Output data type** options change depending on the desired signedness of the output.

If the output integers are `Signed`, you can choose from the following **Output data type** options:

- `Inherit via internal rule`
- `Smallest integer`
- `double`
- `single`
- `int8`
- `int16`
- `int32`

If the output integers are `Unsigned`, you can choose from the following options in addition to the `Signed` options:

- `Same as input`
- `uint8`
- `uint16`
- `uint32`

When you set the parameter to `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `double` or `single`), the output data type is the same as the input data type.
- If the input data type is not floating-point, the output data type is determined as if the parameter is set to `Smallest integer`.

When you set the parameter to `Smallest integer`, the block selects the output data type based on the settings used in the “Hardware Implementation Pane” (Simulink) of the Configuration Parameters dialog box.

- If you select `ASIC/FPGA` for the device vendor, the output data type is the smallest ideal integer or fixed-point data type, based on the setting for the **Number of bits per integer(M)** parameter.
- For all other device vendor selections, the output data type is the smallest available (signed or unsigned) integer word length that is large enough to fit the ideal minimum bit size.

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[ab] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

[a] Fixed-point inputs must be ufix(1).

[b] ufix(N) or sfix(N) when ASIC/FPGA is selected in the Hardware Implementation Pane and output data-type is set to either (a) Smallest integer or, (b) Inherit via internal rule and at the same time input is non floating-point.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Integer to Bit Converter

Functions

`bi2de` | `bin2dec`

Introduced before R2006a

BPSK Demodulator Baseband

Demodulate BPSK-modulated data

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM
Communications Toolbox HDL Support / Modulation / PM

Description

The BPSK Demodulator Baseband block demodulates a signal that was modulated using the binary phase shift keying method. The input is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal. The input signal must be a discrete-time complex signal. The block maps the points exp(jθ) and -exp(jθ) to 0 and 1, respectively, where θ is the **Phase offset** parameter.

Ports

Input

`In` — BPSK-modulated signal
scalar | vector | matrix

BPSK-modulated signal, specified as a scalar, vector, or matrix. When this input is a matrix, each column is treated as an independent channel.

Data Types: `double` | `single` | `fixed point`
Complex Number Support: Yes

Output

`Out` — Demodulated signal
scalar | vector

Demodulated signal, returned as a scalar or vector. If the output is a scalar, the value is an integer. If the output is a vector, it is an integer-valued or binary-valued vector.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Parameters

`Output type` — Output type
`Inherit via internal rule` (default) | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean`

When you set the **Decision type** parameter to `Hard decision`, you can set this parameter to `Inherit via internal rule`, `Smallest unsigned integer`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

When you set this parameter to `Inherit via internal rule`, the block inherits the output data type from the input port. If the input is a floating-point type (`single` or `double`) the output data type is the same as the input data type. If the input data type is fixed-point, the output data type will work as if this parameter is set to `Smallest unsigned integer`.

When you set this parameter to `Smallest unsigned integer`, the block selects the output data type based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If you select `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum one-bit size, that is., `ufix(1)`. For all other selections, the output data type is an unsigned integer with the smallest available word length large enough to fit one bit. This value usually corresponds to the size of a character (for example, `uint8`).

`Derotate factor` — Derotate factor

`Same word length as input` (default) | `Specify word length`

Derotate factor, specified as `Same word length as input` or `Specify word length`. This parameter applies only when the input is fixed-point and the **Phase offset** (rad) parameter is not a multiple of π/2.

`Decision type` — Decision type

`Hard decision` (default) | `Log-likelihood ratio` | `Approximate log-likelihood ratio`

Decision type used during demodulation, specified as `Hard decision`, `Log-likelihood ratio` or `Approximate log-likelihood ratio`. The output values when you select `Log-likelihood ratio` and `Approximate log-likelihood ratio` are of the same data type as the input values. For algorithm details, see “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the Communications Toolbox User's Guide for algorithm details.

`Noise variance source` — Noise variance source

`Dialog` (default) | `Port`

Noise variance source, specified as `Dialog` or `Port`.

Select `Dialog` to specify the noise variance using the **Noise variance** parameter. Select `Port`to enable the port to input the noise variance.

`Noise variance` — Noise Variance

`1` (default)

This parameter specifies the noise variance in the input signal. This parameter is tunable in normal mode, accelerator mode and rapid accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations, in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and yields:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and the signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Dependencies

To enable this parameter, set the **Noise variance source** parameter to `Dialog`.

`Phase offset (rad)` — Phase of zeroth point

`0` (default) | real-valued scalar

Phase of the zeroth point, specified as a real-valued scalar. Units are in radians.

Example: `pi/4`

`Output data type` — Data type of output

`Inherit via internal rule` (default) | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean`

Datatype of output, specified as one of these options.

- `Inherit via internal rule`
- `Smallest unsigned integer`
- `double`
- `single`
- `int8`
- `uint8`
- `int16`
- `uint16`
- `int32`
- `uint32`
- `boolean`

When you set the **Decision type** parameter to

- `Hard decision`, you can set this parameter to `Inherit via internal rule`, `Smallest unsigned integer`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`
- `Inherit via internal rule` (default), the block inherits the output data type from the input port. If the input is a floating-point type (`single` or `double`), the output data type is the same as the input data type. If the input data type is fixed-point, the output data type works as if you set this parameter to `Smallest unsigned integer`.
- `Smallest unsigned integer`, the block selects the output data type based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If you select `ASIC/FPGA` in the **Hardware Implementation** pane, the output data type is the ideal minimum one-bit size, that is, `ufix(1)`. For all other selections, the output data type is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a character (for example, `uint8`).
- `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the block inherits the output data type from the input (for example, if the input is of data type `double`, the output is also of data type `double`).

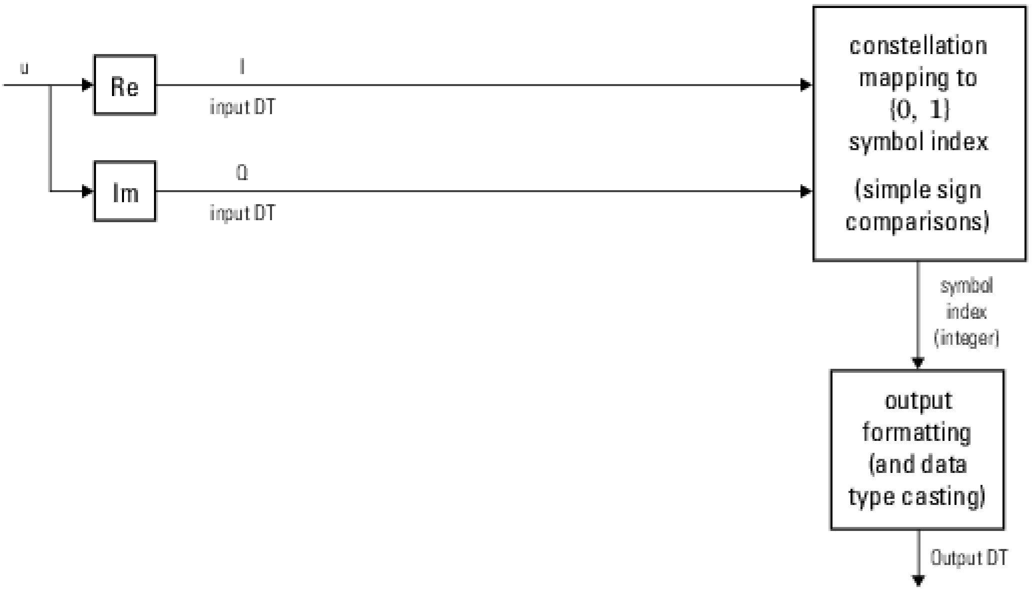
Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[a][b] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

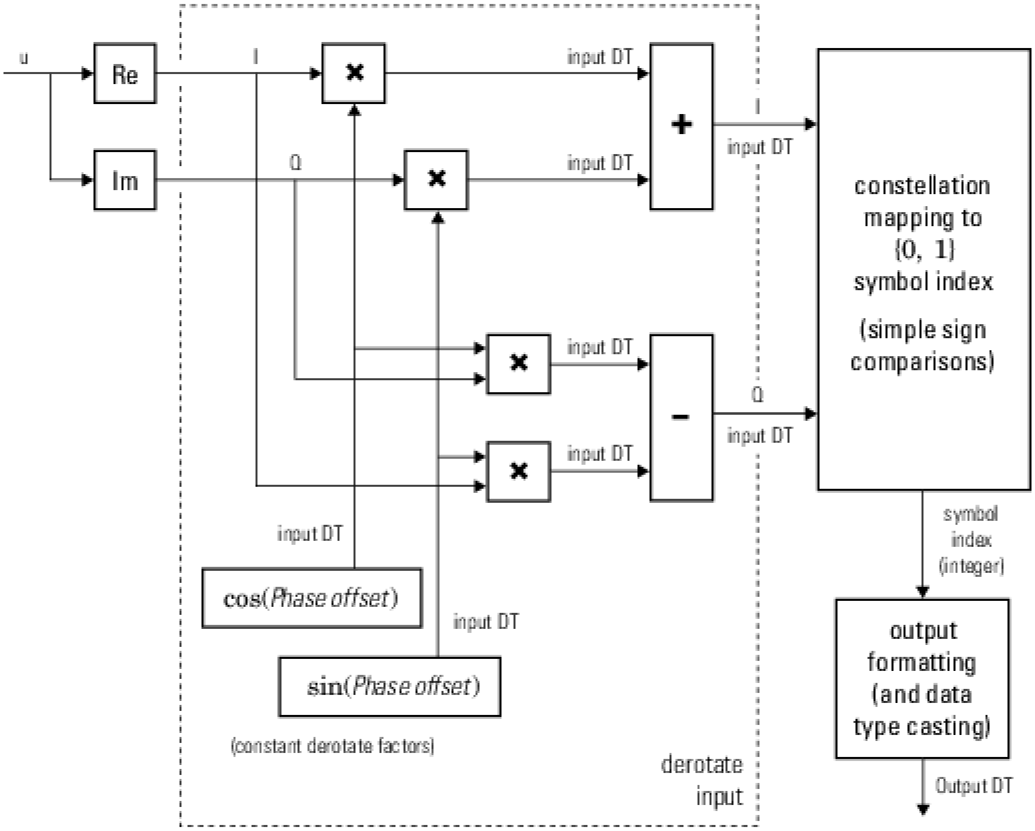
[a] Fixed-point inputs must be signed.

[b] ufix(1) only at the output when ASIC/FPGA is selected in the Hardware Implementation Pane.

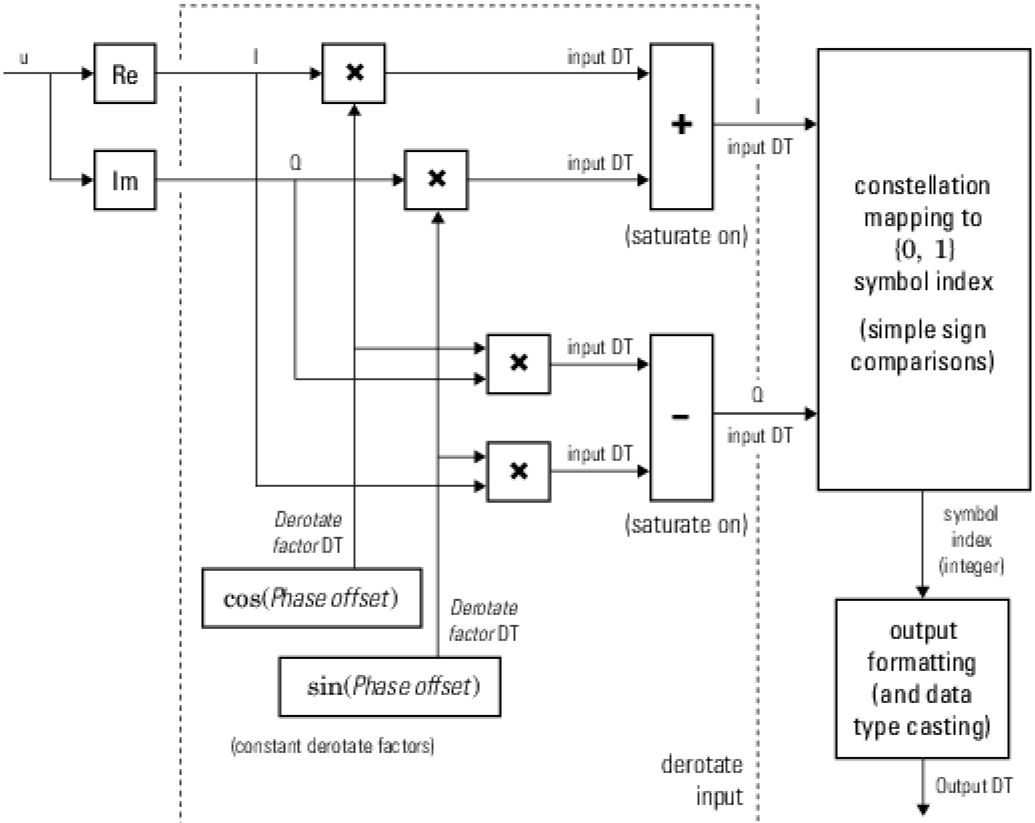
Algorithms


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)



Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

For more details about the exact LLR and approximate LLR cases (soft-decision), see “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the Communications Toolbox User's Guide.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder™ provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see “OutputPipeline” (HDL Coder).

See Also

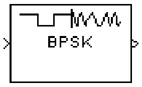
Blocks
BPSK Modulator Baseband | DBPSK Demodulator Baseband | M-PSK Demodulator Baseband | QPSK Demodulator Baseband

Introduced before R2006a

BPSK Modulator Baseband

Modulate signal by using BPSK method

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM
Communications Toolbox HDL Support / Modulation / PM



Description

The BPSK Modulator Baseband block modulates a signal by using the binary phase shift keying (BPSK) method. The output is a baseband representation of the modulated signal.

The input signal must be a discrete-time binary-valued signal. If the input bit is 0 or 1, then the modulated symbol is *exp(jθ)* or *-exp(jθ)*, respectively, where *θ* represents the **Phase offset (rad)** parameter.

Ports

Input

`Port_1` — Input data
scalar | vector

Input signal, specified as a scalar or vector.

- If you specify a scalar, it must be an integer from 0 to M – 1, where M is the modulation order.
- If you specify a vector, the elements must be either integers from 0 to M-1 or binary values. If you specify a binary vector, the number of elements must be an integer multiple of the number of bits per symbol. The number of bits per symbol is equal to $\log_2(M)$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — BPSK-modulated baseband signal
scalar | vector

BPSK-modulated baseband signal, returned as a complex scalar or vector of complex values.

Data Types: `single` | `double` | `fixed point`

Parameters

`Phase offset (rad)` — Phase of zeroth point
`0` (default) | finite real-valued scalar

Specify a finite real-valued scalar for the phase of the zeroth point of the constellation in radians.

Example: `pi/4`

`Output data type` — Output data type

`double` (default) | `single` | `Inherit via back propagation` | `fixdt(1,16)` | `fixdt(1,16,0)` | `<data type expression>`

Data type of output,. Set this parameter to one of the fixed point options or `<data type expression>` to enable parameters in which you specify additional details. Set this parameter to `Inherit via back propagation`, to match the output data type and scaling to the following block in the model.

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[ab] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

a. ufix(ceil(log2(M))) only at the input for M-ary modulation.

b. Fixed-point outputs must be signed.

Tips

The BPSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This constellation visualization feature allows you to visualize a signal constellation for the specified block parameters. For more information, see the “Constellation Visualization” section of the *Communications Toolbox User's Guide*.

Algorithms

Phase modulation is a linear baseband modulation technique in which the message modulates the phase of a constant amplitude signal. Binary Phase Shift Keying (BPSK) is a two phase modulation scheme, where the 0’s and 1’s in a binary message are represented by two different phase states in the carrier signal

$$s_n(t) = \sqrt{\frac{2E_b}{T_b}}\cos(2\pi f_c t + \pi(1 - n));\ n \in \left\{0, 1\right\},$$

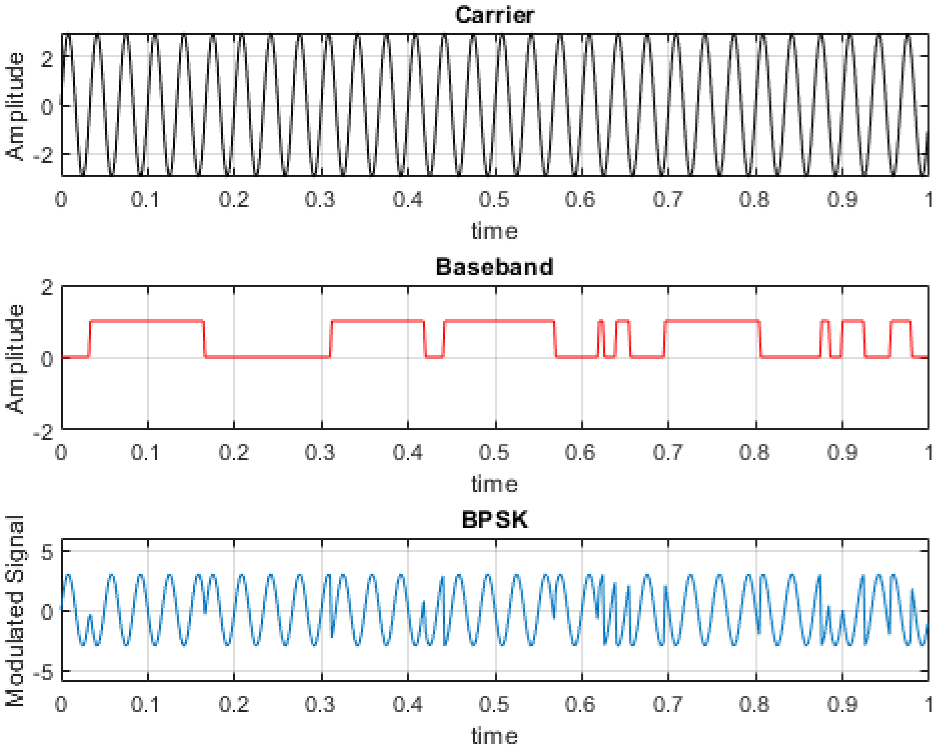
where:

- E_b is the energy per bit.
- T_b is the bit duration.
- f_c is the carrier frequency.

In MATLAB, the baseband representation of a BPSK signal is

$$s_n(t) = \cos(\pi n);\ n \in \{0, 1\}\ .$$

The BPSK signal has two phases: 0 and π.



The probability of a bit error in an AWGN channel is

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right),$$

where N_0 is the noise power spectral density.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Objects

`comm.BPSKDemodulator` | `comm.BPSKModulator` | `comm.DBPSKModulator` | `comm.PSKModulator`

Blocks

BPSK Demodulator Baseband | DBPSK Modulator Baseband | M-PSK Modulator Baseband | QPSK Modulator Baseband

Introduced before R2006a

Carrier Synchronizer

Compensate for carrier frequency offset

Out
Carrier
Synchronizer
PhEst
Carrier Synchronizer

Library

Synchronization

Description

The Carrier Synchronizer block compensates for carrier frequency and phase offsets using a closed-loop approach for BPSK, QPSK, OQPSK, 8-PSK, QAM, and PAM modulation schemes. The block accepts a single input port. To obtain an estimate of the phase error in radians, select the **Estimated phase error output port** check box. The block accepts a sample- or frame-based complex input signal and returns a complex output signal and a an real phase estimate. The block outputs have the same dimensions as the input.

Note

- This block does not resolve phase ambiguities created by the synchronization algorithm. As indicated in this table, the potential phase ambiguity introduced by the synchronizer depends on the modulation type:

Modulation	Phase Ambiguity (degrees)
`'BPSK'` or `'PAM'`	0, 180
`'OQPSK'`, `'QPSK'`, or `'QAM'`	0, 90, 180, 270
`'8PSK'`	0, 45, 90, 135, 180, 225, 270, 315

- For best results, apply carrier synchronization to non-oversampled signals.

Parameters

Modulation

Specify the modulation type as `BPSK`, `QPSK`, `OQPSK`, `8PSK`, `QAM`, or `PAM`.

Modulation phase offset

Specify the method used to calculate the modulation phase offset as either `Auto` or `Custom`.

- `Auto` applies the traditional offset for the specified modulation type.

Modulation	Phase Offset (radians)
BPSK, QAM, or PAM	0
QPSK or OQPSK	$\pi/4$
8PSK	$\pi/8$

- Custom enables the **Custom phase offset (radians)** parameter.

Custom phase offset (radians)

Specify the phase offset in radians as a real scalar. This parameter is available only when **Modulation phase offset** is set to Custom.

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar.

Damping factor

Specify the damping factor of the loop as a positive real finite scalar.

Normalized loop bandwidth

Specify the normalized loop bandwidth as a real scalar between 0 and 1. The bandwidth is normalized by the sample rate of the carrier synchronizer block.

Estimated phase error output port

Select this check box to provide the estimated phase error to an output port.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

`Interpreted execution`

Simulate model using all supported MATLAB functions. Choosing this option can slow simulation performance.

Algorithms

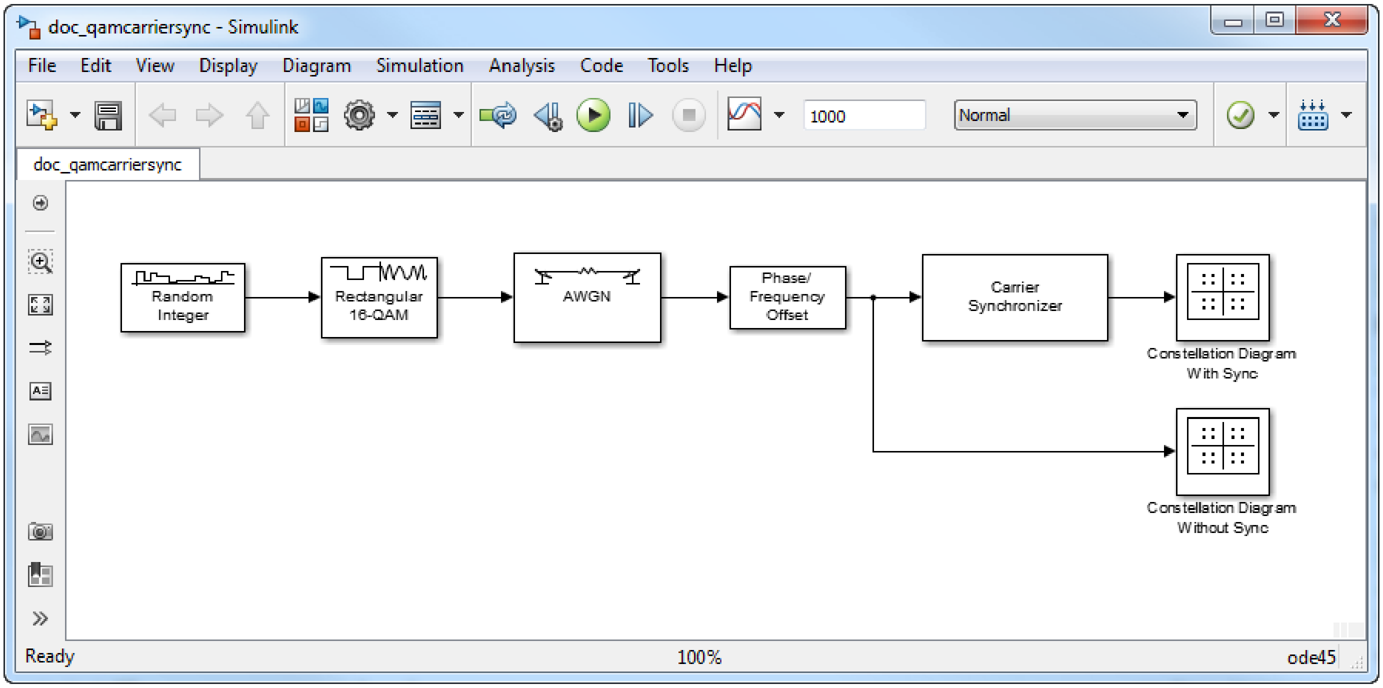
This block implements the algorithm, inputs, and outputs described on the `comm.CarrierSynchronizer` reference page. The object properties correspond to the block parameters.

Examples

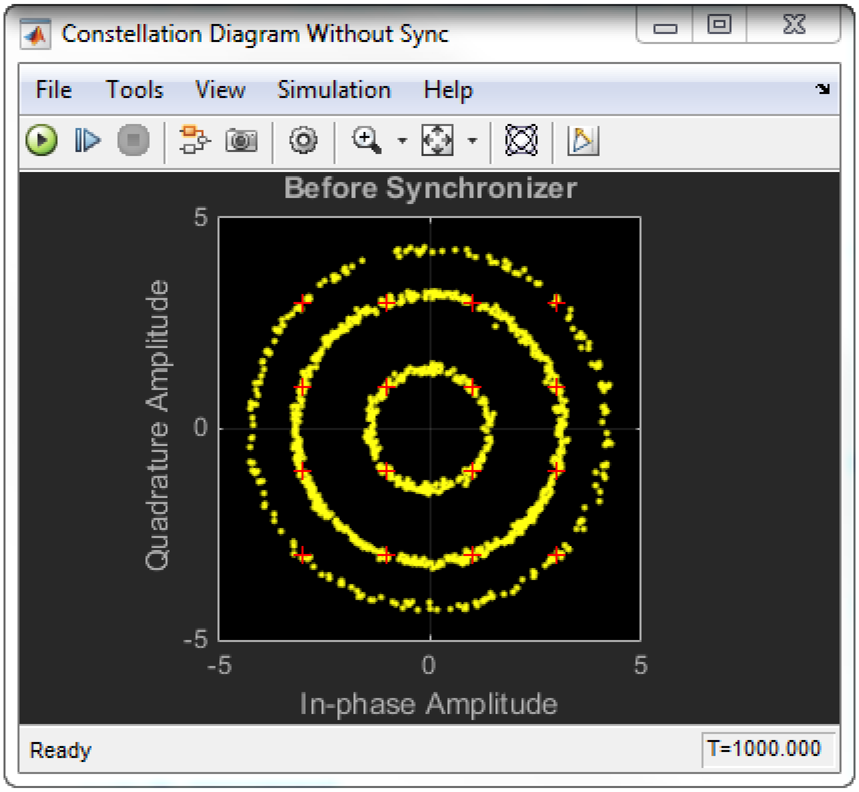
Correct for Frequency and Phase Offset

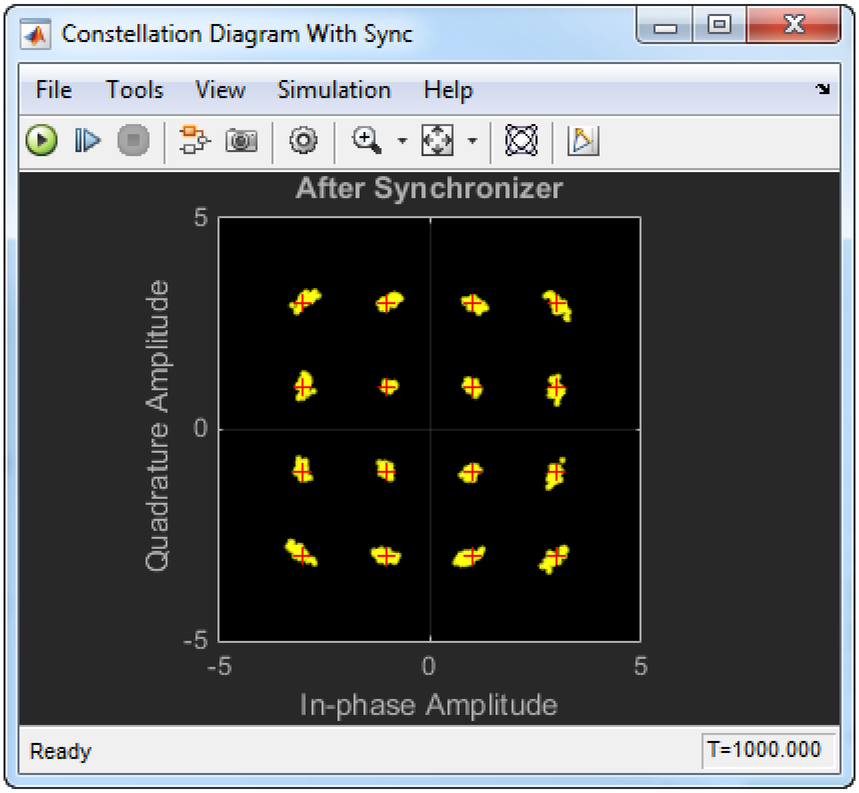
Correct for a phase and frequency offset imposed on a noisy 16-QAM channel using the Carrier Synchronizer block.

Open the doc_qamcarriersync model.


doc_qamcarriersync - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
1000
Normal
doc_qamcarriersync
Random Integer
Rectangular 16-QAM
AWGN
Phase/ Frequency Offset
Carrier Synchronizer
Constellation Diagram With Sync
Constellation Diagram Without Sync
Ready
100%
ode45

Run the model. The Constellation Diagram Without Sync block shows a spiral pattern that indicates a phase and frequency offset. After the carrier synchronizer converges to a solution, the data displayed on the Constellation Diagram With Sync block are grouped around the reference constellation.





Experiment with the parameters in the Phase/Frequency Offset and Carrier Synchronizer blocks. By varying these parameters, you can change how quickly the output conforms to an ideal 16-QAM constellation.

If the signal does not converge to the expected constellation, additional measures can be taken to achieve successful recovery. For more information, see the “Correct Phase and Frequency Offset for 16-QAM Using Coarse and Fine Synchronization” on page 3-275 example.

Supported Data Types

Port	Supported Data Types
Signal Input	• Double-precision floating point • Single-precision floating point
Signal Output	• Double-precision floating point • Single-precision floating point
Phase Error Estimate	• Double-precision floating point • Single-precision floating point

References

[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2009, pp. 359–393.

[2] Huang, Zhijie, Zhiqiang Yi, Ming Zhang, and Kuang Wang. “8PSK Demodulation for New Generation DVB-S2.” *International Conference on Communications, Circuits and Systems, 2004. ICCCAS 2004.* Vol. 2, 2004, pp. 1447–1450.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Biquad Filter

Objects

`comm.CarrierSynchronizer`

Functions

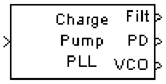
Topics

“MSK Signal Recovery”

Introduced in R2015a

Charge Pump PLL

Implement charge pump phase-locked loop using digital phase detector



Library

Components sublibrary of Synchronization

Description

The Charge Pump PLL (phase-locked loop) block automatically adjusts the phase of a locally generated signal to match the phase of an input signal. It is suitable for use with digital signals.

This PLL has these three components:

- A sequential logic phase detector, also called a digital phase detector or a phase/frequency detector.
- A filter. You specify the filter transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```
- A voltage-controlled oscillator (VCO). You specify characteristics of the VCO using the **VCO input sensitivity**, **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

A sequential logic phase detector operates on the zero crossings of the signal waveform. The equilibrium point of the phase difference between the input signal and the VCO signal equals π. The sequential logic detector can compensate for any frequency difference that might exist between a VCO and an incoming signal frequency. Hence, the sequential logic phase detector acts as a frequency detector.

Parameters

Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

Lowpass filter denominator

The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

VCO input sensitivity (Hz/V)

This value scales the input to the VCO and, consequently, the shift from the **VCO quiescent frequency** value. The units of **VCO input sensitivity** are Hertz per volt.

VCO quiescent frequency (Hz)

The frequency of the VCO signal when the voltage applied to it is zero. This should match the frequency of the input signal.

VCO initial phase (rad)

The initial phase of the VCO signal.

VCO output amplitude

The amplitude of the VCO signal.

See Also

Phase-Locked Loop

References

For more information about digital phase-locked loops, see the works listed in “Selected Bibliography for Synchronization”.

Extended Capabilities

C/C++ Code Generation

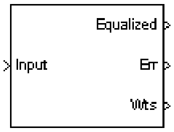
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

CMA Equalizer

(To be removed) Equalize using constant modulus algorithm

Note will be removed in a future release. Use Linear Equalizer instead.



Library

Equalizers

Description

The CMA Equalizer block uses a linear equalizer and the constant modulus algorithm (CMA) to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the CMA to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

When using this block, you should initialize the equalizer weights with a nonzero vector. Typically, CMA is used with differential modulation; otherwise, the initial weights are very important. A typical vector of initial weights has a 1 corresponding to the center tap and zeros elsewhere.

Input and Output Signals

The `Input` port accepts a scalar-valued or column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal.

You can configure the block to have one or more of the extra ports listed in the table below.

Port	Meaning	How to Enable
`Err` output	y(R -|y|^2), where y is the equalized signal and R is a constant related to the signal constellation	Select **Output error**.
`Wts` output	A vector listing the weights after the block has processed either the current input frame or sample.	Select **Output weights**.

Algorithms

Referring to the schematics in “Adaptive Equalizers”, define *w* as the vector of all weights w_i and define *u* as the vector of all inputs u_i. Based on the current set of weights, *w*, this adaptive algorithm creates the new set of weights given by

(`LeakageFactor`) w + (`StepSize`) u^*e

where the * operator denotes the complex conjugate.

Equalizer Delay

The delay between the transmitter's modulator output and the CMA equalizer output is typically unknown (unlike the delay for other adaptive equalizers in this product). If you need to determine the delay, you can use the Find Delay block.

Parameters

Number of taps

The number of taps in the filter of the equalizer.

Number of samples per symbol

The number of input samples for each symbol.

When you set this parameter to 1, the filter weights are updated once for each symbol, for a symbol spaced (i.e. T-spaced) equalizer. When you set this parameter to a value greater than one, the weights are updated once every N^{th} sample, for a fractionally spaced (i.e. T/N-spaced) equalizer.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Step size

The step size of the CMA.

Leakage factor

The leakage factor of the CMA, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that lists the initial weights for the taps.

Output error

If you check this box, the block outputs the error signal described in the table above.

Output weights

If you check this box, the block outputs the current weights.

References

[1] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[2] Johnson, Richard C. Jr., Philip Schniter, Thomas. J. Endres, et al., "Blind Equalization Using the Constant Modulus Criterion: A Review," *Proceedings of the IEEE*, vol. 86, pp. 1927-1950, October 1998.

Compatibility Considerations

CMA Equalizer will be removed

Warns starting in R2020a

- CMA Equalizer will be removed in a future release. Use Linear Equalizer instead with the adaptive algorithm set to CMA.
- The Linear Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the CMA Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

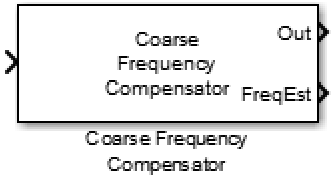
Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Introduced before R2006a

Coarse Frequency Compensator

Compensate for carrier frequency offset for PAM, PSK, or QAM



Library

Synchronization

Description

The Coarse Frequency Compensator block compensates for a carrier frequency offset for BPSK, QPSK, OQPSK, 8-PSK, PAM, and QAM modulation schemes. The block accepts a single input signal. To obtain an estimate of the frequency offset in Hz, select the **Estimated frequency offset output port** check box. The block accepts a sample- or frame-based complex input signal and returns a complex output signal and a real frequency offset estimate. The output signal has the same dimensions as the input signal. The frequency offset estimate is a scalar.

Parameters

Modulation type of input signal

Specify the modulation type as `BPSK`, `QPSK`, `OQPSK`, `8PSK`, `PAM`, or `QAM`.

The default setting is `QAM`.

Estimation algorithm

Specify the frequency offset estimation algorithm as `FFT-based` or `Correlation-based`. This parameter appears when **Modulation type of input signal** is `BPSK`, `QPSK`, `8PSK`, or `PAM`.

The table shows the allowable combinations of the modulation type and the estimation algorithm.

Modulation	FFT-Based Algorithm	Correlation-Based Algorithm
BPSK, QPSK, 8PSK, PAM	✓	✓
OQPSK, QAM	✓	

Frequency resolution (Hz)

Specify the frequency resolution in Hz as a positive real scalar. This option is available when the `FFT-based` algorithm is used. The default setting is `0.001` Hz.

Samples per symbol

Specify the number of samples per symbol as a positive integer scalar greater than or equal to 4. The default setting is 4.

Maximum frequency offset (Hz)

Specify the maximum frequency offset in Hz as a positive real scalar. This option is appears when you set **Estimation algorithm** to `Correlation-based`. The default setting is `0.05` Hz.

Estimated frequency offset output port

Select this check box to provide the estimated frequency offset to an output port. The default for this parameter is selected.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code for the block using only MATLAB functions supported for code generation. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects accept a maximum of nine inputs.

`Interpreted execution`

Simulate your model using all supported MATLAB functions. Choosing this option can slow simulation performance.

The default setting is `Code generation`.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.CoarseFrequencyCompensator` reference page. The object properties correspond to the block parameters.

Examples

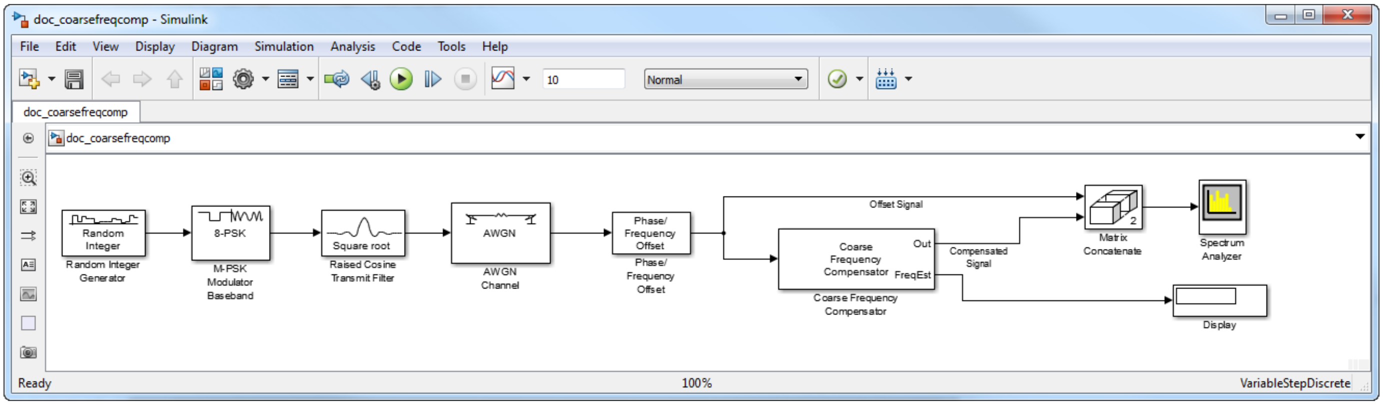
Correct for Frequency and Phase Offset

Correct for a frequency offset imposed on a noisy 8-PSK channel by using the Coarse Frequency Compensator block.

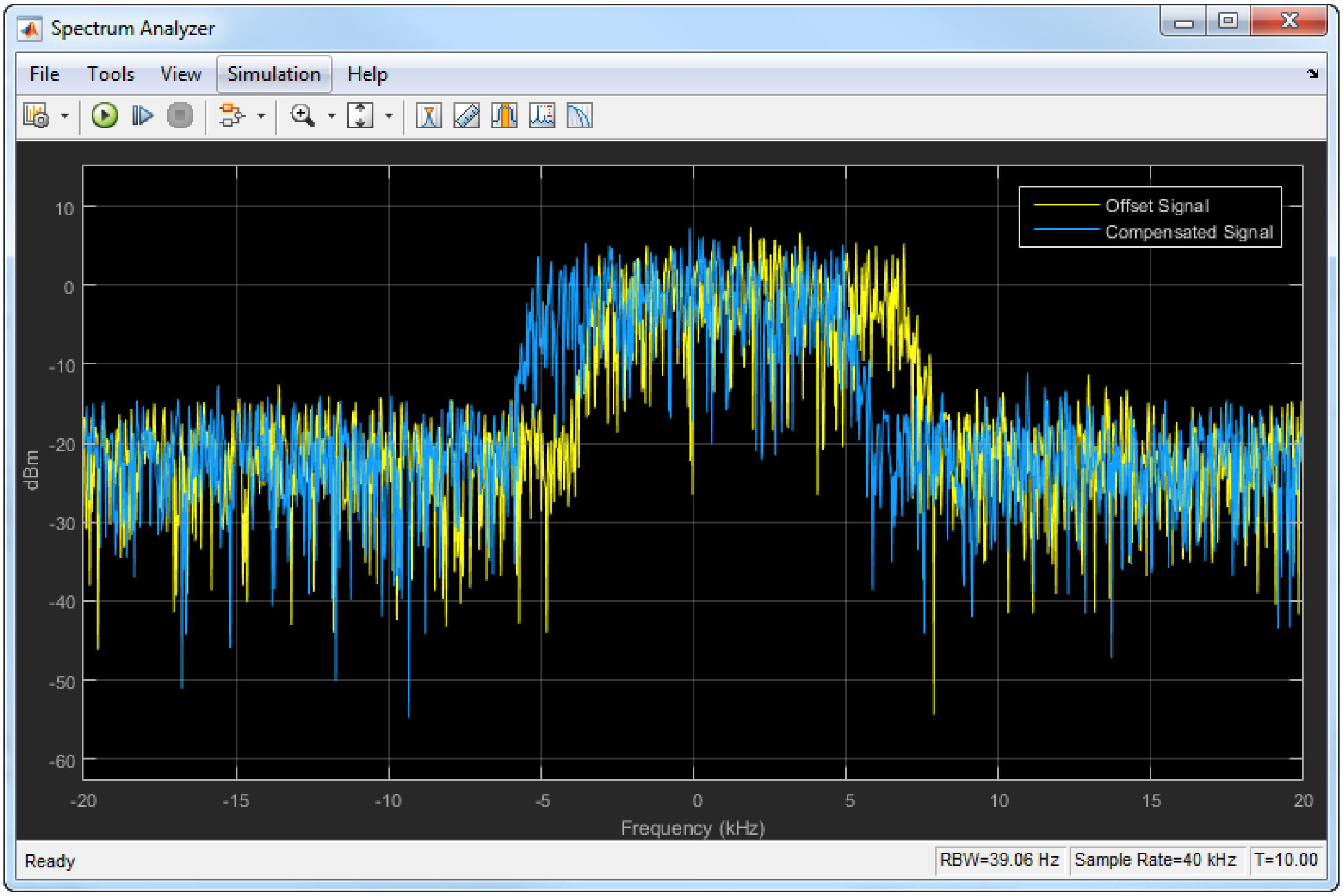
Open the doc_coarsefreqcomp model.

Open the dialog boxes to verify these parameter values:

- Random Integer Generator — **Sample time** is `1e-4`, which is equivalent to a 10 ksym/sec symbol rate.
- Raised Cosine Transmit Filter — **Output samples per symbol** is `4`.
- AWGN Channel — **Mode** is `Signal to noise ratio (SNR)` and **SNR (dB)** is `20`.
- Phase/Frequency Offset — **Frequency offset (Hz)** is `2000`.
- Coarse Frequency Compensator — **Estimation algorithm** is `FFT-based` and **Frequency resolution (Hz)** is `1`.



Run the model. The Spectrum Analyzer block shows both the frequency offset signal and the compensated signal. In addition, the Display block shows the estimate of the frequency offset. Observe that the spectrum plot shows that the Coarse Frequency Compensator correctly centers the signal around 0 Hz. Additionally, the display shows that the estimated frequency offset is 2000 Hz.



Adjust the parameters in the Phase/Frequency Offset and Coarse Frequency Compensator blocks and see their effect on frequency compensation performance.

Supported Data Types

Port	Supported Data Types
Signal Input	• Double-precision floating point • Single-precision floating point
Signal Output	• Double-precision floating point • Single-precision floating point
Frequency Estimate	• Double-precision floating point • Single-precision floating point

References

[1] Luise, M. and R. Regiannini. “Carrier recovery in all-digital modems for burst-mode transmissions.” *IEEE Transactions on Communications*.Vol. 43, No. 2, 3, 4, Feb/Mar/April, 1995, pp. 1169–1178.

[2] Wang, Y., K. Shi, and E. Serpedi. “Non-Data-Aided Feedforward Carrier Frequency Offset Estimators for QAM Constellations: A Nonlinear Least-Squares Approach.” *EURASIP Journal on Applied Signal Processing*. 2004:13, pp. 1993–2001.

[3] Nakagawa, T., M. Matsui, T. Kobayashi, K. Ishihara, R. Kudo, M. Mizoguchi, and Y. Miyamoto. “Non-Data-Aided Wide-Range Frequency Offset Estimator for QAM Optical Coherent Receivers.” *Optical Fiber Communication Conference and Exposition (OFC/NFOEC), 2011 and the National Fiber Optic Engineers Conference*. March 2011, pp. 1–3.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Carrier Synchronizer | Symbol Synchronizer

Objects

`comm.CoarseFrequencyCompensator`

Introduced in R2015b

Complex Phase Difference

Phase difference between two complex signals
Library: Communications Toolbox / Utility Blocks

Description

The Complex Phase Difference block computes the phase difference in radians between the second input signal and the first input signal. The elements of the output are between -π and π. This block independently processes each pair of corresponding elements.

Ports

Input

`In1` — First input signal
scalar | column vector | matrix

First input signal, specified as a scalar, column vector, or matrix. If both input signals are nonscalar, they must be the same dimension.

Note This block processes complex signals. Real values at the input ports are type cast to complex values with `+j0` complex components.

Data Types: `double` | `single`
Complex Number Support: Yes

`In2` — Second input signal
scalar | column vector | matrix

Second input signal, specified as a scalar, column vector, or matrix. If both input signals are nonscalar, they must be the same dimension. **In2** must be the same data type as **In1**.

Note This block processes complex signals. Real values at the input ports are type cast to complex values with `+j0` complex components.

Data Types: `double` | `single`

Output

`Out1` — Phase shift difference
scalar | column vector | matrix

Phase shift difference in radians, returned as a scalar, column vector, or matrix. If either input signal is nonscalar, the output signal dimension matches the dimension of the nonscalar input signal. The output is the phase difference between the second input signal and the first input signal. The elements of the output signal are between -π and π and are the same data type as the input signals.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Complex Phase Shift

Introduced before R2006a

Complex Phase Shift

Apply complex phase shift to complex signal
Library: Communications Toolbox / Utility Blocks

In
Ph
Complex
Phase Shift

Description

The Complex Phase Shift block applies a complex phase shift to a complex signal. This block independently processes each pair of corresponding elements.

Ports

Input

`In` — Input signal
scalar | column vector | matrix

Input signal, specified as a scalar, column vector, or matrix.

Note This block processes and outputs complex signals. Real values at the input port are type cast to complex values with `+j0` complex components.

Data Types: `double` | `single`
Complex Number Support: Yes

Ph — Phase shift
scalar | column vector | matrix

Phase shift in radians, specified as a scalar, column vector, or matrix. If the phase shift is nonscalar, it must have the same dimension as the signal at port **In**.

Data Types: `double`

Output

`Out1` — Phase-shifted signal
scalar | column vector | matrix

Phase-shifted signal, returned as a complex-valued scalar, column vector, or matrix. This output is the same dimension and data type as the input signal.

Block Characteristics

Data Types	`double` | `single`

Multidimensional Signals	no
Variable-Size Signals	no

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

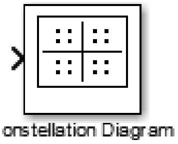
Complex Phase Difference

Introduced before R2006a

Constellation Diagram

Display constellation diagram for input signals

Library: Communications Toolbox / Comm Sinks
Communications Toolbox HDL Support / Comm Sinks

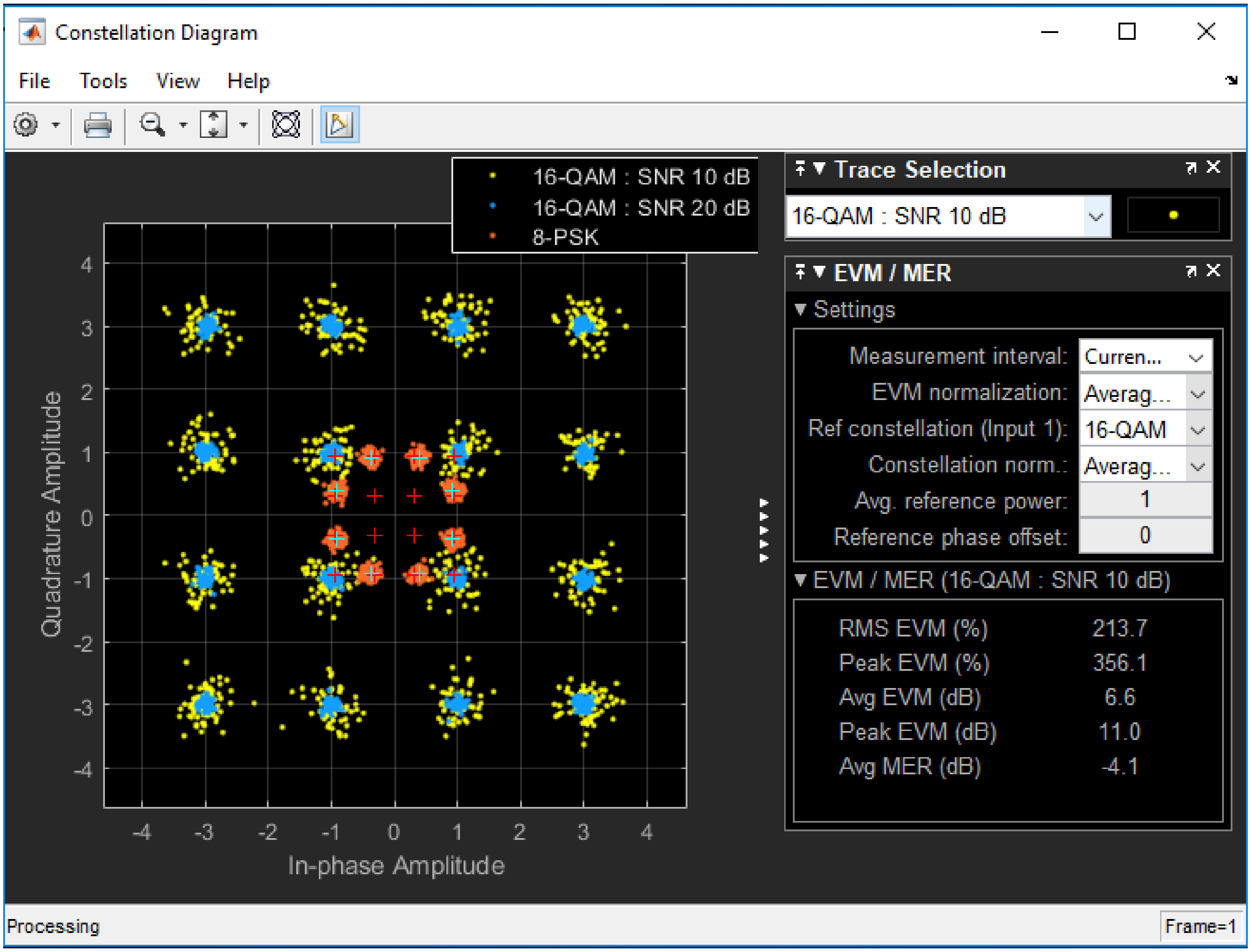


Description

The Constellation Diagram block displays real and complex-valued floating and fixed-point signals in the I/Q plane. Use this block to perform qualitative and quantitative analysis on modulated single-carrier signals.

In the constellation diagram window you can:

- Input and plot multiple signals on a single constellation diagram. You can define one reference constellation for each input signal. For more information, see **Reference constellation**.
- Choose which channels are displayed by selecting signals in the legend. Use the Show legend parameter to display the legend.
- Display the “EVM / MER Measurements” on page 5-120 panel, which displays calculated error vector magnitude (EVM) and modulation error ratio (MER) measurements. When a multichannel signal is input, use Trace Selection to choose the signal being measured.



Trace Selection: 16-QAM : SNR 10 dB

Measurement interval: Curren...
EVM normalization: Averag...
Ref constellation (Input 1): 16-QAM
Constellation norm.: Averag...
Avg. reference power: 1
Reference phase offset: 0

EVM / MER (16-QAM : SNR 10 dB)

```
RMS EVM (%)     213.7
Peak EVM (%)    356.1
Avg EVM (dB)      6.6
Peak EVM (dB)    11.0
Avg MER (dB)     -4.1
```

Ports

Input

`Port_1` — Signal or signals to visualize

column vector | matrix

Connect to the signal or signals you want to visualize as an N_{sym}-by-1 column vector or N_{sym}-by-N_{sig} matrix. N_{sym} is the number of symbols and N_{sig} is the number of input signals.

You can specify up to 20 input signals. Specifically, the maximum number of channels through all the ports is 20. For example, if you create a two-channel signal for every input port, then you can define up 10 number of input ports.

Example: `[-1 + 1i; -1 - 1i; 1 + 1i; 1 - 1i]` specifies a 4-symbol input signal.

Parameters

File

From the Constellation Diagram window, select **File** to view the options available.

`Open at Start of Simulation` — Open constellation diagram at start of simulation
on (default) | off

Select to open constellation diagram window at start of simulation. Deselect to prevent constellation diagram window from opening at start of simulation.

`Number of input ports` — Number of input ports on scope block
`1` (default) | positive integer in the range [1, 20]

Specify the number of input ports on the Scope block, specified as an integer in the range [1, 20].

When multichannel input signals are specified, the maximum number of input ports is limited by the total number of input signals defined. The total number of input signal cannot exceed 20.

Tools > Axes Scaling Properties

From the Constellation Diagram window, under **Tools**, select **Axes Scaling Properties** to open the Axes Scaling Properties: Constellation Diagram dialog box. In this dialog box, you can customize the graphical properties of the axes.

Properties

`Axes scaling` — Axes scaling options
`Manual` (default) | `Auto` | `After N Updates`

Axes scaling options, specified as:

- `Manual` — Applies the *x* and *y* axes limits specified in the Visuals – Constellation Properties: Constellation Diagram dialog box.
- `Auto` — Scales the axes limits as needed during and after simulation.
- `After N Updates` — Scales the axes limits after the specified **Number of updates**.

Tunable: Yes

`Number of updates` — Number of updates after which to scale the axes
`10` (default) | positive integer

Number of updates after which to scale the axes, specified as a positive integer.

Tunable: Yes

Dependencies

This parameter appears when **Axes scaling** is set to `After N Updates`.

`Scale axes limits at stop` — Option to scale axes at end of simulation
`off` (default) | `on`

Select to scale axes at end of the simulation to the data range percentage limits specified by **X-axis Data range (%)** and **Y-axis Data range (%)**.

`X-axis Data range (%)` — Percentage of *x*-axis used to display data

`80` (default) | scalar from 1 to 100

Percentage of *x*-axis used to display data.

Example: `100` scales the x-axis range to the maximum value of the in-phase amplitude component of the input signal.

Tunable: Yes

`X-axis Align` — Align data along *x*-axis

`Center` (default) | `Left` | `Right`

Align data along *x*-axis, specified as `Center`, `Left`, or `Right`.

Example: `Right` aligns the maximum value of the in-phase amplitude component of the input signal toward the upper x-axis limit.

Tunable: Yes

`Y-axis Data range (%)` — Percentage of *y*-axis used to display data

`80` (default) | scalar from 1 to 100

Percentage of *y*-axis used to display data.

Example: `30` scales the y-axis range so that the maximum value of the quadrature amplitude component of the signal occupies 30% of the y-axis range.

Tunable: Yes

`Y-axis Align` — Align data along *y*-axis

`Center` (default) | `Top` | `Bottom`

Align data along *y*-axis, specified as `Center`, `Top`, or `Bottom`.

Example: `Bottom` aligns the maximum value of the quadrature amplitude component of the signal toward the lower y-axis limit.

Tunable: Yes

Tools > Measurements

From the Constellation Diagram window, under **Tools**, select **Measurements** for options to display the **Trace Selection** and **Signal Quality** panes. By default these panes are docked in the Constellation Diagram window when displayed.

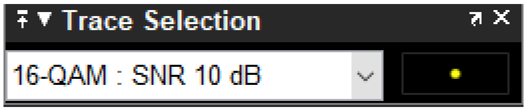
The **Signal Quality** pane contains the **Settings** and **EVM / MER** subpanes. Both subpanes can be independently expanded or collapsed.

For more information about the signal quality measurements, see “EVM / MER Measurements” on page 5-120.

`Trace Selection` — Signal trace used to compute measurements

list of signals

Select the signal trace used to compute measurements from the list of available signals. This parameter lists the signals input to the block.



Tunable: Yes

Dependencies

To set this parameter, the **Trace Selection** pane must be visible. The **Trace Selection** pane automatically appears when you plot multiple signals on the Constellation Diagram window. To hide or display the **Trace Selection** pane from the Constellation Diagram window, under **Tools**, select **Measurements** > **Trace Selection** to toggle the visibility of the traceback selection measurement.

EVM / MER Settings Pane

`Measurement interval` — Duration of EVM or MER measurement

`Current Display` (default) | `All displays` | positive integer

Duration of the EVM or MER measurement in symbols, specified as `Current Display`, `All displays`, or a positive integer. To specify a positive integer, select and then replace `<user-defined>` with your desired value. The value must be positive and less than or equal to **Symbols to display**. The measurement is computed after the number of input data samples exceeds the measurement interval.

Tunable: Yes

`EVM normalization` — Normalization method used for EVM calculation

`Average constellation power` (default) | `Peak constellation power`

Normalization method used for EVM calculation, specified as `Average constellation power` or `Peak constellation power`. The `comm.EVM` System object computes the EVM.

Tunable: Yes

`Reference constellation` — Reference constellation

`QPSK` (default) | `BPSK` | `8-PSK` | `16-QAM` | `64-QAM` | `256-QAM` | vector

Reference constellation, specified as `BPSK`, `QPSK`, `8-PSK`, `16-QAM`, `64-QAM`, `256-QAM`, or a <user-defined> vector. To specify a custom value, select `<user-defined>`, then in the **Custom value** property replace the entry with your desired value.

Each input port can have its own reference constellation. For a multichannel input signal, a single reference constellation is applied for all signals in that input port.

To obtain the EVM/MER measurements, you must set **Reference constellation** to a valid value corresponding to the modulation of the input signal.

Tunable: Yes

`Custom value` — Input reference constellation

vector

Input the reference constellation, specified as a vector.

Dependencies

To enable this parameter, set **Reference constellation** to <user-defined>.

Data Types: `double`
Complex Number Support: Yes

`Average reference power` — Average power of reference constellation

`1` (default) | positive scalar

Average power of the reference constellation in watts, specified as a positive scalar and referenced to a one-ohm load.

Tunable: Yes

`Reference phase offset (rad)` — Phase offset of reference constellation

`pi/4` (default) | scalar

Phase offset of the reference constellation in radians, specified as a scalar.

Tunable: Yes

View > Configuration Properties

From the Constellation Diagram window, select **View** > **Configuration Properties** to open the Visuals – Constellation Properties: Constellation Diagram dialog box. In this dialog box, you can customize the graphical properties of the plotted signals.

Main

`Number of input ports` — Number of input ports on scope block

1 (default) | positive integer in the range [1, 20]

Specify the number of input ports on the Scope block, specified as an integer in the range [1, 20].

`Samples per symbol` — Number of samples used to represent each symbol

`1` (default) | positive integer

Number of samples used to represent each symbol, specified as a positive integer. When **Samples per symbol** is greater than 1, the signal is downsampled before it is plotted.

Tunable: Yes

`Offset (samples)` — Number of samples to skip before plotting points

`0` (default) | nonnegative integer

Number of samples to skip before plotting points, specified as a nonnegative integer less than Samples per symbol. This parameter specifies the number of samples to skip when downsampling the input signal.

Tunable: Yes

`Symbols to display` — Maximum number of symbols to display

`Input frame length` (default) | positive integer

Maximum number of symbols to display, specified as `Input frame length` or a positive integer. To specify a positive integer, select and then replace `<user-defined>` with your desired value.

Use **Symbols to display** to limit the maximum number of symbols displayed when long signals are input. Symbols plotted are the most recent symbols received.

Tunable: Yes

Display

`Show grid` — Display plot grid lines

`on` (default) | `off`

Select to display plot grid lines.

Tunable: Yes

`Show legend` — Display plot legend

`off` (default) | `on`

Select to display plot legend. The names listed in the legend are the signal names from the model.

From the legend, you can control which signals are plotted. This control is equivalent to changing the visibility in the **View** > **Style** dialog box. In the scope legend, click a signal name to hide the signal in the scope. To show the signal, click the signal name again. To show only one signal and hide all other signals, right-click the signal name. To show all signals, press **Esc**.

Tunable: Yes

`Show signal trajectory` — Display signal trajectory

`off` (default) | `on`

Select to display the trajectory between constellation points for the plotted signals.

Tunable: Yes

`Color fading` — Option to add color fading effect

`off` (default) | `on`

When you select **Color fading**, the points in the display fade as the interval of time after they are first plotted increases. **Color fading** is for animation that resembles an oscilloscope.

Tunable: Yes

`X-limits (Minimum)` — Minimum *x*-axis value

`-1.375` (default) | scalar

Minimum *x*-axis value, specified as a scalar.

Tunable: Yes

`X-limits (Maximum)` — Maximum *x*-axis value

`1.375` (default) | scalar

Maximum *x*-axis value, specified as a scalar.

Tunable: Yes

`Y-limits (Minimum)` — Minimum *y*-axis value

`-1.375` (default) | scalar

Minimum *y*-axis value, specified as a scalar.

Tunable: Yes

`Y-limits (Maximum)` — Maximum *y*-axis value

`1.375` (default) | scalar

Maximum *y*-axis value, specified as a scalar.

Tunable: Yes

`Title` — Title on plot

blank (default) | character vector | string

Title on plot, specified as a character vector or string.

Tunable: Yes

`X-axis label` — *x*-axis label

`'In-phase Amplitude'` (default) | character vector | string

x-axis label, specified as a character vector or string.

Tunable: Yes

`Y-axis label` — *y*-axis label

`'Quadrature Amplitude'` (default) | character vector | string

y-axis label, specified as a character vector or string.

Tunable: Yes

Reference constellation

`Show reference constellation` — Select to display reference constellation

on (default) | off

Select to display the reference constellation.

Tunable: Yes

`Input` — Input port number

`1` (default) | integer

Select the input port number for which you want to view/change the reference constellation and the associated attributes.

Dependencies

To enable this parameter, set the **Number of input ports** parameter to a value greater than 1.

`Reference constellation` — Reference constellation

`QPSK` (default) | `BPSK` | `8-PSK` | `16-QAM` | `64-QAM` | `256-QAM` | vector

Reference constellation, specified as `BPSK`, `QPSK`, `8-PSK`, `16-QAM`, `64-QAM`, `256-QAM`, or a <user-defined> vector. To specify the value, select and then replace `<user-defined>` with your desired value. When defined by the user, the reference constellation values can be specified as a vector.

Each input port can have its own reference constellation. For a multichannel input signal, a single reference constellation is applied for all signals in that input port.

To obtain the EVM/MER measurements, you must set **Reference constellation** to a valid value corresponding to the modulation of the input signal.

Tunable: Yes

Data Types: `double`
Complex Number Support: Yes

`Average reference power` — Average power of reference constellation

`1` (default) | positive scalar

Average power of the reference constellation in watts, specified as a positive scalar and referenced to a one-ohm load.

Tunable: Yes

`Reference phase offset (rad)` — Phase offset of reference constellation

`pi/4` (default) | scalar

Phase offset of the reference constellation in radians, specified as a scalar.

Tunable: Yes

View > Style

From the Constellation Diagram window, select **View** > **Style** to open the Constellation Diagram - Style dialog box. In this dialog box, you can customize the graphical properties of the components in the Constellation Diagram window.

`Figure color` — Select background color

gray (default)

Select the background color within the Constellation Diagram window and outside the scope axes.

`Axes colors` — Select colors of plot and measurement panes

black (default for background) | gray (default for axes)

Select colors of plot and measurement panes. The first color option specifies the background color of the plot figure and the measurement panes. The second option specifies the color of the plot figure axes (ticks, labels, and grid lines) and the text. For more a description of the measurement panes, see in the “Tools > Measurements” on page 5-113.

Tunable: Yes

`Properties for channel` — View or change graphical properties of each channel

`Channel 1` (default)

Select a channel to view or change its graphical properties.

`Bring To Front` — Bring channel to front

button

Bring the active channel, as indicated by **Properties for channel**, to the front.

`Show signal and reference constellation` — Option to hide channel

`on` (default) | `off`

Clear to hide the active channel and its associated reference constellation. Use **Properties for channel** to select the active channel. The setting is synchronized with actions in the interactive legend, see Show legend.

`Symbols` — Set properties of symbols

symbol properties

Set graphical properties of the symbols for the active channel. Adjust style, size, line width, and color of the marker. Use **Properties for channel** to select the active channel.

Dependencies

To set marker shape to `none`, **Show signal trajectory** must be selected.

`Signal trajectory` — Select properties of signal trajectory

no line (default) | trajectory properties

Set graphical properties for the signal trajectory of the active channel. Adjust style, width, and color of the line. Use **Properties for channel** to select the active channel.

Dependencies

To adjust signal trajectory properties, **Show signal trajectory** must be selected. When **Show signal trajectory** is selected, the **Signal trajectory** line style cannot be set to `no line`.

Reference Constellation

`Input` — Input port number

`1` (default) | integer

View or change the graphical properties for the symbol of the reference constellation for each port individually.

If none of the input are multichannel signals, then the graphical properties of the reference constellation can be adjusted by selecting the channel from the **Properties for channel** parameter.

Dependencies

To enable this parameter, set the **Number of input ports** parameter to a value greater than 1 and specify at least one multichannel input signal.

`Properties` — Select properties of reference constellation symbols

symbol properties

Select the graphical properties for the symbols of the reference constellation. Adjust the style, size, line width, and color of the marker.

Dependencies

To adjust reference constellation graphical properties, **Show reference constellation** must be selected.

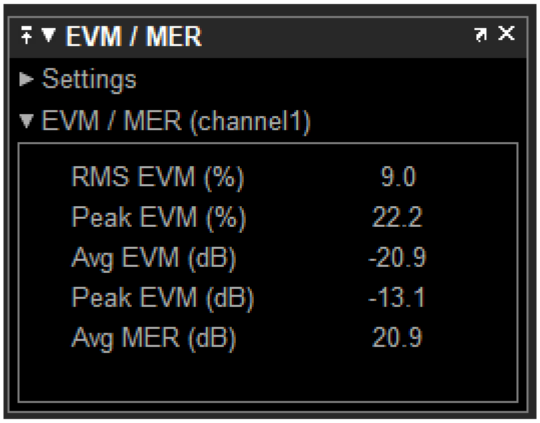
Block Characteristics

Data Types	`Boolean` | `double` | `enumerated` | `fixed point` | `integer` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`
Zero-Crossing Detection	`no`

More About

EVM / MER Measurements

The **EVM / MER** signal quality pane displays the measurement settings, and error vector magnitude (EVM) and modulation error ratio (MER) measurement calculation results for the specified trace selection.



RMS EVM (%)	9.0
Peak EVM (%)	22.2
Avg EVM (dB)	-20.9
Peak EVM (dB)	-13.1
Avg MER (dB)	20.9

- **EVM** — An error vector is a vector in the IQ plane from the ideal constellation point to the actual point at the receiver. The root mean square error vector magnitude, EVM_{RMS}, is measured for the average and peak constellation power.

  On the constellation diagram, you can display the EVM_{RMS} measurements normalized by either the `Average constellation power` or `Peak constellation power` method as computed using these algorithms.

EVM Normalization Method	Algorithm
`Average constellation power`	Average constellation power normalization: $$\text{EVM}_k = 100\sqrt{\frac{e_k}{P_{\text{avg}}}}$$ EVM_{RMS}, in percent, for average constellation power normalization: $$\text{EVM}_{\text{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\text{avg}}}}$$
`Peak constellation power`	Peak constellation power normalization $$\text{EVM}_k = 100\sqrt{\frac{e_k}{P_{\text{max}}}}$$ EVM_{RMS}, in percent, for peak constellation power normalization $$\text{EVM}_{\text{RMS}}(\%) = 100\sqrt{\frac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\text{max}}}}$$

The **EVM / MER** pane shows the average and peak EVM_{RMS} in both percent and decibels for the selected trace. The EVM reported in decibels is computed as EVM (dB) = 10-log10(EVM_{MS}) = 20-log10(EVM_{RMS}), where:

- $e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k is the in-phase value of the kth symbol in the input vector.
- Q_k is the quadrature phase value of the kth symbol in the input vector.
- I_k and Q_k represent ideal (reference) symbol values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbol values.
- N is the input vector length.
- P_{avg} is the value for `Average constellation power`.
- P_{max} is the value for `Peak constellation power`.
- $\text{EVM}_{\text{RMS}} = \sqrt{\text{EVM}_{\text{MS}}}$

The maximum EVM value in a vector is $\text{EVM}_{\text{max}} = \max\limits_{k \in [1, \ldots, N]} \{\text{EVM}_k\}$, where k is the kth symbol in a vector of length N.

For more information, see `comm.EVM`.

- **MER** — MER is the ratio of the average power of the transmitted signal to the average power of the error vector. The **EVM / MER** pane indicates average MER measurement result in decibels for the selected trace.

  MER is a measure of the SNR in a modulated signal, calculated in dB. The MER over N symbols is

$$\mathrm{MER} = 10 \cdot \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}\left(e_k\right)}\right)\mathrm{dB},$$

where:

- $e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k is the in-phase value of the kth symbol in the input vector.
- Q_k is the quadrature phase value of the kth symbol in the input vector.
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

For more information, see `comm.MER`.

Programmatic Configuration

You can programmatically configure the scope properties with callbacks or within scripts using a scope configuration object as, described in “Control Scope Blocks Programmatically” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is excluded from the generated code when code generation is performed on a system containing this block.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

See Also

Blocks

Eye Diagram

Objects

`comm.ConstellationDiagram` | `comm.EVM` | `comm.MER`

Functions

`scatterplot`

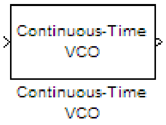
Topics

“Constellation Visualization”
“Control Scope Blocks Programmatically” (Simulink)
“View Simulation Results” (Simulink)
Line Properties

Introduced in R2013b

Continuous-Time VCO

Implement voltage-controlled oscillator



Library

Components sublibrary of Synchronization

Description

The Continuous-Time VCO (voltage-controlled oscillator) block generates a signal with a frequency shift from the **Quiescent frequency** parameter that is proportional to the input signal. The input signal is interpreted as a voltage. If the input signal is $u(t)$, then the output signal is

$$y(t) = A_c\cos\left(2\pi f_c t + 2\pi k_c\int_0^t u(\tau)d\tau + \varphi\right)$$

where A_c is the **Output amplitude** parameter, f_c is the **Quiescent frequency** parameter, k_c is the **Input sensitivity** parameter, and φ is the **Initial phase** parameter.

This block uses a continuous-time integrator to interpret the equation above.

The input and output are both sample-based scalar signals.

Parameters

Output amplitude

The amplitude of the output.

Quiescent frequency

The frequency of the oscillator output when the input signal is zero.

Input sensitivity

This value scales the input voltage and, consequently, the shift from the **Quiescent frequency** value. The units of **Input sensitivity** are Hertz per volt.

Initial phase

The initial phase of the oscillator in radians.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

See Also

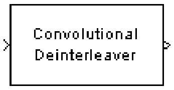
Blocks

Discrete-Time VCO

Introduced before R2006a

Convolutional Deinterleaver

Restore ordering of symbols that were permuted using shift registers



Library

Convolutional sublibrary of Interleaving

Description

The Convolutional Deinterleaver block recovers a signal that was interleaved using the Convolutional Interleaver block. Internally, this block uses a set of shift registers. The delay value of the k^{th} shift register is (N-k) times the **Register length step** parameter. The number of shift registers, N, is the value of the **Rows of shift registers** parameter. The parameters in the two blocks must have the same values.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

This block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point.

Parameters

Rows of shift registers

The number of shift registers that the block uses internally.

Register length step

The difference in symbol capacity of each successive shift register, where the last register holds zero symbols.

Initial conditions

Indicates the values that fill each shift register at the beginning of the simulation (except for the last shift register, which has zero delay).

- When you select a scalar value for **Initial conditions**, the value fills all shift registers (except for the last one)
- When you select a column vector with a length equal to the **Rows of shift registers** parameter, each entry fills the corresponding shift register.

The value of the first element of the **Initial conditions** parameter is unimportant, since the last shift register has zero delay.

Examples

For an example that uses this block, see “Convolutional Interleaving”.

Pair Block

Convolutional Interleaver

See Also

General Multiplexed Deinterleaver, Helical Deinterleaver

References

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

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

You can generate HDL code for the Convolutional Deinterleaver block using a shift-register-based implementation, or a RAM-based implementation.

The default implementation for the Convolutional Deinterleaver block is shift register-based. To suppress generation of reset logic, set the implementation parameter `ResetType` to`'none'`. When you set `ResetType` to`'none'`, reset is not applied to the shift registers.

When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

When you select the `RAM` implementation for a Convolutional Deinterleaver block, HDL Coder uses RAM resources instead of shift registers.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also "ResetType" (HDL Coder).

Restrictions

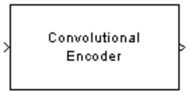
When you select the `RAM` implementation:

- Double or single data types are not supported for either input or output signals.
- You must set **Initial conditions** for the block to zero.
- At least two rows of interleaving are required.

Introduced before R2006a

Convolutional Encoder

Create convolutional code from binary data



Library

Convolutional sublibrary of Error Detection and Correction

Description

The Convolutional Encoder block encodes a sequence of binary input vectors to produce a sequence of binary output vectors. This block can process multiple symbols at a time.

This block can accept inputs that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Input and Output Sizes

If the encoder takes *k* input bit streams (that is, it can receive 2^k possible input symbols), the block input vector length is L*k for some positive integer L. Similarly, if the encoder produces *n* output bit streams (that is, it can produce 2^n possible output symbols), the block output vector length is L*n.

This block accepts a column vector input signal with any positive integer for L. For variable-size inputs, the L can vary during simulation. The operation of the block is governed by the **Operation mode** parameter.

For both its inputs and outputs for the data ports, the block supports `double`, `single`, `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, and `ufix1`. The port data types are inherited from the signals that drive the block. The input reset port supports `double` and `boolean` typed signals.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code”. You can use this parameter field in two ways:

- If you have a variable in the MATLAB workspace that contains the trellis structure, enter its name in the **Trellis structure** parameter. This way is preferable because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage described next.
- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, use a `poly2trellis` command in the **Trellis structure** parameter. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

```
poly2trellis(7,[171 133],171)
```

The encoder registers begin in the all-zeros state. Set the **Operation mode** parameter to `Reset on nonzero input via port` to reset all encoder registers to the all-zeros state during the simulation. This selection opens a second input port, labeled `Rst`, which accepts a scalar-valued input signal. When the input signal is nonzero, the block resets before processing the data at the first input port. To reset the block after it processes the data at the first input port, select **Delay reset action to next time step**.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode, the block retains the encoder states at the end of each input, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each input independently. The encoder states are reset to all-zeros state at the start of each input.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated (reset every frame)` or `Terminate trellis by appending bits`, the block's state resets at every input time step.

In `Terminate trellis by appending bits` mode, the block treats each input independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by $y = n \cdot (x + s)/k$, where x is the number of input bits, and s = constraint length – 1 (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`).

Note This block works for cases $k \geq 1$, where it has the same values for constraint lengths in each input stream (e.g., constraint lengths of [2 2] or [7 7] will work, but [5 4] will not).

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

Delay reset action to next time step

When you select **Delay reset action to next time step**, the Convolutional Encoder block resets after computing the encoded data. This check box only appears when you set the **Operation mode** parameter to `Reset on nonzero input via port`.

The delay in the reset action allows the block to support HDL code generation. In order to generate HDL code, you must have an HDL Coder license.

Output final state

When you select **Output final state**, the second output port signal specifies the output state for the block. The output signal is a scalar, integer value. You can select **Output final state** for all operation modes except `Terminate trellis by appending bits` .

Specify initial state via input port

When you select **Specify initial state via input port** the second input port signal specifies the starting state for every frame in the block. The input signal must be a scalar, integer value. **Specify initial state via input port** appears only in `Truncated` operation mode.

Puncture code

Selecting this option opens the field **Puncture vector**.

Puncture vector

Vector used to puncture the encoded data. The puncture vector is a pattern of `1`s and `0`s where the `0`s indicate the punctured bits. This field appears when you select **Punctured code**.

Puncture Pattern Examples

For some commonly used puncture patterns for specific rates and polynomials, see the last three references listed on this page.

See Also

Viterbi Decoder, APP Decoder

References

[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, 1992.

[3] Yasuda, Y., et. al., “High rate punctured convolutional codes for soft decision Viterbi decoding,” *IEEE Transactions on Communications*, Vol. COM-32, No. 3, pp 315–319, March 1984.

[4] Haccoun, D., and Begin, G., “High-rate punctured convolutional codes for Viterbi and Sequential decoding,” *IEEE Transactions on Communications*, Vol. 37, No. 11, pp 1113–1125, Nov. 1989.

[5] Begin, G., et.al., “Further results on high-rate punctured convolutional codes for Viterbi and sequential decoding,” *IEEE Transactions on Communications*, Vol. 38, No. 11, pp 1922–1928, Nov. 1990.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- Input data requirements:
  - Must be sample-based,
  - Must have a `boolean` or `ufix1` data type.
- HDL Coder supports only the following coding rates:
  - ½ to 1/7
  - 2/3
- The coder supports only constraint lengths for 3 to 9.
- Specify **Trellis structure** by the `poly2trellis` function.
- The coder supports the following **Operation mode** settings:
  - `Continuous`
  - `Reset on nonzero input via port`

    If you select this mode, you must select the **Delay reset action to next time step** option. When you select this option, the Convolutional Encoder block finishes its current computation before executing a reset.
- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

Introduced before R2006a

Convolutional Interleaver

Permute input symbols using set of shift registers

Library

Convolutional sublibrary of Interleaving

Description

The Convolutional Interleaver block permutes the symbols in the input signal. Internally, it uses a set of shift registers. The delay value of the kth shift register is (k-1) times the **Register length step** parameter. The number of shift registers is the value of the **Rows of shift registers** parameter.

The **Initial conditions** parameter indicates the values that fill each shift register at the beginning of the simulation (except for the first shift register, which has zero delay). If **Initial conditions** is a scalar, then its value fills all shift registers except the first; if **Initial conditions** is a column vector whose length is the **Rows of shift registers** parameter, then each entry fills the corresponding shift register. The value of the first element of the **Initial conditions** parameter is unimportant, since the first shift register has zero delay.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

Parameters

Rows of shift registers

The number of shift registers that the block uses internally.

Register length step

The number of additional symbols that fit in each successive shift register, where the first register holds zero symbols.

Initial conditions

The values that fill each shift register when the simulation begins.

Examples

For an example that uses this block, see “Convolutional Interleaving”.

Pair Block

Convolutional Deinterleaver

See Also

General Multiplexed Interleaver, Helical Interleaver

References

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

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

You can generate HDL code for the Convolutional Interleaver block using a shift-register-based implementation, or a RAM-based implementation.

The default implementation for the Convolutional Interleaver block is shift register-based. To suppress generation of reset logic, set the implementation parameter `ResetType` to`'none'`. When you set `ResetType` to`'none'`, reset is not applied to the shift registers.

When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

When you select the `RAM` implementation for a Convolutional Interleaver block, HDL Coder uses RAM resources instead of shift registers.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also "ResetType" (HDL Coder).

Restrictions

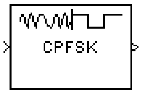
When you select the `RAM` implementation:

- Double or single data types are not supported for either input or output signals.
- You must set **Initial conditions** for the block to zero.
- At least two rows of interleaving are required.

Introduced before R2006a

CPFSK Demodulator Baseband

Demodulate CPFSK-modulated data



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The CPFSK Demodulator Baseband block demodulates a signal that was modulated using the continuous phase frequency shift keying method. The input to this block is a baseband representation of the modulated signal. The **M-ary number** parameter, *M*, is the size of the input alphabet. *M* must have the form 2^K for some positive integer *K*.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, then the block produces odd integers between -(M-1) and M-1.

When you set the **Output type** parameter to `Bit`, then the block produces groupings of *K* bits. Each grouping is called a binary *word*.

In binary output mode, the block first maps each input symbol to an intermediate value as in the integer output mode. The block then maps the odd integer k to the nonnegative integer (k+M-1)/2. Finally, the block maps each nonnegative integer to a binary word, using a mapping that depends on whether the **Symbol set ordering** parameter is set to `Binary` or `Gray`.

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is *K* times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, *D*, in this block is the number of trellis branches that the algorithm uses to construct each traceback path. *D* influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the “five-times-the-constraint-length” rule, which corresponds to 5·log2(*numStates*).

For the definition of the number of states, see CPM Demodulator Baseband Help page.

Parameters

M-ary number

The size of the alphabet.

Output type

Determines whether the output consists of integers or groups of bits.

Symbol set ordering

Determines how the block maps each integer to a group of output bits. This field is active only when **Output type** is set to `Bit`.

Modulation index

Specify the modulation index {h_i}. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see “Upsample Signals and Rate Changes” in *Communications Toolbox User*'s *Guide*.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth

The number of trellis branches that the CPFSK Demodulator Baseband block uses to construct each traceback path.

Output datatype

The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

CPFSK Modulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

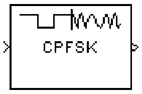
See Also

Blocks
CPFSK Modulator Baseband | CPM Demodulator Baseband | M-FSK Demodulator Baseband | Viterbi Decoder

Introduced before R2006a

CPFSK Modulator Baseband

Modulate using continuous phase frequency shift keying method



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The CPFSK Modulator Baseband block modulates a signal using the continuous phase frequency shift keying method. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, *M*, represents the size of the input alphabet. *M* must have the form 2^K for some positive integer *K*.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the block accepts odd integers between -(M-1) and M-1.

When you set the **Input type** parameter to `Bit`, the block accepts groupings of *K* bits. Each grouping is called a binary *word*. The input vector length must be an integer multiple of *K*.

In binary input mode, the block maps each binary word to an integer between 0 and M-1, using a mapping scheme that depends on whether you set the **Symbol set ordering** parameter to `Binary` or `Gray`. The block then maps the integer *k* to the intermediate value 2k-(M-1) and proceeds as if it operates in the integer input mode. For more information, see “Integer-Valued Signals and Binary-Valued Signals”.

This block accepts a scalar-valued or column vector input signal. If you set **Input type** to `Bit`, then the input signal can also be a vector of length *K*.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input must be a column vector with a width that is an integer multiple of *K*, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Parameters

M-ary number

The size of the alphabet.

Input type

Indicates whether the input consists of integers or groups of bits.

Symbol set ordering

Determines how the block maps each group of input bits to a corresponding integer. This field is active only when **Input type** is set to `Bit`.

Modulation index

Specify the modulation index {h_i}. The default is `0.5`. The value of this property must be a real, nonnegative scalar or column vector.

This block supports multi-h **Modulation index**. See CPM Modulator Baseband for details.

Phase offset (rad)

The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes”.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Output data type

Select the data type of the output signal. The output data type can be `single` or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (When **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

CPFSK Demodulator Baseband

See Also

CPM Modulator Baseband, M-FSK Modulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

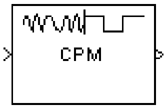
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

CPM Demodulator Baseband

Demodulate CPM-modulated data

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / CPM



Description

The CPM Demodulator Baseband block demodulates a signal that was modulated using continuous phase modulation (CPM).

CPM is a modulation method with memory. The block processing includes a correlator followed by a maximum-likelihood sequence detector (MLSD) that searches the paths through the state trellis for the minimum Euclidean distance path. The block uses the Viterbi algorithm to perform MLSD.

For more information about this demodulation and the filtering applied, see “CPM Demodulation” on page 5-146 and “Pulse Shape Filtering” on page 5-147.

Ports

Input

`In` — Input signal
scalar | column vector

Input signal, specified as a scalar or column vector. The length of the input signal must be an integer multiple of the number of samples per symbol specified in the **Samples per symbol** parameter. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 5-148.

Data Types: `double` | `single`

Output

`Out` — Output signal
scalar | column vector

Output signal, returned as a scalar or column vector. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 5-148.

Supported Data Types

- Double-precision floating point
- Boolean (when **Output type** is set to `Bit`)
- 8-, 16-, and 32-bit signed integers (when **Output type** is set to `Integer`)

Data Types: `double` | `Boolean` | `int8` | `int16` | `int32`

For more information on the processing rates, see “Single-Rate Processing” on page 5-148, and “Multirate Processing” on page 5-149.

Parameters

`M-ary number` — Modulation order

4 (default) | positive integer

Modulation order indicating the alphabet size, specified as a positive integer that is a nonzero power of two. M must have the form 2^K for some positive integer K, where K is the number of bits per symbol.

`Output type` — Determines whether output consists of integers or groups of bits

`Integer` (default) | `Bit`

Determines whether the output consists of integers or groups of bits, specified as `Integer` or `Bit`.

`Symbol set ordering` — Bit mapping

`Binary` (default) | `Gray`

Bit mapping, specified as `Binary` or `Gray`.

This parameter determines how the block maps each integer to a group of output bits. For more information, see “Integer-Valued and Binary-Valued Output Signals” on page 5-148.

Dependencies

To enable this parameter, set **Output type** to `Bit`.

`Modulation index` — Modulation index {h_i}

`0.5` (default) | nonnegative scalar | column vector

Modulation index {h_i}, specified as a nonnegative scalar or column vector.

{h} represents a sequence of modulation indices. For more information, see “CPM Demodulation” on page 5-146.

`Frequency pulse shape` — Type of pulse shaping

`Rectangular` (default) | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`

Type of pulse shaping used to smooth the phase transitions of the modulated signal, specified as `Rectangular`, `Raised Cosine`, `Spectral Raised Cosine`, `Gaussian`, or `Tamed FM`. For more information on the filtering options, see “Pulse Shape Filtering” on page 5-147.

`Main lobe pulse duration (symbol intervals)` — Number of symbol intervals of largest lobe of the spectral raised cosine pulse

1 (default) | positive scalar

Number of symbol intervals of the largest lobe of the spectral raised cosine pulse, specified as a positive scalar.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`Rolloff` — Rolloff factor of spectral raised cosine pulse shape

`0.2` (default) | nonnegative scalar

Rolloff factor of the spectral raised cosine pulse shape, specified as a scalar from 0 to 1.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`BT product` — Product of bandwidth and time

`0.3` (default) | nonnegative scalar

Product of bandwidth and time, specified as a nonnegative scalar. Use **BT product** to reduce the bandwidth, at the expense of increased intersymbol interference.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Gaussian`.

`Pulse length (symbol intervals)` — Frequency pulse length

`1` (default) | positive scalar

Frequency pulse shape length, specified as a positive scalar. Refer to *LT* in “Pulse Shape Filtering” on page 5-147 for more information on the frequency pulse length.

`Symbol prehistory` — Data symbols used before the start of simulation

`1` (default) | scalar | vector

Data symbols used before the start of simulation in reverse chronological order. If **Symbol prehistory** is a vector, then its length must be one less than the **Pulse length** parameter value.

`Phase offset (rad)` — Initial phase offset

`0` (default) | scalar

Initial phase offset of output in radians, specified as a scalar.

`Samples per symbol` — Symbol sampling rate

`8` (default) | positive scalar

Symbol sampling rate, specified as a positive scalar. This parameter represents the number of samples output for each integer or binary word input. For all nonbinary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes”.

`Rate options` — Block processing rate

`Enforce single-rate processing` (default) | `Allow multirate processing`

Block processing rate, specified as one of these options:

- `Enforce single-rate processing` — The input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).

- `Allow multirate processing` — The input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

`Traceback depth` — Number of trellis branches

16 (default) | positive integer

Number of trellis branches used to construct each traceback path, specified as a positive integer. For more information, see "Traceback Depth and Output Delays" on page 5-149.

`Output data type` — Output data type

`double` (default) | `boolean` | `int8` | `int16` | `int32`

Output data type, specified as `double`, `boolean`, `int8`, `int16`, or `int32`. For more information, see **Supported Data Types** in `Out`.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

CPM Demodulation

The CPM demodulation processing consists of a correlator followed by a maximum-likelihood sequence detector (MLSD) that searches the paths through the state trellis for the minimum Euclidean distance path. When the modulation index is rational ($h = m/p$), there are a finite number of phase states in the symbol. The block uses the Viterbi algorithm to perform MLSD.

$\{h_\mathrm{i}\}$ represents a sequence of modulation indices that moves cyclically through a set of indices $\{h_0, h_1, h_2, \ldots, h_{H\text{-}1}\}$.

- $h_\mathrm{i} = m_\mathrm{i}/p_\mathrm{i}$ represents the modulation index in proper rational form.
- m_i represents the numerator of the modulation index.
- p_i represents the denominator of the modulation index.
- m_i and p_i are relatively prime positive numbers.
- The least common multiple (LCM) of $\{p_0, p_1, p_2, \ldots, p_{H\text{-}1}\}$ is denoted as p.
- $h_\mathrm{i} = m'_\mathrm{i} / p$

$\{h_\mathrm{i}\}$ determines the number of phase states:

$$numPhaseStates = \begin{Bmatrix} p,\ for\ all\ even\ m'_i \\ 2p,\ for\ any\ odd\ m'_i \end{Bmatrix}$$

and affects the number of trellis states:

$numStates = numPhaseStates{*}M^{(L\text{-}1)}$

- L represents the **Pulse length**.
- M represents the **M-ary number**.

CPM Modulation

The input to the demodulator is a baseband representation of the modulated signal:

$$s(t) \ = \ \exp\left[j\,2\pi \sum_{i\ =\ 0}^{n} \alpha_i h_i q(t - iT)\right],$$
$$nT < \ t < (n+1)T$$

where:

- $\{\alpha_\mathrm{i}\}$ represents a sequence of M-ary data symbols selected from the alphabet ±1, ±3, $\pm(M–1)$.
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_\mathrm{i}\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{h_0, h_1, h_2, ..., h_{H\text{-}1}\}$. When H=1, there is only one modulation index, h_0, which is denoted as h.

Pulse Shape Filtering

Continuous phase modulation uses pulse shaping to smooth the phase transitions of the modulated signal. The function $q(t)$ is the phase response obtained from the frequency pulse, $g(t)$, through this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to these pulse shape expressions, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t) = \begin{cases} \dfrac{1}{2LT}, & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$
`Raised Cosine`	$g(t) = \begin{cases} \dfrac{1}{2LT}\left[1-\cos\left(\dfrac{2\pi t}{LT}\right)\right], & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$
`Spectral Raised Cosine`	$g(t) = \dfrac{1}{L_{main}T} \dfrac{\sin\left(\dfrac{2\pi t}{L_{main}T}\right)}{\dfrac{2\pi t}{L_{main}T}} \dfrac{\cos\left(\beta \dfrac{2\pi t}{L_{main}T}\right)}{1-\left(\dfrac{4\beta}{L_{main}T}t\right)^2}, \quad 0 \le \beta \le 1$

Pulse Shape	Expression
`Gaussian`	$g(t)=\frac{1}{2T}\left\{Q\left[2\pi B_b\frac{t-\frac{T}{2}}{\sqrt{\ln 2}}\right]-Q\left[2\pi B_b\frac{t+\frac{T}{2}}{\sqrt{\ln 2}}\right]\right\}$, where $Q(t)=\int_t^{\infty}\frac{1}{\sqrt{2\pi}}e^{-\tau^2/2}d\tau$
`Tamed FM` (tamed frequency modulation)	$g(t)=\frac{1}{8}\left[g_0(t-T)+2g_0(t)+g_0(t+T)\right]$, where $g_0(t)\approx\frac{1}{T}\Bigg[\frac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}}-\frac{\pi^2}{24}\frac{2\sin\left(\frac{\pi t}{T}\right)-\frac{2\pi t}{T}\cos\left(\frac{\pi t}{T}\right)-\left(\frac{\pi t}{T}\right)^2\sin\left(\frac{\pi t}{T}\right)}{\left(\frac{\pi t}{T}\right)^3}$

- L_{main} is the main lobe pulse duration in symbol intervals.
- β is the rolloff factor of the spectral raised cosine.
- $B_{\text{b}}$is the product of the bandwidth and the Gaussian pulse.
- The duration of the pulse, *LT*, is the pulse length in symbol intervals. As defined by the expressions, the `Spectral Raised Cosine`, `Gaussian`, and `Tamed FM` pulse shapes have infinite length. For all practical purposes, *LT* specifies the truncated finite length.

For more information on pulse shape filtering, see [1]

Integer-Valued and Binary-Valued Output Signals

When the **Output type** parameter is set to `Integer`:

- The block produces odd integers between –(*M*–1) and *M*–1. The modulation order, *M*, is specified by the **M-ary number** parameter.
- The **Output datatype** parameter cannot be set to `boolean`.

When the **Output type** parameter is set to `Bit`:

- The block produces groupings of *K* bits. Each grouping is called a binary word.
- The **Output datatype** can only be `double` or `boolean`.
- In binary output mode, the block processing follows this procedure:

  1. Maps each input symbol to an intermediate value, as in the integer output mode.
  2. Maps the odd integer *k* to the nonnegative integer (*k*+*M*–1)/2.
  3. Maps each nonnegative integer to a binary word, using `Binary` or `Gray` mapping, as specified by the **Symbol set ordering** parameter.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is *K* times the number of input symbols.

- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

The **Traceback depth** parameter, *D*, is the number of trellis branches used to construct each traceback path. *D* influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When the **Rate options** parameter is set to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay vector consists of D+1 zero-value symbols.
- When the **Rate options** parameter is set to `Enforce single-rate processing`, the delay vector consists of *D* zero-value symbols.

The optimal **Traceback depth** parameter value depends on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the five-times-the-constraint-length rule, which corresponds to 5 · log2(*numStates*).

For a binary raised cosine pulse shape with a pulse length of 3 and h=2/3, applying this rule ($5*\log2(3*2^2)$ = 18) gives a result that is close to the optimum value of 20.

Pair Block

CPM Modulator Baseband — Modulates data using continuous phase modulation.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CPFSK Demodulator Baseband | CPM Modulator Baseband | GMSK Demodulator Baseband | MSK Demodulator Baseband | Viterbi Decoder

Topics

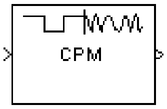
“CPM Phase Tree”

Introduced before R2006a

CPM Modulator Baseband

Modulate using continuous phase modulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / CPM



Description

The CPM Modulator Baseband block modulates an input signal using continuous phase modulation (CPM). The output of the modulator is a baseband representation of the modulated signal:

$$s(t) = \exp\left[j2\pi \sum_{i=0}^{n} \alpha_i h_i q(t - iT)\right],$$
$$nT < t < (n+1)T$$

where:

- $\{\alpha_\mathrm{i}\}$ represents a sequence of M-ary data symbols selected from the alphabet ±1, ±3, ±(M–1).
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_\mathrm{i}\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{h_0, h_1, h_2, ..., h_{H\text{-}1}\}$. When H=1, there is only one modulation index, h_0, which is denoted as h.

For more information about this modulation and the filtering applied, see “CPM Modulation” on page 5-154 and “Pulse Shape Filtering” on page 5-155.

Ports

Input

In — Input signal
scalar | column vector

Input signal, specified as a scalar or column vector.

When the **Input type** parameter is set to `Integer`, the block accepts odd integers between –(M–1) and M–1. M represents the **M-ary number** parameter.

When the **Input type** parameter is set to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits. K is the number of bits per symbol and M is the modulation order. The input vector length must be an integer multiple of K. The block maps each group of K bits onto a symbol, as specified by the **Symbol set ordering** parameter. For each group of K bits, the block outputs one modulated symbol, oversampled by the **Samples per symbol** parameter value.

Supported Data Types

- Double-precision floating point
- Boolean (when **Input type** is set to `Bit`)
- 8-, 16-, and 32-bit signed integers (when **Input type** is set to `Integer`)

Data Types: `double` | `Boolean` | `int8` | `int16` | `int32`

Output

`Out` — Output signal

scalar | column vector

Output signal, returned as a scalar or column vector.

- When the **Input type** parameter is set to `Integer`, the block outputs one modulated symbol for each input symbol.
- When the **Input type** parameter is set to `Bit`, the block outputs one modulated symbol for each group of *K* bits.

In both cases, the modulated symbols are oversampled by the **Samples per symbol** parameter value.

Data Types: `double` | `single`

For more information on the processing rates, see “Single-Rate Processing” on page 5-156, and “Multirate Processing” on page 5-156.

Parameters

`M-ary number` — Modulation order

4 (default) | positive integer

Modulation order indicating the alphabet size, specified as a positive integer that is a nonzero power of two. M must have the form 2^K for some positive integer K, where K is the number of bits per symbol.

`Input type` — Integer or group of bits input indicator

`Integer` (default) | `Bit`

Indicates whether the input consists of integers or groups of bits, specified as `Integer` or `Bit`.

`Symbol set ordering` — Bit mapping

`Binary` (default) | `Gray`

Bit mapping, specified as `Binary` or `Gray`. For more information, see “Symbol Sets” on page 5-156.

Dependencies

To enable this parameter, set **Input type** to `Bit`.

`Modulation index` — Modulation index $\{h_i\}$

`0.5` (default) | nonnegative scalar | column vector

Modulation index $\{h_i\}$, specified as a nonnegative scalar or column vector.

$\{h\}$ represents a sequence of modulation indices. For more information, see “CPM Modulation” on page 5-154.

`Frequency pulse shape` — Type of pulse shaping

`Rectangular` (default) | `Raised Cosine` | `Spectral Raised Cosine` | `Gaussian` | `Tamed FM`

Type of pulse shaping used to smooth the phase transitions of the modulated signal, specified as `Rectangular`, `Raised Cosine`, `Spectral Raised Cosine`, `Gaussian`, or `Tamed FM`. For more information on the filtering options, see “Pulse Shape Filtering” on page 5-155.

`Main lobe pulse duration (symbol intervals)` — Number of symbol intervals of largest lobe of spectral raised cosine pulse

`1` (default) | positive scalar

Number of symbol intervals of the largest lobe of the spectral raised cosine pulse, specified as a positive scalar.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`Rolloff` — Rolloff factor of spectral raised cosine pulse shape

`0.2` (default) | nonnegative scalar

Rolloff factor of the spectral raised cosine pulse, specified as a scalar from 0 to 1.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Spectral Raised Cosine`.

`BT product` — Product of bandwidth and time

`0.3` (default) | nonnegative scalar

Product of bandwidth and time, specified as a nonnegative scalar. Use **BT product** to reduce the bandwidth, at the expense of increased intersymbol interference.

Dependencies

To enable this parameter, set **Frequency pulse shape** to `Gaussian`.

`Pulse length (symbol intervals)` — Frequency pulse length

`1` (default) | positive scalar

Frequency pulse length, specified as a positive scalar. Refer to *LT* in “Pulse Shape Filtering” on page 5-155 for more information on the frequency pulse length.

`Symbol prehistory` — Data symbols used before the start of simulation

`1` (default) | scalar | vector

Data symbols used before the start of simulation, specified as a scalar or vector in reverse chronological order. If **Symbol prehistory** is a vector, then its length must be one less than the **Pulse length (symbol intervals)** parameter value.

`Phase offset (rad)` — Initial phase offset

`0` (default) | scalar

Initial phase offset of output in radians, specified as a scalar.

`Samples per symbol` — Symbol sampling rate
8 (default) | positive scalar

Symbol sampling rate, specified as a positive scalar. This parameter represents the number of samples output for each integer or binary word input. For all nonbinary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes”.

`Rate options` — Block processing rate
`Enforce single-rate processing` (default) | `Allow multirate processing`

Block processing rate, specified as one of these options:

- `Enforce single-rate processing` — The input and output signals have the same sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — The input and output signals have different sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

`Output data type` — Output data type
`double` (default) | `single`

Output data type, specified as `double` or `single`.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

CPM Modulation

The output of the modulator is a baseband representation of the modulated signal:

$$s(t) = \exp\left[j2\pi\sum_{i=0}^{n}\alpha_i h_i q(t - iT)\right],$$
$$nT < t < (n+1)T$$

where:

- $\{\alpha_i\}$ represents a sequence of M-ary data symbols selected from the alphabet ±1, ±3, $\pm(M–1)$.
- M must have the form 2^K for some positive integer K, where M is the modulation order and specifies the size of the symbol alphabet.
- $\{h_i\}$ represents a sequence of modulation indices and h_i moves cyclically through a set of indices $\{h_0, h_1, h_2, ..., h_{H\text{-}1}\}$. When H=1, there is only one modulation index, h_0, which is denoted as h.

h_i specifies the modulation index. When h_i varies from interval to interval, the block operates in multi-h. To ensure a finite number of phase states, h_i must be a rational number.

Pulse Shape Filtering

Continuous phase modulation uses pulse shaping to smooth the phase transitions of the modulated signal. The function $q(t)$ is the phase response obtained from the frequency pulse, $g(t)$, through this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to these pulse shape expressions, $g(t)$.

Pulse Shape	**Expression**
`Rectangular`	$g(t) = \begin{cases} \dfrac{1}{2LT}, & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$
`Raised Cosine`	$g(t) = \begin{cases} \dfrac{1}{2LT}\left[1-\cos\left(\dfrac{2\pi t}{LT}\right)\right], & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$
`Spectral Raised Cosine`	$g(t) = \dfrac{1}{L_{main}T} \dfrac{\sin\left(\dfrac{2\pi t}{L_{main}T}\right)}{\dfrac{2\pi t}{L_{main}T}} \dfrac{\cos\left(\beta \dfrac{2\pi t}{L_{main}T}\right)}{1-\left(\dfrac{4\beta}{L_{main}T}t\right)^2}, \quad 0 \le \beta \le 1$
`Gaussian`	$g(t) = \dfrac{1}{2T}\left\{Q\left[2\pi B_b \dfrac{t-\frac{T}{2}}{\sqrt{\ln 2}}\right] - Q\left[2\pi B_b \dfrac{t+\frac{T}{2}}{\sqrt{\ln 2}}\right]\right\}$, where $Q(t) = \int_t^{\infty} \dfrac{1}{\sqrt{2\pi}} e^{-\tau^2/2} d\tau$
`Tamed FM` (tamed frequency modulation)	$g(t) = \frac{1}{8}\left[g_0(t-T) + 2g_0(t) + g_0(t+T)\right]$, where $g_0(t) \approx \dfrac{1}{T}\left[\dfrac{\sin(\frac{\pi t}{T})}{\frac{\pi t}{T}} - \dfrac{\pi^2}{24} \dfrac{2\sin\left(\frac{\pi t}{T}\right) - \frac{2\pi t}{T}\cos\left(\frac{\pi t}{T}\right) - \left(\frac{\pi t}{T}\right)^2 \sin\left(\frac{\pi t}{T}\right)}{\left(\frac{\pi t}{T}\right)^3}\right.$

- L_{main} is the main lobe pulse duration in symbol intervals.
- β is the rolloff factor of the spectral raised cosine.
- $B_b$is the product of the bandwidth and the Gaussian pulse.
- The duration of the pulse, LT, is the pulse length in symbol intervals. As defined by the expressions, the `Spectral Raised Cosine`, `Gaussian`, and `Tamed FM` pulse shapes have infinite length. For all practical purposes, LT specifies the truncated finite length.

For more information on pulse shape filtering, see [1].

Symbol Sets

In binary input mode, the block processing follows this procedure:

1. Maps each binary word to k, an integer from 0 to M–1. The binary word mapping options are `Binary` or `Gray`, as specified by the **Symbol set ordering** parameter.
2. Maps k to the intermediate value $2k–(M–1)$
3. Proceeds with block processing as in the integer input mode.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. In this mode, the input to the block can be multiple symbols. The block implicitly implements the rate change by making a size change at the output when compared to the input.

- When you set **Input type** to `Integer`, the input can be a scalar or a column vector with the length equal to the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of the number of bits per symbol.

The output width equals $N_{\text{Sym}} \times N_{\text{SPS}}$, where N_{Sym} is the number of symbols in the frame and N_{SPS} is the number of samples per symbol.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals T_{Sym} / N_{SPS}, where T_{Sym} is the symbol period and N_{SPS} is the number of samples per symbol.

Pair Block

CPM Demodulator Baseband — Demodulates continuous phase modulated data.

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
CPFSK Modulator Baseband | CPM Demodulator Baseband | GMSK Modulator Baseband | MSK Modulator Baseband

Topics
"CPM Phase Tree"
"CPM Modulation"

Introduced before R2006a

CPM Phase Recovery

(Removed) Recover carrier phase using 2P-Power method

Note CPM Phase Recovery has been removed. Use the Carrier Synchronizer block instead.

Library

Carrier Phase Recovery sublibrary of Synchronization

Description

The CPM Phase Recovery block recovers the carrier phase of the input signal using the 2P-Power method. This feedforward, non-data-aided, clock-aided method is suitable for systems that use these types of baseband modulation: continuous phase modulation (CPM), minimum shift keying (MSK), continuous phase frequency shift keying (CPFSK), and Gaussian minimum shift keying (GMSK). This block is suitable for use with blocks in the Baseband Continuous Phase Modulation library.

If you express the modulation index for CPM as a proper fraction, h = K / P, then P is the number to which the name "2P-Power" refers. The observation interval parameter must be an integer multiple of the input signal vector length.

The 2P-Power method assumes that the carrier phase is constant over a series of consecutive symbols, and returns an estimate of the carrier phase for the series. The **Observation interval** parameter is the number of symbols for which the carrier phase is assumed constant. This number must be an integer multiple of the input signal's vector length.

Input and Output Signals

This block accepts a scalar or column vector input signal of type `double` or `single`. The input signal represents a baseband signal at the symbol rate, so it must be complex-valued and must contain one sample per symbol.

The outputs are as follows:

- The output port labeled `Sig` gives the result of rotating the input signal counterclockwise, where the amount of rotation equals the carrier phase estimate. The `Sig` output is thus a corrected version of the input signal, and has the same sample time and vector size as the input signal.
- The output port labeled `Ph` outputs the carrier phase estimate, in degrees, for all symbols in the observation interval. The `Ph` output is a scalar signal.

> **Note** Because the block internally computes the argument of a complex number, the carrier phase estimate has an inherent ambiguity. The carrier phase estimate is between -90/P and 90/P degrees and might differ from the actual carrier phase by an integer multiple of 180/P degrees.

Delays and Latency

The block's algorithm requires it to collect symbols during a period of length **Observation interval** before computing a single estimate of the carrier phase. Therefore, each estimate is delayed by

Observation interval symbols and the corrected signal has a latency of **Observation interval** symbols, relative to the input signal.

Parameters

P

The denominator of the modulation index for CPM (h = K / P) when expressed as a proper fraction.

Observation interval

The number of symbols for which the carrier phase is assumed constant. The observation interval parameter must be an integer multiple of the input signal vector length.

When this parameter is exactly equal to the vector length of the input signal, then the block always works. When the integer multiple is not equal to `1`, on the **Simulation** tab, select **Model Settings**. Then in the **Solver** > **Solver selection** section, choose **Type:** `Fixed-step` and clear the **Treat each discrete rate as a separate task** checkbox.

Algorithm

If the symbols occurring during the observation interval are x(1), x(2), x(3),..., x(L), then the resulting carrier phase estimate is

$$\frac{1}{2P}\arg\left\{\sum_{k=1}^{L}(x(k))^{2P}\right\}$$

where the arg function returns values between -180 degrees and 180 degrees.

References

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

See Also

M-PSK Phase Recovery, CPM Modulator Baseband

Compatibility Considerations

CPM Phase Recovery has been removed

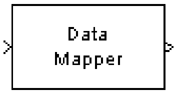
Errors starting in R2020a

CPM Phase Recovery has been removed. Use Carrier Synchronizer instead.

Introduced before R2006a

Data Mapper

Map integer symbols from one coding scheme to another
Library: Communications Toolbox / Utility Blocks



Description

The Data Mapper block accepts integer inputs and maps them to integer outputs. The mapping types include: binary to Gray coded, Gray coded to binary, and user defined. Additionally, a pass through option is available.

Gray coding is an ordering of binary numbers such that all adjacent numbers differ by only one bit.

Input/Output Ports

Input

`Port_1` — Input port
scalar | column vector | matrix

Input signal, specified as a scalar, vector, or matrix of integers. Elements of the input signal must be nonnegative values. The block truncates noninteger values to integer values. When the input is a matrix, the columns are treated as independent channels.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

`Port_2` — Output signal
scalar | column vector | matrix

Output signal, returned as a scalar, column vector, or matrix. The dimensions of the output signal match those of the input signal.

Data Types: `double` | `single` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Parameters

`Mapping mode` — Mapping mode
`Binary to Gray` (default) | `Gray to Binary` | `User Defined` | `Straight through`

Mapping mode, specified as one of the four options. The mapping for the `Binary to Gray` and the `Gray to Binary` modes are shown in the following table when the inputs range from 0 to 7.

Binary to Gray Mode		Gray to Binary Mode	
Input	**Output**	**Input**	**Output**
0	0 *(000)*	0 *(000)*	0
1	1 *(001)*	1 *(001)*	1
2	3 *(011)*	2 *(010)*	3
3	2 *(010)*	3 *(011)*	2
4	6 *(110)*	4 *(100)*	7
5	7 *(111)*	5 *(101)*	6
6	5 *(101)*	6 *(110)*	4
7	4 *(100)*	7 *(111)*	5

When you select the `User Defined` mode, you can use any arbitrary mapping by providing a vector to specify the output ordering. When you select the `Straight Through` mode, the output equals the input.

`Symbol set size (M)` — Symbol set size
8 (default) | positive integer

Symbol set size, specified as a positive integer. This parameter restricts the inputs and outputs to integers in the range of 0 to M-1.

`Mapping vector` — Maps input elements to the output elements
`[0 1 3 2 7 6 4 5]` (default) | vector

Mapping vector, specified as vector of nonnegative integers whose length equals . This parameter defines the relationship between the input and output integers. For example, the vector `[1 5 0 4 2 3]` defines the following mapping:

0 → 1
1 → 5
2 → 0
3 → 4
4 → 2
5 → 3

Block Characteristics

Data Types	`double` | `fixed point` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

`bin2gray` | `gray2bin`

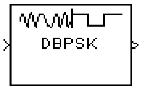
Topics

"Phase Modulation"

Introduced before R2006a

DBPSK Demodulator Baseband

Demodulate DBPSK-modulated data



Library

PM, in Digital Baseband sublibrary of Modulation

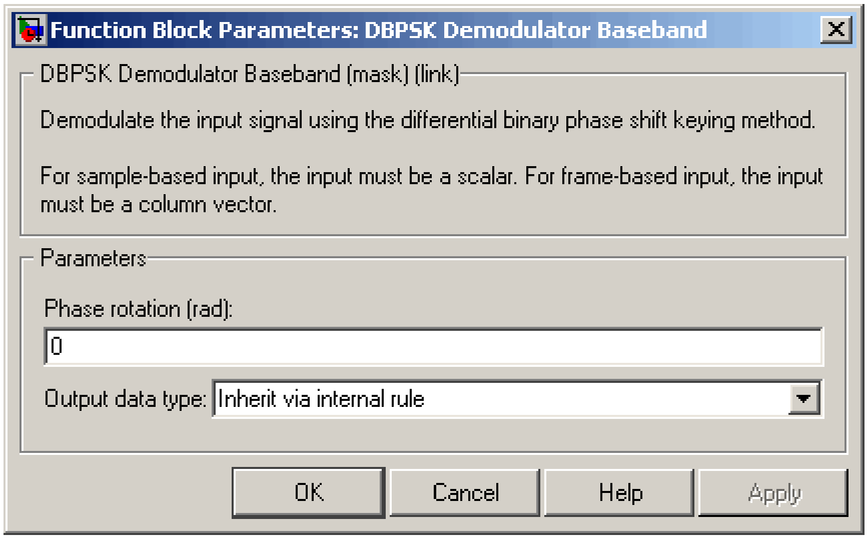
Description

The DBPSK Demodulator Baseband block demodulates a signal that was modulated using the differential binary phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. The block compares the current symbol to the previous symbol. It maps phase differences of θ and π+θ, respectively, to outputs of 0 and 1, respectively, where θ is the **Phase rotation** parameter. The first element of the block's output is the initial condition of zero because there is no previous symbol with which to compare the first symbol.

This block accepts a scalar or column vector input signal. The input signal can be of data types `single` and `double`. For information about the data types each block port supports, see "Supported Data Types" on page 5-164.

Dialog Box


Function Block Parameters: DBPSK Demodulator Baseband

DBPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential binary phase shift keying method.

For sample-based input, the input must be a scalar. For frame-based input, the input must be a column vector.

Parameters

Phase rotation (rad): 0

Output data type: Inherit via internal rule

Phase rotation (rad)

This phase difference between the current and previous modulated symbols results in an output of zero.

Output data type

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For additional information, see “Supported Data Types” on page 5-164.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

DBPSK Modulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BPSK Demodulator Baseband | DQPSK Demodulator Baseband | M-DPSK Demodulator Baseband

Introduced before R2006a

DBPSK Modulator Baseband

Modulate using differential binary phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

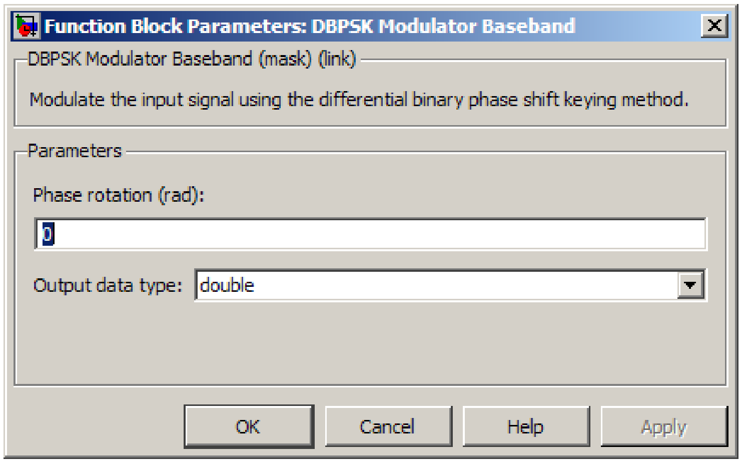
The DBPSK Modulator Baseband block modulates using the differential binary phase shift keying method. The output is a baseband representation of the modulated signal.

This block accepts a scalar or column vector input signal. The input must be a discrete-time binary-valued signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-166.

The following rules govern this modulation method when the **Phase rotation** parameter is θ:

- If the first input bit is 0 or 1, respectively, then the first modulated symbol is exp(jθ) or -exp(jθ), respectively.
- If a successive input bit is 0 or 1, respectively, then the modulated symbol is the previous modulated symbol multiplied by exp(jθ) or -exp(jθ), respectively.

Dialog Box



DBPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential binary phase shift keying method.

Parameters

Phase rotation (rad): 0

Output data type: double

Phase rotation (rad)

The phase difference between the previous and current modulated symbols when the input is zero.

Output Data type

The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point

Pair Block

DBPSK Demodulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

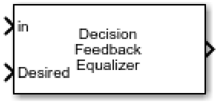
Blocks

BPSK Modulator Baseband | DQPSK Modulator Baseband

Introduced before R2006a

Decision Feedback Equalizer

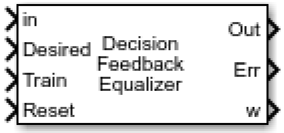
Equalize modulated signals using decision feedback filtering
Library: Communications Toolbox / Equalizers



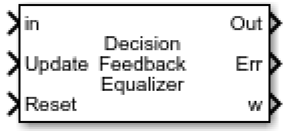
Description

The Decision Feedback Equalizer block uses a decision feedback filter tap delay line with a weighted sum to equalize modulated signals transmitted through a dispersive channel. Using an estimate of the channel modeled as a finite input response (FIR) filter, the block processes input frames and outputs the estimated signal.

This icon shows the block with all ports enabled for configurations that use the LMS or RLS adaptive algorithm.



This icon shows the block with all ports enabled for configurations that use the CMA adaptive algorithm.



Ports

Input

`in` — Input signal

column vector

Input signal, specified as a column vector. The vector length of **in** must be equal to an integer multiple of the **Number of input samples per symbol** parameter. For more information, see “Symbol Tap Spacing” on page 5-456.

Data Types: `double`
Complex Number Support: Yes

`Desired` — Training symbols

column vector

Training symbols, specified as a column vector. The vector length of **Desired** must be less than or equal to the length of input **in**. The **Desired** input port is ignored when the **Train** input port is `0`.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `LMS` or `RLS`.

Data Types: `double`
Complex Number Support: Yes

`Train` — Train equalizer flag
boolean scalar

Train equalizer flag, specified as `1` or `0`. The block starts training when this value changes from `0` to `1` (at the rising edge). The block trains until all symbols in the **Desired** input port are processed.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `LMS` or `RLS` and select the **Enable training control input** parameter.

Data Types: `Boolean`

`Update` — Update tap weights flag
`1` | `0`

Update tap weights flag, specified as `1` or `0`. The tap weights are updated when this value is `1`.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `CMA` and the **Source of adapt weights flag** parameter to `Input port`.

Data Types: `Boolean`

`Reset` — Reset equalizer flag
`1` | `0`

Reset equalizer flag, specified as `1` or `0`. If **Reset** is set to `1`, the block resets the tap weights before processing the incoming signal. The block performs initial training until all symbols in the **Desired** input port are processed.

Dependencies

To enable this port, select the **Enable reset input** parameter.

Data Types: `Boolean`

Output

`Out` — Equalized symbols
column vector

Equalized symbols, returned as a column vector that has the same length as input signal **in**.

This port is unnamed until you select the **Output error signal** or **Output taps weights** parameter.

`Err` — Error signal
column vector

Error signal, returned as a column vector that has the same length as input signal **in**.

`w` — Tap weights

column vector

Tap weights, returned as an N_{Taps}-by-1 vector, where N_{Taps} is equal to the sum of the **Number of forward taps** and **Number of feedback taps** parameter values. **w** contains the tap weights from the last tap weight update.

Parameters

Structure parameters

`Number of forward taps` — Number of forward equalizer taps

`5` (default) | positive integer

Number of forward equalizer taps, specified as a positive integer. The number of forward equalizer taps must be greater than or equal to the value of the **Number of input samples per symbol** parameter.

`Number of feedback taps` — Number of feedback equalizer taps

`3` (default) | positive integer

Number of feedback equalizer taps, specified as a positive integer.

`Signal constellation` — Signal constellation

`pskmod(0:3,4,pi/4)` (default) | vector

Signal constellation, specified as a vector. The default value is a QPSK constellation generated using this code: `pskmod(0:3,4,pi/4)`.

Tunable: Yes

`Number of input samples per symbol` — Number of input samples per symbol

`1` (default) | positive integer

Number of input samples per symbol, specified as a positive integer. Setting this parameter to any number greater than `1` effectively creates a fractionally spaced equalizer. For more information, see “Symbol Tap Spacing” on page 5-456.

Algorithm parameters

`Adaptive algorithm` — Adaptive algorithm

`LMS` (default) | `RLS` | `CMA`

Adaptive algorithm used for equalization, specified as one of these values:

- `LMS` — Update the equalizer tap weights using the “Least Mean Square (LMS) Algorithm” on page 5-457.
- `RLS` — Update the equalizer tap weights using the “Recursive Least Square (RLS) Algorithm” on page 5-457.
- `CMA` — Update the equalizer tap weights using the “Constant Modulus Algorithm (CMA)” on page 5-458.

`Step size` — Step size

`0.01` (default) | positive scalar

Step size used by the adaptive algorithm, specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `CMA`.

`Forgetting factor` — Forgetting factor

`0.99` (default) | scalar in the range (0, 1]

Forgetting factor used by the adaptive algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `RLS`.

`Initial inverse correlation matrix` — Initial inverse correlation matrix

`0.1` (default) | scalar | matrix

Initial inverse correlation matrix, specified as a scalar or an N_{Taps}-by-N_{Taps} matrix. N_{Taps} is equal to the sum of the **Number of forward taps** and **Number of feedback taps** parameter values. If you specify this value as a scalar, *a*, the equalizer sets the initial inverse correlation matrix to *a* times the identity matrix: a(`eye`(N_{Taps})).

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `RLS`.

Control parameters

`Reference tap` — Reference tap

`3` (default) | positive integer

Reference tap, specified as a positive integer less than or equal to the **Number of forward taps** parameter value. The equalizer uses the reference tap location to track the main energy of the channel.

`Input signal delay (samples)` — Input signal delay

`0` (default) | nonnegative integer

Input signal delay in samples relative to the reset time of the equalizer, specified as a nonnegative integer. If the input signal is a vector of length greater than 1, then the input delay is relative to the start of the input vector. If the input signal is a scalar, then the input delay is relative to the first call of the block and to the first call of the block after the **Reset** input port toggles to `1`.

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Source of adapt weights flag` — Source of adapt tap weights request

`Property` (default) | `Input port`

Source of the adapt tap weights request, specified as one of these values:

- `Property` — Specify this value to use the **Adaptive algorithm** parameter to control when the block adapts tap weights.
- `Input port` — Specify this value to use the **Update** input port to control when the block adapts tap weights.

Dependencies

To enable this parameter, set **Adaptive algorithm** to `CMA`.

`Adapt tap weights` — Adapt tap weights

`on` (default) | `off`

Select this parameter to adaptively update the equalizer tap weights. If this parameter is cleared, the block keeps the equalizer tap weights unchanged.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `CMA` and **Source of adapt weights flag** to `Property`.

`Initial tap weights source` — Source for initial tap weights

`Auto` (default) | `Property`

Source for initial tap weights, specified as one of these values:

- `Auto` — Initialize the tap weights to the algorithm-specific default values, as described in the **Initial weights** parameter.
- `Property` — Initialize the tap weights using the **Initial weights** parameter value.

`Initial weights` — Initial tap weights

`0` or `[0;0;1;0;0]` (default) | scalar | column vector

Initial tap weights used by the adaptive algorithm, specified as a scalar or an N_{Taps}-by-1 vector. N_{Taps} is equal to the sum of the **Number of forward taps** and **Number of feedback taps** parameter values. The default is `0` when the **Adaptive algorithm** parameter is set to `LMS` or `RLS`. The default is `[0;0;1;0;0]` when the **Adaptive algorithm** parameter is set to `CMA`.

If you specify **Initial weights** as a vector, the vector length must be N_{Taps}. If you specify **Initial weights** as a scalar, the equalizer uses scalar expansion to create a vector of length N_{Taps} with all values set to **Initial weights**.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial tap weights source** to `Property`.

`Tap weight update period (symbols)` — Tap weight update period

`1` (default) | positive integer

Tap weight update period in symbols, specified as a positive integer. The equalizer updates the tap weights after processing this number of symbols.

`Enable training control input` — Enable training control input

`off` (default) | `on`

Select this parameter to enable input port **Train**. If this parameter is cleared, the block does not reenter training mode after the initial tap training.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Update tap weights when not training` — Update tap weights when not training

`on` (default) | `off`

Select this parameter to use decision directed mode to update equalizer tap weights. If this parameter is cleared, the block keeps the equalizer tap weights unchanged after training.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Enable reset input` — Enable reset input

`off` (default) | `on`

Select this parameter to enable input port **Train**. If this parameter is cleared, the block does not reenter training mode after the initial tap training.

Tunable: Yes

Diagnostic parameters

`Output error signal` — Enable error signal output

`off` (default) | `on`

Select this parameter to enable output port **Err** containing the equalizer error signal.

Tunable: Yes

`Output taps weights` — Enable tap weights output

`off` (default) | `on`

Select this parameter to enable output port **w** containing tap weights from the last tap weight update.

Tunable: Yes

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

Symbol Tap Spacing

You can configure the equalizer to operate as a symbol-spaced equalizer or as a fractional symbol-spaced equalizer.

- To operate the equalizer at a symbol-spaced rate, specify the number of samples per symbol as `1`. Symbol-rate equalizers have taps spaced at the symbol duration. Symbol-rate equalizers are sensitive to timing phase.
- To operate the equalizer at a fractional symbol-spaced rate, specify the number of input samples per symbol as an integer greater than `1` and provide an input signal oversampled at that sampling rate. Fractional symbol-spaced equalizers have taps spaced at an integer fraction of the input symbol duration. Fractional symbol-spaced equalizers are not sensitive to timing phase.

Algorithms

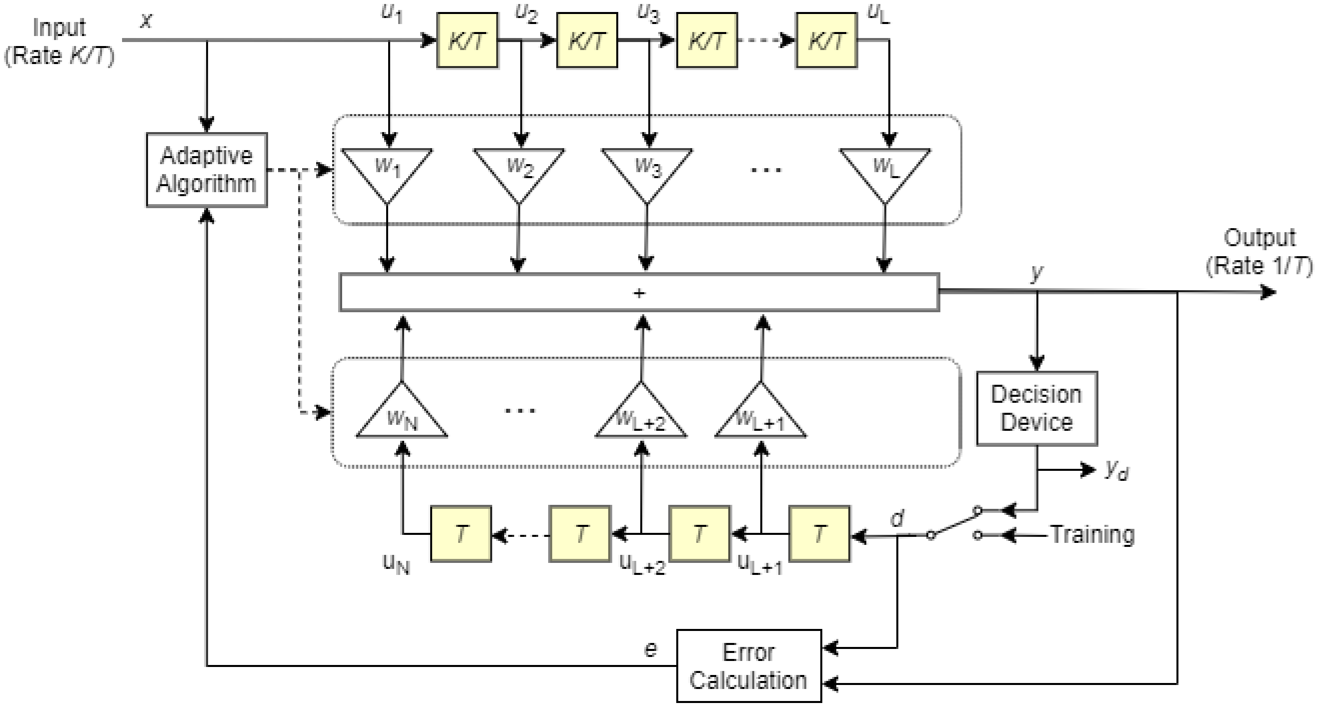
Decision Feedback Equalizers

A decision feedback equalizer (DFE) is a nonlinear equalizer that reduces intersymbol interference (ISI) in frequency-selective channels. If a null exists in the frequency response of a channel, DFEs do not enhance the noise. A DFE consists of a tapped delay line that stores samples from the input signal and contains a forward filter and a feedback filter. The forward filter is similar to a linear equalizer. The feedback filter contains a tapped delay line whose inputs are the decisions made on the equalized signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period.

DFEs can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, K, is 1. The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, K, is an integer greater than 1. Typically, K is 4 for fractional symbol-spaced equalizers. The output sample rate is $1/T$ and the input sample rate is K/T. Tap weight updating occurs at the output rate.

This schematic shows a fractional symbol-spaced DFE with a total of *N* weights, a symbol period of *T*, and *K* samples per symbol. The filter has *L* forward weights and *N*-*L* feedback weights. The forward filter is at the top, and the feedback filter is at the bottom. If *K* is 1, the result is a symbol-spaced DFE instead of a fractional symbol-spaced DFE.



In each symbol period, the equalizer receives *K* input samples at the forward filter and one decision or training sample at the feedback filter. The equalizer then outputs a weighted sum of the values in the forward and feedback delay lines and updates the weights to prepare for the next symbol period.

Note The algorithm for the Adaptive Algorithm block in the schematic jointly optimizes the forward and feedback weights. Joint optimization is especially important for convergence in the recursive least square (RLS) algorithm.

For more information, see “Equalization”.

Least Mean Square (LMS) Algorithm

For the LMS algorithm, in the previous schematic, *w* is a vector of all weights w_i, and *u* is a vector of all inputs u_i. Based on the current set of weights, the LMS algorithm creates the new set of weights as

$$w_{\text{new}} = w_{\text{current}} + (StepSize)\, ue^*.$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed when using the LMS adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Recursive Least Square (RLS) Algorithm

For the RLS algorithm, in the previous schematic, w is the vector of all weights w_i, and u is the vector of all inputs u_i. Based on the current set of inputs, u, and the inverse correlation matrix, P, the RLS algorithm first computes the Kalman gain vector, K, as

$$K = \frac{Pu}{(ForgettingFactor) + u^H Pu}.$$

The forgetting factor used by the adaptive algorithm is specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalized output signal to be less stable. H denotes the Hermitian transpose. Based on the current inverse correlation matrix, the new inverse correlation matrix is

$$P_{\text{new}} = \frac{(1 - Ku^H)P_{\text{current}}}{ForgettingFactor}.$$

Based on the current set of weights, the RLS algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}}+K^*e.$

The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Constant Modulus Algorithm (CMA)

For the CMA adaptive algorithm, in the previous schematic, w is the vector of all weights w_i, and u is the vector of all inputs u_i. Based on the current set of weights, the CMA adaptive algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + (StepSize)\ u^*e.$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed by the CMA adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = y(R - |y|^2)$, where R is a constant related to the signal constellation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Linear Equalizer | MLSE Equalizer

Objects

`comm.DecisionFeedback`

Topics

“Equalization”
“Adaptive Equalizers”

Introduced in R2019a

Deinterlacer

Distribute elements of input vector alternately between two output vectors
Library: Communications Toolbox / Sequence Operations

Deinterlacer
O
E

Description

The Deinterlacer block accepts an even length column vector input signal. The block alternately places the elements in two output vectors. As a result, each output vector size is half the input vector size. The outputs inherit the sample time from the **Sample time** parameter of the input block.

Ports

Input

In — Input signal
column vector

Input signal, specified as an even length column vector.

Data Types: `double` | `single`

Output

0 — Odd-numbered elements
vector

Odd-numbered elements of the input vector, returned as a vector. If the input is a vector of length *N*, the output length is *N*/2.

E — Even-numbered elements
vector

Even-numbered elements of the input vector, returned as a vector. If the input is a vector of length *N*, the output length is *N*/2.

Block Characteristics

Data Types	`Boolean` | `double` | `enumerated` | `fixed point` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Demux | Interlacer

Introduced before R2006a

Derepeat

Reduce sampling rate by averaging consecutive samples
Library: Communications Toolbox / Sequence Operations

Description

The Derepeat block resamples the discrete input at a rate 1/*N* times the input sample rate by averaging *N* consecutive samples. *N* represents the `Derepeat factor, N` parameter.

Ports

Input

`In` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Output signal
scalar | vector | matrix

Output signal, returned as a scalar or column vector.

Data Types: `double`
Complex Number Support: Yes

For more information on the processing rates, see “Single-Rate Processing” on page 5-180, and “Multirate Processing” on page 5-181.

Parameters

`Derepeat factor, N` — Derepeat factor
5 (default) | integer

Derepeat factor, specified as an integer. The derepeat factor is the number of consecutive input samples to average to produce each output sample.

Data Types: `double`

`Input processing` — Input processing control
`Columns as channels (frame based)` (default) | `Elements as channels (sample based)`

Input processing control, specified as one of these options:

- `Columns as channels (frame based)` — The block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — The block treats each element of the input as a separate channel.

`Rate options` — Block processing rate
`Allow multirate processing` (default) | `Enforce single-rate processing`

Block processing rate, specified as one of these options:

- `Allow multirate processing` — The block downsamples the signal such that the output sample rate is `Derepeat factor, N` times slower than the input sample rate. For more information, see "Multirate Processing" on page 5-181.
- `Enforce single-rate processing` — The block maintains the input sample rate by decreasing the output frame size by a factor equal to the `Derepeat factor, N` parameter value. Also, in single-rate processing mode you can use this block in a triggered subsystem. For more information, see "Single-Rate Processing" on page 5-180

`Initial condition` — Initial condition
0 (default) | scalar | vector | matrix

Initial condition, specified as a scalar, vector, or matrix. This parameter specifies values that are output when it is too early for the input data to show up in the output. If the dimensions of the **Initial condition** parameter match the output dimensions, then the parameter represents the initial output value. If **Initial condition** is a scalar, then it represents the initial value of each element in the output. The block does not support empty matrices for initial conditions.

Data Types: `double`

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Single-Rate Processing

The block derepeats each frame, treating distinct channels independently. Each element of the output is the average of *N* consecutive elements along a *column* of the input matrix. *N* must be less than the frame size. *N* represents the `Derepeat factor, N` parameter.

When you set the `Rate options` parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. The block reduces the sampling rate by using a proportionally smaller frame *size* than the input. To process all input values, *N* must be an integer factor of the number of rows in the input vector or matrix. For derepetition by a factor of *N*, the output frame size is 1/*N* times the input frame size, but the input and output frame rates are equal.

When you use this option, the `Initial condition` parameter does not apply and the block incurs no delay, because the input data immediately shows up in the output.

For example, for a single-channel input with 64 elements that is derepeated by a factor of 4, the block outputs 16 elements. The input and output frame periods are equal.

Also, in single-rate processing mode you can use this block in a triggered subsystem.

Multirate Processing

When you set the `Rate options` parameter to `Allow multirate processing`, the input and output of the block are the same size, but the sample rate of the output is *N* times slower than the input. *N* represents the `Derepeat factor, N` parameter.

- When you set the `Input processing` parameter to `Elements as channels (sample based)`, the block assumes that the input is a vector or matrix whose elements represent samples from independent channels. The block averages samples from each channel independently over time. The output period is *N* times the input period, and the input and output sizes are identical. The output is delayed by one output period, and the first output value is the `Initial condition` value. If you set `Rate options` to `Enforce single-rate processing`, the block generates an error message.
- When you set the `Input processing` parameter to `Columns as channels (frame based)`, the block reduces the sampling rate by using a proportionally longer frame *period* at the output port than at the input port. For derepetition by a factor of *N*, the output frame period is *N* times the input frame period, but the input and output frame sizes are equal. The output is delayed by one output frame, and the first output frame is the `Initial condition` value. The block derepeats each frame, treating distinct channels independently. Each element of the output is the average of *N* consecutive elements along a *column* of the input matrix. The derepeat factor must be less than the frame size.

  For example, for a single-channel input with a frame period of 1 second that is derepeated by a factor of 4, the output has a frame period of 4 seconds. The input and output frame sizes are equal.

Pair Block

Repeat — This block is one possible inverse operation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Downsample | Repeat

Introduced before R2006a

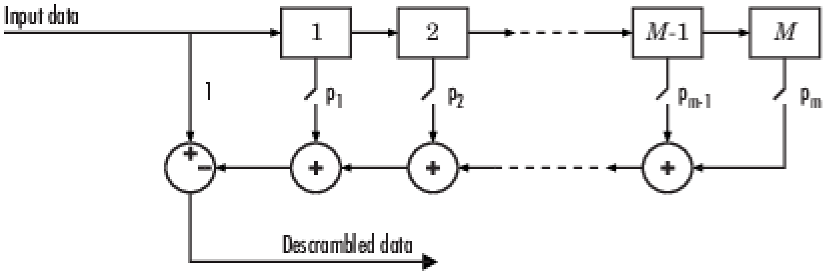
Descrambler

Descramble input signal
Library: Communications Toolbox / Sequence Operations

Description

The Descrambler block descrambles a scalar or column vector input signal. The Descrambler block is the inverse of the Scrambler block. If you use the Scrambler block in a transmitter, then you use the Descrambler block in the related receiver.

This schematic shows the descrambler operation. The adders and subtracter operate modulo *N*, where *N* is the value specified by the Calculation base parameter.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Scramble polynomial parameter, you specify the on or off state for each switch in the descrambler. To make the Descrambler block reverse the operation of the Scrambler block, use the same parameter settings in both blocks. If there is no signal delay between the scrambler and the descrambler, then the **Initial states** in the two blocks must be the same.

To achieve repeatable initial descrambler conditions, you can use one of these optional input ports:

- Select the Reset on nonzero input via port parameter and reset the scrambler with Rst.
- Set the Initial states source parameter to `Input port` and provide the initial states with ISt.

This block can accept input sequences that vary in length during simulation. For more information about sequences that vary in length, see “Variable-Size Signal Basics” (Simulink).

Ports

Input

`in` — Input data signal

vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal. The input values must be integers from `0` to Calculation base – `1`.

Data Types: `double`

`Rst` — Reset scrambler

scalar

Reset scrambler, specified as a scalar. The scrambler is reset if a nonzero input is applied to the port.

Dependencies

To enable this port, set Initial states source to `Dialog Parameter` and select Reset on nonzero input via port.

`ISt` — Initial states

scalar

Initial states of the descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **ISt** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from `0` to Calculation base – `1`.

Dependencies

To enable this port, set Initial states source to `Input port`.

Output

`Out1` — Output descrambled data

vector

Output descrambled data, returned as an N_S-by-1 vector. N_S equals the number of samples in the input signal.

Data Types: `double`

Parameters

`Calculation base` — Calculation base

4 (default) | nonnegative integer

Calculation base used in the descrambler for modulo operations, specified as a nonnegative integer. The input and output of this block are integers from `0` to **Calculation base** – `1`.

`Scramble polynomial` — Polynomial that defines connections in descrambler

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Polynomial that defines the connections in the descrambler, specified as a character vector, integer vector, or binary vector. The **Scramble polynomial** parameter defines if each switch in the descrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + \ldots$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of *z* that appear in the polynomial that has a coefficient of 1. In this case, the order of the descramble polynomial is one less than the binary vector length.

Example: '1 + z^-6 + z^-8', [0 -6 -8], and [1 0 0 0 0 0 1 0 1] all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Initial states source — Set the source for descrambler initial states
Dialog Parameter (default) | Input port

- Dialog Parameter – Specify descrambler initial states by using the Initial states parameter.
- Input port – Specify descrambler initial states by using the ISt port.

Initial states — Initial states of descrambler registers
[0 1 2 3] (default) | nonnegative integer vector

Initial states of descrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **Initial states** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from 0 to Calculation base – 1.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

Reset on nonzero input via port — Reset descrambler via input port
off (default) | on

Select this parameter to reset the Descrambler block via input port Rst.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

Block Characteristics

Data Types	Boolean | double | integer
Multidimensional Signals	no
Variable-Size Signals	no

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

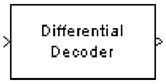
Blocks
PN Sequence Generator | Scrambler

Objects
comm.Descrambler

Introduced before R2006a

Differential Decoder

Decode binary signal using differential coding



Library

Source Coding

Description

The Differential Decoder block decodes the binary input signal. The output is the logical difference between the consecutive input element within a channel. More specifically, the block's input and output are related by

$m(i_0)$ = $d(i_0)$ XOR **Initial condition** parameter value

$m(i_k) = d(i_k)$ XOR $d(i_{k-1})$

where

- d is the differentially encoded input.
- m is the output message.
- i_k is the kth element.
- XOR is the logical exclusive-or operator.

This block accepts a scalar, column vector, or matrix input signal and treats columns as channels.

Parameters

Initial conditions

The logical exclusive-or of this value with the initial input value forms the initial output value.

Supported Data Type

Port	Supported Data Types
In	• double • single • boolean • integer • fixed-point

Port	Supported Data Types
Out	- double - single - boolean - integer - fixed-point

References

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, Sixth edition, Upper Saddle River, N. J., Prentice Hall, 2001.

Pair Block

Differential Encoder

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

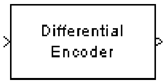
Blocks

Differential Encoder

Introduced before R2006a

Differential Encoder

Encode binary signal using differential coding



Library

Source Coding

Description

The Differential Encoder block encodes the binary input signal within a channel. The output is the logical difference between the current input element and the previous output element. More specifically, the input and output are related by

$d(i_0)$ = $m(i_0)$ XOR **Initial condition** parameter value

$d(i_k) = d(i_{k-1})$ XOR $m(i_k)$

where

- m is the input message.
- d is the differentially encoded output.
- i_k is the kth element.
- XOR is the logical exclusive-or operator.

This block accepts a scalar or column vector input signal and treats columns as channels.

Parameters

Initial conditions

The logical exclusive-or of this value with the initial input value forms the initial output value.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • Integer • Fixed-point

Port	Supported Data Types
Out	• Double-precision floating point • Single-precision floating point • Boolean • Integer • Fixed-Point

References

[1] Couch, Leon W., II, *Digital and Analog Communication Systems*, Sixth edition, Upper Saddle River, N. J., Prentice Hall, 2001.

Pair Block

Differential Decoder

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Differential Decoder

Introduced before R2006a

Discrete-Time VCO

Implement voltage-controlled oscillator in discrete time

Library

Components sublibrary of Synchronization

Description

The Discrete-Time VCO (voltage-controlled oscillator) block generates a signal whose frequency shift from the **Quiescent frequency** parameter is proportional to the input signal. The input signal is interpreted as a voltage. If the input signal is $u(t)$, then the output signal is

$$y(t) = A_c\cos\left(2\pi f_c t + 2\pi k_c\int_0^t u(\tau)d\tau + \varphi\right)$$

where A_c is the **Output amplitude**, f_c is the **Quiescent frequency**, k_c is the **Input sensitivity**, and φ is the **Initial phase**

This block uses a discrete-time integrator to interpret the equation above.

This block accepts a scalar-valued input signal with a data type of `single` or `double`. The output signal inherits its data type from the input signal. The block supports double precision only for code generation.

Parameters

Output amplitude

The amplitude of the output.

Quiescent frequency (Hz)

The frequency of the oscillator output when the input signal is zero.

Input sensitivity

This value scales the input voltage and, consequently, the shift from the **Quiescent frequency** value. The units of **Input sensitivity** are Hertz per volt.

Initial phase (rad)

The initial phase of the oscillator in radians.

Sample time

The calculation sample time.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Continuous-Time VCO

Introduced before R2006a

DPD

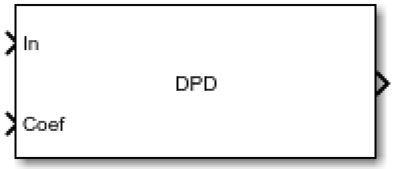
Digital predistorter

Library: Communications Toolbox / RF Impairments Correction

Description

Apply digital predistortion (DPD) to a complex baseband signal using a memory polynomial to compensate for nonlinearities in a power amplifier. For more information, see “Digital Predistortion” on page 5-194.

This icon shows the block with all ports enabled.



Ports

Input

`In` — Input baseband signal
column vector

Input baseband signal, specified as a column vector. This port is unnamed until the **Coefficient source** parameter is set to `Input port`.

Data Types: `double`
Complex Number Support: Yes

`Coef` — Memory-polynomial coefficients
matrix

Memory-polynomial coefficients, specified as a matrix. The number of rows in the matrix must equal the memory depth of the memory polynomial.

- If the **Polynomial type** parameter is set to `Memory polynomial`, the number of columns in the matrix is the degree of the memory polynomial.
- If **Polynomial type** is set to `Cross-term memory polynomial`, the number of columns in the matrix must equal $m(n\text{-}1)+1$. m is the memory depth of the polynomial, and n is the degree of the memory polynomial.

Example: `complex([1 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0])`

Dependencies

To enable this port, set the **Coefficient source** parameter to `Input port`.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Predistorted baseband signal
column vector

Predistorted baseband signal, returned as a column vector of the same length as the input signal.

Parameters

`Polynomial type` — Polynomial type
`Memory polynomial` (default) | `Cross-term memory polynomial`

Polynomial type used for predistortion, specified as one of these values:

- `Memory polynomial` — Computes predistortion coefficients by using a memory polynomial without cross terms
- `Cross-term memory polynomial` — Computes predistortion coefficients by using a memory polynomial with cross terms

For more information, see "Digital Predistortion" on page 5-194.

`Coefficient source` — Source of memory-polynomial coefficients
`Property` (default) | `Input port`

Source of the memory polynomial coefficients, specified as one of these values:

- `Property` — Specify this value to use the **Coefficients** parameter to define the memory-polynomial coefficients
- `Input port` — Specify this value to use the **Coef** input port to define the memory-polynomial coefficients

`Coefficients` — Memory-polynomial coefficients
`complex([1 0 0 0 0; 0 0 0 0 0; 0 0 0 0 0])` (default) | matrix

Memory-polynomial coefficients, specified as a matrix. The number of rows must equal the memory depth of the memory polynomial.

- If the **Polynomial type** is set to `Memory polynomial`, the number of columns is the degree of the memory polynomial.
- If the **Polynomial type** is set to `Cross-term memory polynomial`, the number of columns must equal $m(n\text{-}1)+1$. m is the memory depth of the polynomial, and n is the degree of the memory polynomial.

For more information, see "Digital Predistortion" on page 5-194.

Dependencies

To enable this parameter, set **Coefficient source** to `Property`.

Data Types: `double`
Complex Number Support: Yes

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

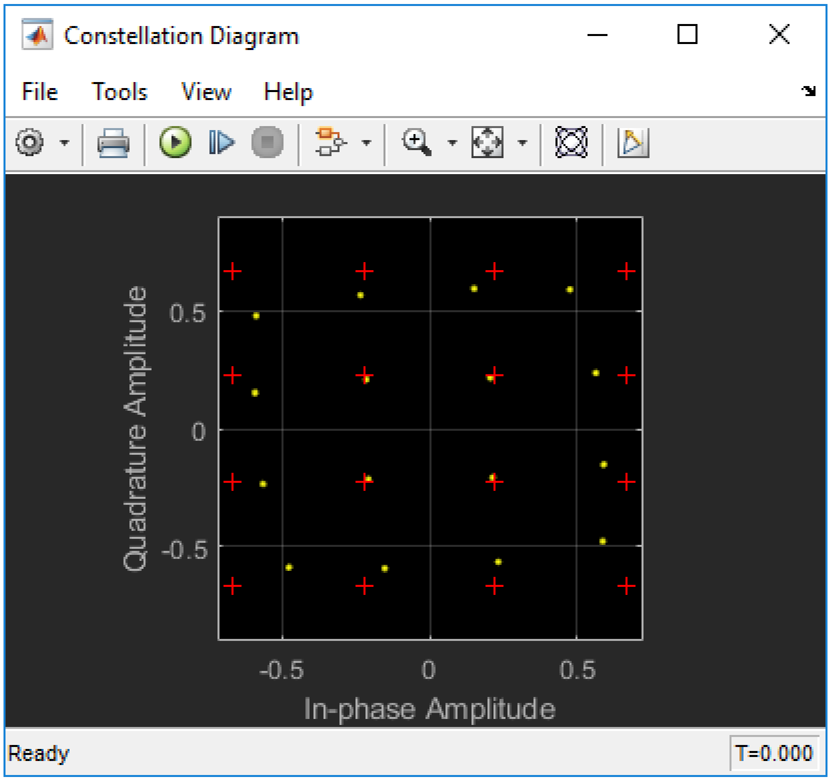
Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

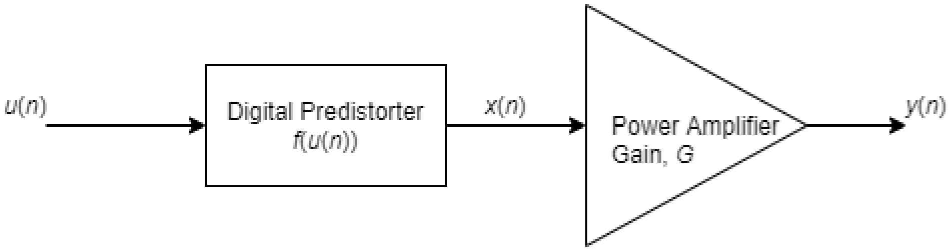
More About

Digital Predistortion

Wireless communication transmissions commonly require wide bandwidth signal transmission over a wide signal dynamic range. To transmit signals over a wide dynamic range and achieve high efficiency, RF power amplifiers (PAs) commonly operate in their nonlinear region. As this constellation diagram shows, the nonlinear behavior of a PA causes signal constellation distortions that pinch the amplitude (AM-AM distortion) and twist phase (AM-PM distortion) of constellation points proportional to the amplitude of the constellation point.



The goal of digital predistortion is to find a nonlinear function that linearizes the net effect of the PA nonlinear behavior at the PA output across the PA operating range. When the PA input is $x(n)$, and the predistortion function is $f(u(n))$, where $u(n)$ is the true signal to be amplified, the PA output is approximately equal to $G\times u(n)$, where G is the desired amplitude gain of the PA.



The digital predistorter can be configured to use a memory polynomial with or without cross terms.

- The memory polynomial with cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1} c_m \times u(n\text{-}m) + \sum_{m=0}^{M-1}\sum_{j=0}^{M-1}\sum_{k=0}^{K-1} a_{mjk} \times u(n\text{-}m) \times \left|u(n\text{-}j)\right|^k .$$

  The memory polynomial with cross terms has (M+M×M×(K-1)) coefficients for c_m and a_{mjk}.
- The memory polynomial without cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1}\sum_{k=0}^{K-1} a_{mk} \times u(n\text{-}m) \times \left|u(n\text{-}m)\right|^k .$$

The polynomial without cross terms has $M \times K$ coefficients for a_{mk}.

Estimating Predistortion Function and Coefficients

The DPD coefficient estimation uses an indirect learning architecture to find function $f(u(n))$ to predistort input signal $u(n)$ which precedes the PA input.

The DPD coefficient estimation algorithm models nonlinear PA memory effects based on the work in reference papers by Morgan, et al [1], and by Schetzen [2], using the theoretical foundation developed for Volterra systems.

Specifically, the inverse mapping from the PA output normalized by the PA gain, $\{y(n)/G\}$, to the PA input, $\{x(n)\}$, provides a good approximation to the function $f(u(n))$, needed to predistort $\{u(n)\}$ to produce $\{x(n)\}$.

Referring to the memory polynomial equations above, estimates are computed for the memory-polynomial coefficients:

- c_m and a_{mjk} for a memory polynomial with cross terms
- a_{mk} for a memory polynomial without cross terms

The memory-polynomial coefficients are estimated by using a least squares fit algorithm or a recursive least squares algorithm. The least squares fit algorithm or a recursive least squares algorithms use the memory polynomial equations above for a memory polynomial with or without cross terms, by replacing $\{u(n)\}$ with $\{y(n)/G\}$. The function order and dimension of the coefficient matrix are defined by the degree and depth of the memory polynomial.

For an example that details the process of accurately estimating memory-polynomial coefficients and predistorting a PA input signal, see “Digital Predistortion to Compensate for Power Amplifier Nonlinearities”.

For background reference material, see the works listed in [1] and [2].

References

[1] Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A Generalized Memory Polynomial Model for Digital Predistortion of Power Amplifiers." *IEEE Transactions on Signal Processing*. Vol. 54, Number 10, October 2006, pp. 3852–3860.

[2] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems.* New York: Wiley, 1980.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
DPD Coefficient Estimator

Objects
`comm.DPD`

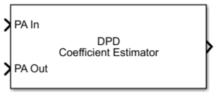
Topics

“Digital Predistortion to Compensate for Power Amplifier Nonlinearities”

Introduced in R2019a

DPD Coefficient Estimator

Estimate memory-polynomial coefficients for digital predistortion
Library: Communications Toolbox / RF Impairments Correction



Description

Estimate memory-polynomial coefficients for digital predistortion (DPD) of a nonlinear power amplifier.

This icon shows the block with all ports enabled.

PA In
PA Out
DPD
Coefficient Estimator
Forgetting Factor

Ports

Input

`PA In` — Power amplifier baseband-equivalent input
column vector

Power amplifier baseband-equivalent input, specified as a column vector.

Data Types: `double`
Complex Number Support: Yes

`PA Out` — Power amplifier baseband-equivalent output
column vector

Power amplifier baseband-equivalent output, specified as a column vector of the same length as **PA In**.

Data Types: `double`
Complex Number Support: Yes

`Forgetting Factor` — Forgetting factor
scalar in the range (0, 1]

Forgetting factor used by the recursive least squares algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the convergence time but causes the output estimates to be less stable.

Dependencies

To enable this port, set **Algorithm** to `Recursive least squares` and set **Forgetting factor source** to `Input port`.

Data Types: `double`

Output

`Out` — Memory-polynomial coefficients
matrix

Memory-polynomial coefficients, returned as a matrix. For more information, see “Digital Predistortion” on page 5-201.

Parameters

`Desired amplitude gain (dB)` — Desired amplitude gain
`10` (default) | scalar

Desired amplitude gain in dB, specified as a scalar. This parameter value expresses the desired signal gain at the compensated amplifier output.

Tunable: Yes

Data Types: `double`

`Polynomial type` — Polynomial type
`Memory polynomial` (default) | `Cross-term memory polynomial`

Polynomial type used for predistortion, specified as one of these values:

- `Memory polynomial` — Computes predistortion coefficients by using a memory polynomial without cross terms
- `Cross-term memory polynomial` — Computes predistortion coefficients by using a memory polynomial with cross terms

For more information, see “Digital Predistortion” on page 5-201.

`Degree` — Memory-polynomial degree
`5` (default) | positive integer

Memory-polynomial degree, specified as a positive integer.

Data Types: `double`

`Memory depth` — Memory-polynomial depth
`3` (default) | positive integer

Memory-polynomial depth in samples, specified as a positive integer.

Data Types: `double`

`Algorithm` — Estimation algorithm
`Least squares` (default) | `Recursive least squares`

Adaptive algorithm used for equalization, specified as one of these values:

- `Least squares` — Estimate the memory-polynomial coefficients by using a least squares algorithm
- `Recursive least squares` — Estimate the memory-polynomial coefficients by using a recursive least squares algorithm

For algorithm reference material, see the works listed in [1] and [2].

Data Types: `char` | `string`

`Forgetting factor source` — Source of forgetting factor

`Property` (default) | `Input port`

Source of the forgetting factor, specified as one of these values:

- `Property` — Specify this value to use the **Forgetting factor** parameter to specify the forgetting factor.
- `Input port` — Specify this value to use the **Forgetting Factor** input port to specify the forgetting factor.

Dependencies

To enable this parameter, set **Algorithm** to `Recursive least squares`.

Data Types: `double`

`Forgetting factor` — Forgetting factor

`0.99` (default) | scalar in the range (0, 1]

Forgetting factor used by the recursive least squares algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the convergence time but causes the output estimates to be less stable.

Dependencies

To enable this parameter, set **Algorithm** to `Recursive least squares` and set **Forgetting factor source** to `Property`.

Data Types: `double`

`Initial coefficient estimate` — Initial coefficient estimate

`[]` (default) | matrix

Initial coefficient estimate for the recursive least squares algorithm, specified as a matrix.

- If you specify this value as an empty matrix, the initial coefficient estimate for the recursive least squares algorithm is chosen automatically to correspond to a memory polynomial that is an identity function, so that the output is equal to input.
- If you specify this value as a nonempty matrix, the number of rows must be equal to the **Memory depth** parameter value.
  - If the **Polynomial type** parameter is set to `Memory polynomial`, the number of columns is the degree of the memory polynomial.
  - If the **Polynomial type** parameter is set to `Cross-term memory polynomial`, the number of columns must equal $m(n\text{-}1)+1$. m is the memory depth of the polynomial, and n is the degree of the memory polynomial.

For more information, see “Digital Predistortion” on page 5-201.

Dependencies

To enable this parameter, set **Algorithm** to `Recursive least squares`.

Data Types: `double`
Complex Number Support: Yes

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

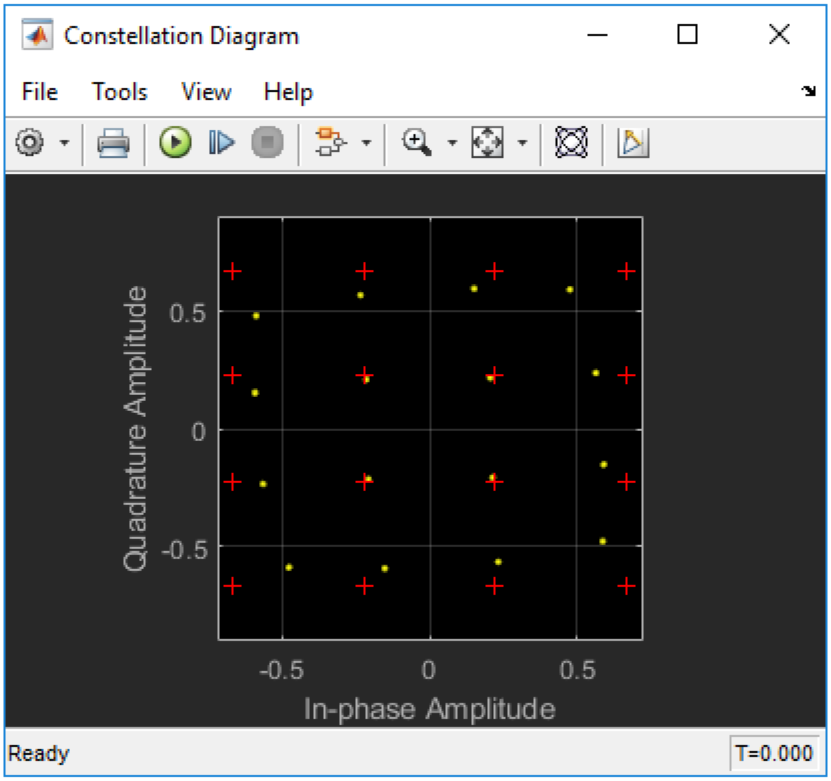
Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

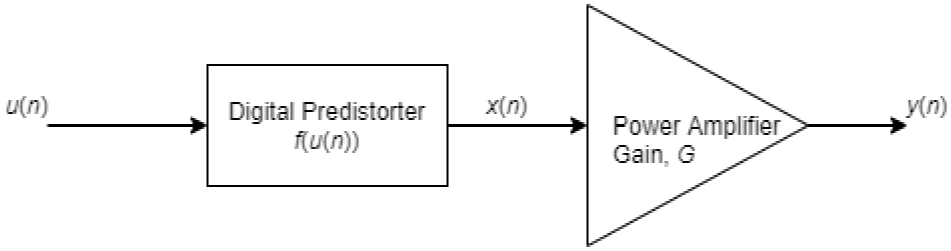
More About

Digital Predistortion

Wireless communication transmissions commonly require wide bandwidth signal transmission over a wide signal dynamic range. To transmit signals over a wide dynamic range and achieve high efficiency, RF power amplifiers (PAs) commonly operate in their nonlinear region. As this constellation diagram shows, the nonlinear behavior of a PA causes signal constellation distortions that pinch the amplitude (AM-AM distortion) and twist phase (AM-PM distortion) of constellation points proportional to the amplitude of the constellation point.



The goal of digital predistortion is to find a nonlinear function that linearizes the net effect of the PA nonlinear behavior at the PA output across the PA operating range. When the PA input is $x(n)$, and the predistortion function is $f(u(n))$, where $u(n)$ is the true signal to be amplified, the PA output is approximately equal to $G\times u(n)$, where G is the desired amplitude gain of the PA.



The digital predistorter can be configured to use a memory polynomial with or without cross terms.

- The memory polynomial with cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1} c_m \times u(n\text{-}m) + \sum_{m=0}^{M-1}\sum_{j=0}^{M-1}\sum_{k=0}^{K-1} a_{mjk} \times u(n\text{-}m) \times \left|u(n\text{-}j)\right|^k .$$

  The memory polynomial with cross terms has $(M+M\times M\times(K\text{-}1))$ coefficients for c_m and a_{mjk}.
- The memory polynomial without cross terms predistorts the input signal as

$$x(n) = f(u(n)) \triangleq \sum_{m=0}^{M-1}\sum_{k=0}^{K-1} a_{mk} \times u(n\text{-}m) \times \left|u(n\text{-}m)\right|^k .$$

The polynomial without cross terms has $M \times K$ coefficients for a_{mk}.

Estimating Predistortion Function and Coefficients

The DPD coefficient estimation uses an indirect learning architecture to find function $f(u(n))$ to predistort input signal $u(n)$ which precedes the PA input.

The DPD coefficient estimation algorithm models nonlinear PA memory effects based on the work in reference papers by Morgan, et al [1], and by Schetzen [2], using the theoretical foundation developed for Volterra systems.

Specifically, the inverse mapping from the PA output normalized by the PA gain, $\{y(n)/G\}$, to the PA input, $\{x(n)\}$, provides a good approximation to the function $f(u(n))$, needed to predistort $\{u(n)\}$ to produce $\{x(n)\}$.

Referring to the memory polynomial equations above, estimates are computed for the memory-polynomial coefficients:

- c_m and a_{mjk} for a memory polynomial with cross terms
- a_{mk} for a memory polynomial without cross terms

The memory-polynomial coefficients are estimated by using a least squares fit algorithm or a recursive least squares algorithm. The least squares fit algorithm or a recursive least squares algorithms use the memory polynomial equations above for a memory polynomial with or without cross terms, by replacing $\{u(n)\}$ with $\{y(n)/G\}$. The function order and dimension of the coefficient matrix are defined by the degree and depth of the memory polynomial.

For an example that details the process of accurately estimating memory-polynomial coefficients and predistorting a PA input signal, see “Digital Predistortion to Compensate for Power Amplifier Nonlinearities”.

For background reference material, see the works listed in [1] and [2].

References

[1] Morgan, Dennis R., Zhengxiang Ma, Jaehyeong Kim, Michael G. Zierdt, and John Pastalan. "A Generalized Memory Polynomial Model for Digital Predistortion of Power Amplifiers." *IEEE Transactions on Signal Processing*. Vol. 54, Number 10, October 2006, pp. 3852–3860.

[2] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems.* New York: Wiley, 1980.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
DPD

Objects
`comm.DPDCoefficientEstimator`

Topics
“Digital Predistortion to Compensate for Power Amplifier Nonlinearities”

Introduced in R2019a

DQPSK Demodulator Baseband

Demodulate DQPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The DQPSK Demodulator Baseband block demodulates a signal that was modulated using the differential quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. The output depends on the phase difference between the current symbol and the previous symbol. The first integer (or binary pair, if you set the **Output type** parameter to `Bit`) at the block output is the initial condition of zero because there is no previous symbol.

This block accepts either a scalar or column vector input signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-206.

Outputs and Constellation Types

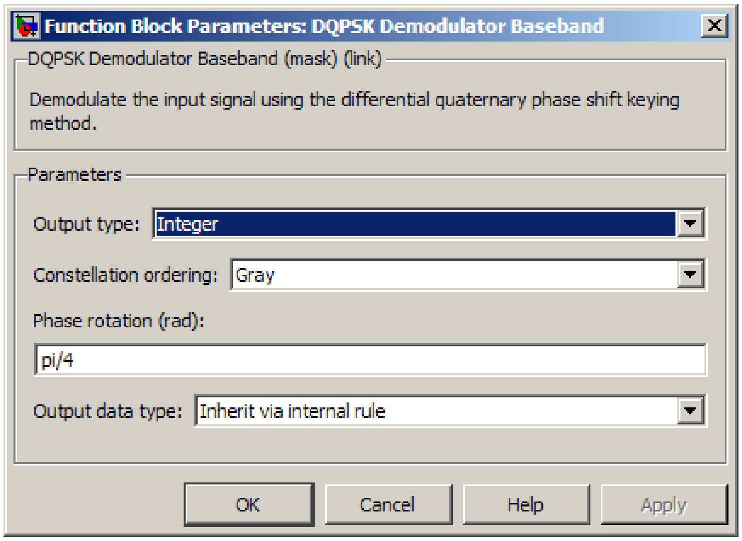
When you set **Output type** parameter to `Integer`, the block maps a phase difference of

$\theta + \pi m/2$

to *m*, where θ represents the **Phase rotation** parameter and *m* is 0, 1, 2, or 3.

When you set the **Output type** parameter to `Bit`, then the output contains pairs of binary values. The reference page for the DQPSK Modulator Baseband block shows which phase differences map to each binary pair, for the cases when the **Constellation ordering** parameter is either `Binary` or `Gray`.

Dialog Box



Function Block Parameters: DQPSK Demodulator Baseband

DQPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential quaternary phase shift keying method.

Parameters

Output type: Integer

Constellation ordering: Gray

Phase rotation (rad):

pi/4

Output data type: Inherit via internal rule

Output type

Determines whether the output consists of integers or pairs of bits.

Constellation ordering

Determines how the block maps each integer to a pair of output bits.

Phase rotation (rad)

This phase difference between the current and previous modulated symbols results in an output of zero.

Output data type

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For integer outputs, this block can output the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, output can be `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is`Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

DQPSK Modulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DBPSK Demodulator Baseband | DQPSK Modulator Baseband | M-DPSK Demodulator Baseband | QPSK Demodulator Baseband

Introduced before R2006a

DQPSK Modulator Baseband

Modulate using differential quadrature phase shift keying method

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The DQPSK Modulator Baseband block modulates using the differential quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

The input must be a discrete-time signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-210.

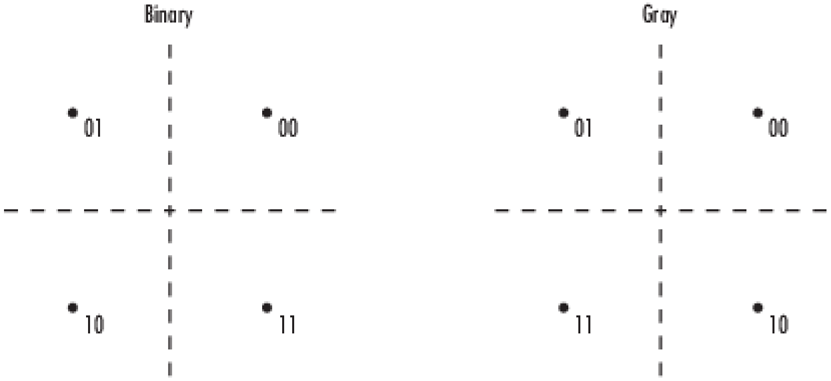
Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the valid input values are 0, 1, 2, and 3. In this case, the block accepts a scalar or column vector input signal. If the first input is *m*, then the modulated symbol is

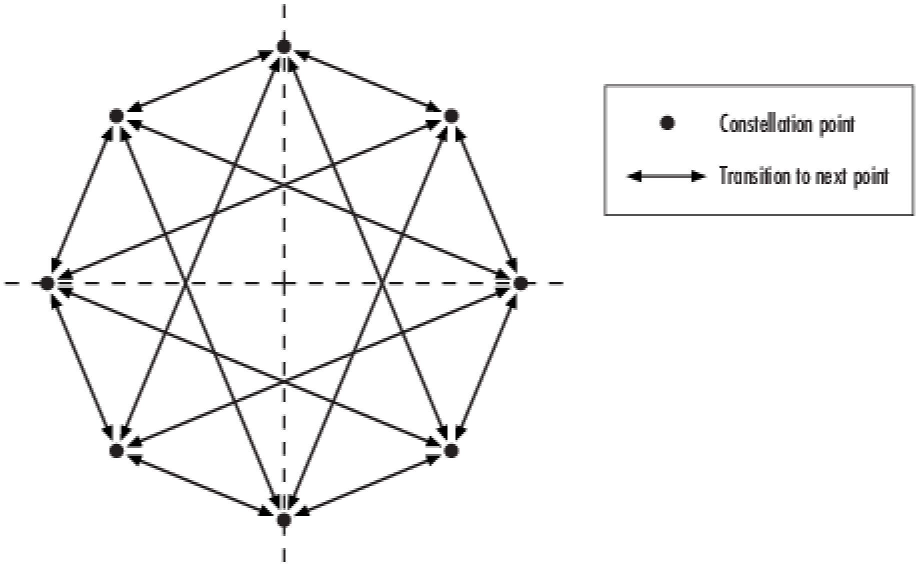
$\exp(j\theta + j\pi m/2)$

where θ represents the **Phase rotation** parameter. If a successive input is *m*, then the modulated symbol is the previous modulated symbol multiplied by $\exp(j\theta + j\pi m/2)$.

When you set the **Input type** parameter to `Bit`, the input contains pairs of binary values. In this case, the block accepts a column vector whose length is an even integer. The following figure shows the complex numbers by which the block multiples the previous symbol to compute the current symbol, depending on whether you set the **Constellation ordering** parameter to `Binary` or `Gray`. The following figure assumes that you set the **Phase rotation** parameter to $\frac{\Pi}{4}$; in other cases, the two schematics would be rotated accordingly.

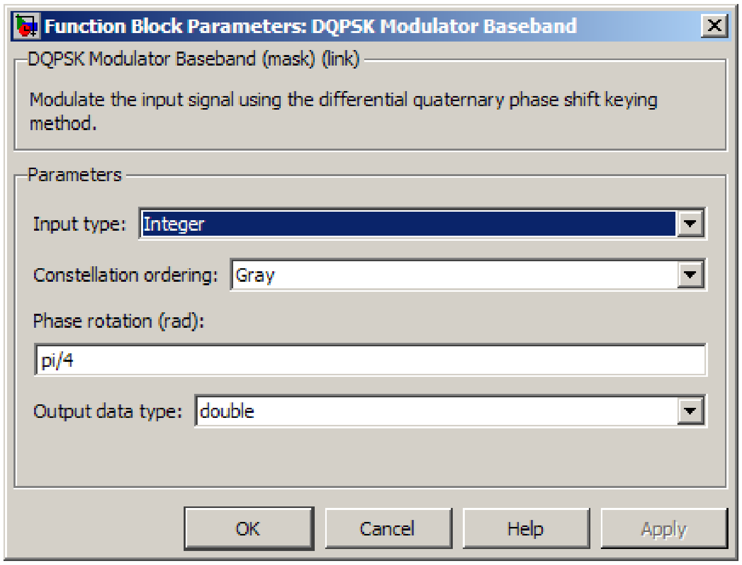


The following figure shows the signal constellation for the DQPSK modulation method when you set the **Phase rotation** parameter to $\frac{\Pi}{4}$. The arrows indicate the four possible transitions from each symbol to the next symbol. The `Binary` and `Gray` options determine which transition is associated with each pair of input values.



More generally, if the **Phase rotation** parameter has the form $\frac{\Pi}{k}$for some integer k, then the signal constellation has $2k$ points.

Dialog Box


Function Block Parameters: DQPSK Modulator Baseband

DQPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential quaternary phase shift keying method.

Parameters

Input type: Integer

Constellation ordering: Gray

Phase rotation (rad):

pi/4

Output data type: double

Input type

Indicates whether the input consists of integers or pairs of bits.

Constellation ordering

Determines how the block maps each pair of input bits to a corresponding integer, using either a Binary or Gray mapping scheme.

Phase rotation (rad)

The phase difference between the previous and current modulated symbols when the input is zero.

Output Data type

The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point

Pair Block

DQPSK Demodulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

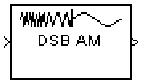
Blocks

DBPSK Modulator Baseband | DQPSK Demodulator Baseband | M-DPSK Modulator Baseband | QPSK Modulator Baseband

Introduced before R2006a

DSB AM Demodulator Passband

Demodulate DSB-AM-modulated data



Library

Analog Passband Modulation, in Modulation

Description

The DSB AM Demodulator Passband block demodulates a signal that was modulated using double-sideband amplitude modulation. The block uses the envelope detection method. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

In the course of demodulating, this block uses a filter whose order, coefficients, passband ripple and stopband ripple are described by their respective lowpass filter parameters.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Input signal offset

The same as the **Input signal offset** parameter in the corresponding DSB AM Modulator Passband block.

Carrier frequency (Hz)

The frequency of the carrier in the corresponding DSB AM Modulator Passband block.

Initial phase (rad)

The initial phase of the carrier in radians.

Lowpass filter design method

The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order

The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency (Hz)

The cutoff frequency of the lowpass digital filter specified in the **Lowpass filter design method** field in Hertz.

Passband ripple (dB)

Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband ripple (dB)

Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

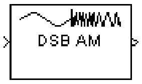
Pair Block

DSB AM Modulator Passband

Introduced before R2006a

DSB AM Modulator Passband

Modulate using double-sideband amplitude modulation



Library

Analog Passband Modulation, in Modulation

Description

The DSB AM Modulator Passband block modulates using double-sideband amplitude modulation. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$(u(t) + k)\cos(2\pi f_c t + \theta)$$

where:

- k is the **Input signal offset** parameter.
- f_c is the **Carrier frequency** parameter.
- θ is the **Initial phase** parameter.

It is common to set the value of k to the maximum absolute value of the negative part of the input signal $u(t)$.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Input signal offset

The offset factor k. This value should be greater than or equal to the absolute value of the minimum of the input signal.

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier.

Pair Block

DSB AM Demodulator Passband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DSB AM Demodulator Passband | DSBSC AM Modulator Passband | SSB AM Modulator Passband

Introduced before R2006a

DSBSC AM Demodulator Passband

Demodulate DSBSC-AM-modulated data

Library

Analog Passband Modulation, in Modulation

Description

The DSBSC AM Demodulator Passband block demodulates a signal that was modulated using double-sideband suppressed-carrier amplitude modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

In the course of demodulating, this block uses a filter whose order, coefficients, passband ripple and stopband ripple are described by the their respective lowpass filter parameters.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The carrier frequency in the corresponding DSBSC AM Modulator Passband block.

Initial phase (rad)

The initial phase of the carrier in radians.

Lowpass filter design method

The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order

The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency (Hz)

The cutoff frequency of the lowpass digital filter specified in the Lowpass filter design method field in Hertz.

Passband Ripple (dB)

Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband Ripple (dB)

Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

Pair Block

DSBSC AM Modulator Passband

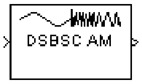
See Also

DSB AM Demodulator Passband, SSB AM Demodulator Passband

Introduced before R2006a

DSBSC AM Modulator Passband

Modulate using double-sideband suppressed-carrier amplitude modulation



Library

Analog Passband Modulation, in Modulation

Description

The DSBSC AM Modulator Passband block modulates using double-sideband suppressed-carrier amplitude modulation. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$u(t)\cos(2\pi f_c t + \theta)$$

where f_c is the **Carrier frequency** parameter and θ is the **Initial phase** parameter.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier in radians.

Pair Block

DSBSC AM Demodulator Passband

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

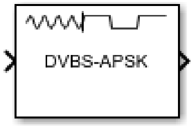
Blocks
DSB AM Modulator Passband | DSBSC AM Demodulator Passband | SSB AM Modulator Passband

Introduced before R2006a

DVBS-APSK Demodulator Baseband

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) demodulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / APM
Communications Toolbox / Modulation / Digital Baseband Modulation / Standard-Compliant



Description

The DVBS-APSK Demodulator Baseband block demodulates the input signal using Digital Video Broadcasting (“DVB-S2/S2X/SH” on page 5-223) standard-specific amplitude phase shift keying (APSK) demodulation. For a description of DVB-compliant APSK demodulation, see “DVB Compliant APSK Hard Demodulation” on page 5-224 and “DVB Compliant APSK Soft Demodulation” on page 5-224.

This icon shows the block with all ports enabled:

In
DVBS-APSK
Var

Ports

Input

In — DVB-S2/S2X/SH standard-specific APSK modulated signal
scalar | vector | matrix

DVB-S2/S2X/SH standard-specific APSK modulated signal, specified as a scalar, vector, or matrix. When this input is a matrix, each column is treated as an independent channel. This port is unnamed until the Var port is enabled.

Data Types: double | single
Complex Number Support: Yes

Var — Noise variance
positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values. When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “DVB Compliant APSK Soft Demodulation” on page 5-224 for demodulation decision type considerations.

Dependencies

This parameter applies when Noise variance source is set to Input port.

Data Types: double | single

Output

Out — Demodulated signal
scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of the demodulated signal depend on the specified Output type and Decision type. This port is unnamed on the block.

Output type	Decision type	Demodulated Signal Description	Dimensions of Demodulated Signal
Integer	—	Demodulated integer values in the range [0, (M – 1)]	The output signal has the same dimensions as the input signal.
Bit	Hard decision	Demodulated bits	The number of rows in the output signal is $\log_2(M)$ times the number of rows in the input signal. Each demodulated symbol is mapped to a group of $\log_2(M)$ elements in a column, where the first element represents the MSB and the last element represents the LSB.
	Log-likelihood ratio	Log-likelihood ratio value for each bit	
	Approximate log-likelihood ratio	Approximate log-likelihood ratio value for each bit	
M is the value of Modulation order.			

Use Output data type to specify the output data type.

Parameters

DVB standard suffix — Standard suffix
S2 | S2X | SH

Standard suffix for DVB modulation variant, specified as S2, S2X, or SH.

Frame length — Frame length
Normal (default) | Short

Frame length, specified as Normal or Short.

Dependencies

This parameter applies only when DVB standard suffix is set to S2 or S2X.

Modulation order — Modulation order
16 (default) | integer

Modulation order, specified as a power of two. The modulation order specifies the total number of points in the constellation of the input signal. The list of valid modulation orders varies depending on the setting for DVB standard suffix and Frame length.

DVB standard suffix	Frame length	Modulation order Options
S2	Normal or Short	16 or 32
S2X	Normal	8, 16, 32, 64, or 256
	Short	16 or 32
SH	Not applicable	16

Code identifier — Code identifier

2/3 | character vector

Code identifier, specified as a character vector. The list of valid code identifier values varies depending on the setting for DVB standard suffix, Frame length, and Modulation order. This table lists the available options for **Code identifier** values.

Modulation order	DVB standard suffix	Frame length	Code identifier Options
8	S2X	Normal	100/180 or 104/180
16	S2 or S2X	Normal	2/3, 3/4, 4/5, 5/6, 8/9, or 9/10
		Short	2/3, 3/4, 4/5, 5/6, or 8/9
16	S2X	Normal	26/45, 3/5, 28/45, 23/36, 25/36, 13/18, 140/180, 154/180, 100/180, 96/180, 90/180, 18/30, or 20/30
		Short	7/15, 8/16, 26/45, 3/5, or 32/45
32	S2 or S2X	Normal	3/4, 4/5, 5/6, 8/9, or 9/10
		Short	3/4, 4/5, 5/6, or 8/9
32	S2X	Normal	128/180, 132/180, 140/180, or 2/3
		Short	2/3 or 32/45
64	S2X	Normal	128/180 , 132/180, 7/9, 4/5, or 5/6
128	S2X	Normal	135/180 or 140/180
256	S2X	Normal	116/180, 124/180, 128/180, 20/30, or 135/180

For more information, refer to Tables 9 and 10 in the DVB-S2 standard [1] and Table 17a in the DVB-S2X standard [2].

Dependencies

This parameter applies only when **DVB standard suffix** is set to S2 or S2X.

Constellation scaling — Constellation scaling

Outer radius as 1 (default) | Unit average power

Constellation scaling, specified as Outer radius as 1 or Unit average power.

Dependencies

This input argument applies only when DVB standard suffix is set to S2 or S2X.

Output type — Output type

Integer (default) | Bit

Output type, specified as Integer or Bit.

Data Types: char | string

Decision type — Demodulation decision type

Hard decision (default) | Log-likelihood ratio | Approximate log-likelihood ratio

Demodulation decision type, specified as Hard decision, Log-likelihood ratio, or Approximate log-likelihood ratio. See "DVB Compliant APSK Soft Demodulation" on page 5-224 for algorithm selection considerations.

Dependencies

This parameter applies when Output type is set to Bit.

Noise variance source — Noise variance source

Property (default) | Input port

Noise variance source, specified as:

- Property — The noise variance is set using the Noise variance parameter.
- Input port — The noise variance is set using the Var input port.

Noise variance — Noise variance

1 (default) | positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values.

- When specified as a scalar, that value is used on all elements in the input signal.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal. Each noise variance vector element is applied to its corresponding column in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see "DVB Compliant APSK Soft Demodulation" on page 5-224 for demodulation decision type considerations.

Dependencies

This parameter applies when Noise variance source is set to Property and Decision type is set to either Log-likelihood ratio or Approximate log-likelihood ratio.

Data Types: double

Output data type — Output data type

double (default) | ...

Output data type, specified as one of the acceptable values from this table. Acceptable **Output data type** values depend on the Output type and Decision type parameter values.

Output type	Decision type	Output data type Options
Integer	Not applicable	double, single, int8, uint8, int16, uint16, int32, or uint32
Bit	Hard decision	double, single, int8, uint8, int16, uint16, int32, uint32, or logical
	Log-likelihood ratio or Approximate log-likelihood ratio	The output signal is the same data type as the input signal.

Dependencies

This parameter applies only when Output type is set to Integer or when Output type is set to Bit and Decision type is set to Hard decision.

Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run, specified as:

- Code generation –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- Interpreted execution –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In Interpreted execution mode, you can debug the source code of the block.

For information on execution speed, see “Tips” on page 5-225.

Block Characteristics

Data Types	Boolean | double | integer | single
Multidimensional Signals	yes
Variable-Size Signals	no

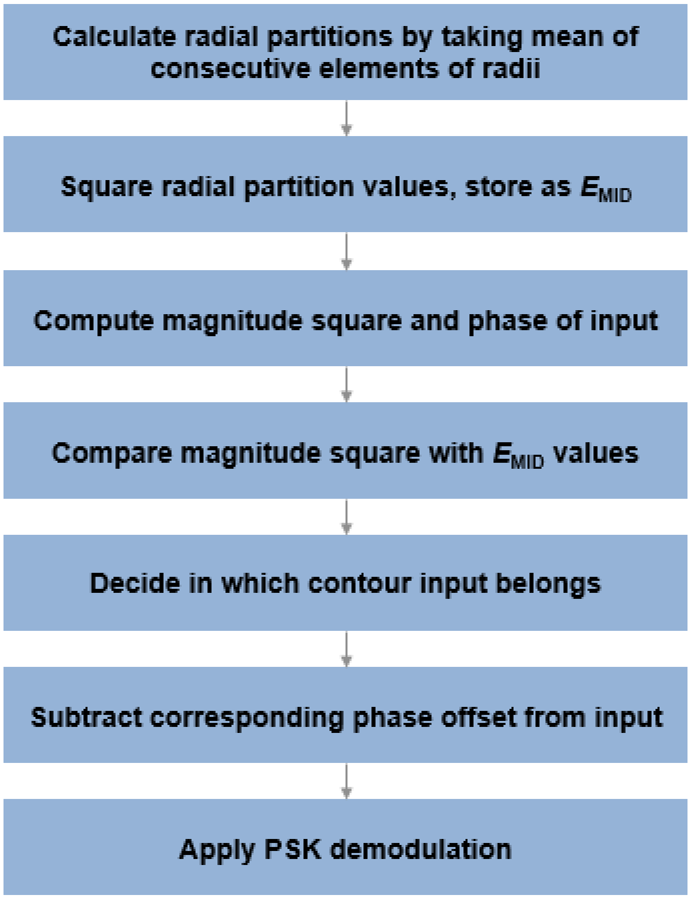
More About

DVB-S2/S2X/SH

Digital video broadcasting (DVB) standards specify S2, S2X, and SH standard-specific amplitude phase shift keying (APSK) modulation. For further information on the DVB-S2/S2X/SH standards, see [1], [2], and [3], respectively.

DVB Compliant APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding as described in [4].


Calculate radial partitions by taking mean of consecutive elements of radii
Square radial partition values, store as E_{MID}
Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide in which contour input belongs
Subtract corresponding phase offset from input
Apply PSK demodulation

DVB Compliant APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value

- NaN if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

Tips

- For faster execution of the DVBS-APSK Demodulator Baseband block, set the Simulate using parameter to:
  - `Code generation` when using hard decision demodulation.
  - `Interpreted execution` when using soft decision demodulation.

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

[4] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DVBS-APSK Modulator Baseband | M-APSK Demodulator Baseband | MIL188-QAM Demodulator Baseband

Functions

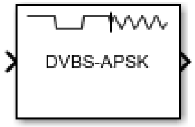
`dvbsapskdemod`

Introduced in R2018b

DVBS-APSK Modulator Baseband

DVB-S2/S2X/SH standard-specific amplitude phase shift keying (APSK) modulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / APM
Communications Toolbox / Modulation / Digital Baseband Modulation / Standard-Compliant



Description

The DVBS-APSK Modulator Baseband block modulates the input signal using Digital Video Broadcasting ("DVB-S2/S2X/SH" on page 5-229) standard-specific amplitude phase shift keying (APSK) modulation.

Ports

Input

In — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The input signal must be binary values or integers in the range [0, (M – 1)], where M is the Modulation order. This port is unnamed on the block.

Note To process the input signal as binary elements, set the Input type parameter value to `Bit`. For binary inputs, the number of rows must be an integer multiple of $\log_2(M)$. Groups of $\log_2(M)$ bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Out` — DVB-S2/S2X/SH standard-specific APSK modulated signal
scalar | vector | matrix

DVB-S2/S2X/SH standard-specific APSK modulated signal, returned as a complex scalar, vector, or matrix. The output signal dimensions depend on the specified Input type value. This port is unnamed on the block.

Input type	Dimensions of Output Signal
`Integer`	The output signal has the same dimensions as the input signal.

Input type	Dimensions of Output Signal
Bit	The number of rows in the output signal equals the number of rows in the input signal divided by $\log_2(M)$, where M is the Modulation order.

Use Output data type to specify the output data type.

Parameters

DVB standard suffix — Standard suffix

S2 (default) | S2X | SH

Standard suffix for DVB modulation variant, specified as S2, S2X, or SH.

Frame length — Frame length

Normal (default) | Short

Frame length, specified as Normal or Short.

Dependencies

This parameter applies only when DVB standard suffix is set to S2 or S2X.

Modulation order — Modulation order

16 (default) | 8 | 32 | 64 | 256

Modulation order, M, specified as a power of two. The modulation order specifies the total number of points in the constellation of the output signal. The list of valid modulation orders varies depending on the values of DVB standard suffix and Frame length.

DVB standard suffix	Frame length	Modulation order Options
S2	Normal or Short	16 or 32
S2X	Normal	8, 16, 32, 64, or 256
	Short	16 or 32
SH	Not applicable	16

Code identifier — Code identifier

2/3 | character vector

Code identifier, specified as a character vector. The list of valid code identifier values varies depending on the specified values of DVB standard suffix, Frame length, and Modulation order. This table lists the options for **Code identifier** values.

Modulation order	DVB standard suffix	Frame length	Code identifier Options
8	S2X	Normal	100/180 or 104/180
16	S2 or S2X	Normal	2/3, 3/4, 4/5, 5/6, 8/9, or 9/10
		Short	2/3, 3/4, 4/5, 5/6, or 8/9

Modulation order	DVB standard suffix	Frame length	Code identifier Options
16	S2X	Normal	26/45, 3/5, 28/45, 23/36, 25/36, 13/18, 140/180, 154/180, 100/180, 96/180, 90/180, 18/30, or 20/30
		Short	7/15, 8/16, 26/45, 3/5, or 32/45
32	S2 or S2X	Normal	3/4, 4/5, 5/6, 8/9, or 9/10
		Short	3/4, 4/5, 5/6, or 8/9
32	S2X	Normal	128/180, 132/180, 140/180, or 2/3
		Short	2/3 or 32/45
64	S2X	Normal	128/180 , 132/180, 7/9, 4/5, or 5/6
128	S2X	Normal	135/180 or 140/180
256	S2X	Normal	116/180, 124/180, 128/180, 20/30, or 135/180

For more information, refer to Tables 9 and 10 in the DVB-S2 standard [1] and Table 17a in the DVB-S2X standard [2].

Dependencies

This parameter applies only when **DVB standard suffix** is set to S2 or S2X.

Constellation scaling — Constellation scaling

Outer radius as 1 (default) | Unit average power

Constellation scaling, specified as Outer radius as 1 or Unit average power.

Dependencies

This parameter applies only when DVB standard suffix is set to S2 or S2X.

Input type — Input type

Integer (default) | Bit

Input type, specified as Integer or Bit. To use Integer, the input signal must consist of integers in the range [0, (M – 1)]. To use Bit, the input data must contain binary values, and the number of rows must be an integer multiple of $\log_2(M)$, where M is the Modulation order.

Output data type — Output data type

double (default) | single

Output data type, specified as double or single.

`View Constellation` — Plot reference constellation

`button`

To plot the reference constellation, click the **View Constellation** button.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as:

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`

More About

DVB-S2/S2X/SH

Digital video broadcasting (DVB) standards specify S2, S2X, and SH standard-specific amplitude phase shift keying (APSK) modulation. For further information on the DVB-S2/S2X/SH standards, see [1], [2], and [3], respectively.

References

[1] ETSI Standard EN 302 307 V1.4.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2), European Telecommunications Standards Institute, Valbonne, France, 2005-03.

[2] ETSI Standard EN 302 307-2 V1.1.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2X), European Telecommunications Standards Institute, Valbonne, France, 2015-02.

[3] ETSI Standard EN 302 583 V1.1.1: Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for Satellite Services to Handheld devices (SH), European Telecommunications Standards Institute, Valbonne, France, 2008-03.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DVBS-APSK Demodulator Baseband | M-APSK Modulator Baseband | MIL188-QAM Modulator Baseband

Functions

`dvbsapskmod`

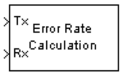
Topics

“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018b

Error Rate Calculation

Compute bit error rate or symbol error rate of input data



Library

Comm Sinks

Description

The Error Rate Calculation block compares input data from a transmitter with input data from a receiver. It calculates the error rate as a running statistic, by dividing the total number of unequal pairs of data elements by the total number of input data elements from one source.

Use this block to compute either symbol or bit error rate, because it does not consider the magnitude of the difference between input data elements. If the inputs are bits, then the block computes the bit error rate. If the inputs are symbols, then it computes the symbol error rate.

Note When you set the **Output data** parameter to `Workspace`, the block generates no code. Similarly, no data is saved to the workspace if the **Simulation mode** is set to `Accelerator` or `Rapid Accelerator`. If you need error rate information in these cases, set **Output data** to `Port`.

Input Data

This block has between two and four input ports, depending on how you set the dialog parameters. The input ports marked `Tx` and `Rx` accept transmitted and received signals, respectively. The `Tx` and `Rx` signals must share the same sampling rate.

The `Tx` and `Rx` input ports accept scalar or column vector signals. For information about the data types each block port supports, see the “Supported Data Types” on page 5-235 table on this page.

If `Tx` is a scalar and `Rx` is a vector, or vice-versa, then the block compares the scalar with each element of the vector. In this case, the block behaves as if you had preprocessed the scalar signal by using the Repeat block with the **Rate options** parameter set to `Enforce single rate`.

If you select **Reset port**, then an additional input port appears, labeled `Rst`. The `Rst` input accepts only a scalar signal (of type `double` or `boolean`) and must have the same port sample time as the `Tx` and `Rx` ports. When the `Rst` input is nonzero, the block clears and then recomputes the error statistics.

If you set the **Computation mode** parameter to `Select samples from port`, then an additional input port appears, labeled `Sel`. The `Sel` input indicates which elements of a frame are relevant for the computation. The `Sel` input can be a column vector of type `double`.

The guidelines below indicate how you should configure the inputs and the dialog parameters depending on how you want this block to interpret your `Tx` and `Rx` data.

- If both data signals are scalar, then this block compares the Tx scalar signal with the Rx scalar signal. For this configuration, use the **Computation mode** parameter default value, `Entire frame`.
- If both data signals are vectors, then this block compares some or all of the Tx and Rx data:
  - If you set the **Computation mode** parameter to `Entire frame`, then the block compares all of the Tx frame with all of the Rx frame.
  - If you set the **Computation mode** parameter to `Select samples from mask`, then the **Selected samples from frame** field appears in the dialog. This parameter field accepts a vector that lists the indices of those elements of the Rx frame that you want the block to consider. For example, to consider only the first and last elements of a length-six receiver frame, set the **Selected samples from frame** parameter to `[1 6]`. If the **Selected samples from frame** vector includes zeros, then the block ignores them.
  - If you set the **Computation mode** parameter to `Select samples from port`, then an additional input port, labeled `Sel`, appears on the block icon. The data at this input port must have the same format as that of the **Selected samples from frame** parameter described above.
- If one data signal is a scalar and the other is a vector, then the scalar is with each entry of the vector. In this case, if Rx is a scalar, then the phrase “Rx frame” above refers to the vector expansion of Rx.

Note This block does not support variable-size signals. If you choose the `Select samples from port` option and want the number of elements in the subframe to vary during the simulation, then you should pad the `Sel` signal with zeros. The Error Rate Calculation block ignores zeros in the `Sel` signal.

Output Data

This block produces a vector of length three, whose entries correspond to:

- The error rate
- The total number of errors, that is, the number of instances that an Rx element does not match the corresponding Tx element
- The total number of comparisons that the block made

The block sends this output data to the base MATLAB workspace or to an output port, depending on how you set the **Output data** parameter:

- If you set the **Output data** parameter to `Workspace` and fill in the **Variable name** parameter, then that variable in the base MATLAB workspace contains the current value when the simulation *ends*. Pausing the simulation does not cause the block to write interim data to the variable.

  If you plan to use this block along with the Simulink Coder software, then you should not use the `Workspace` option. Instead, use the `Port` option and connect the output port to a Simulink To Workspace block.
- If you set the **Output data** parameter to `Port`, then an output port appears. This output port contains the *running* error statistics.

Delays

The **Receive delay** and **Computation delay** parameters implement two different types of delays for this block. One delay is useful if you want this block to compensate for the delay in the received signal. The other is useful if you want to ignore the initial transient behavior of both input signals.

- The **Receive delay** parameter represents the number of samples by which the received data lags behind the transmitted data. The transmit signal is implicitly delayed by that same amount before the block compares it to the received data. This value is helpful when you delay the transmit signal so that it aligns with the received signal. The receive delay persists throughout the simulation.
- The **Computation delay** parameter represents the number of samples the block ignores at the beginning of the comparison.

Use the Find Delay block to determine the delay, and then set the **Receive delay** to the delay reported by the Find Delay block.

If you use the `Select samples from mask` or `Select samples from port` option, then each delay parameter refers to the number of samples that the block receives, whether the block ultimately ignores some of them or not.

If using the **Sel** port to calculate errors on a delayed signal, the delay must be added to the **Sel** indices. For more information, see “Calculate Errors for Delayed Selected Samples”.

Stopping the Simulation Based on Error Statistics

You can configure this block so that its error statistics control the duration of simulation. This is useful for computing reliable steady-state error statistics without knowing in advance how long transient effects might last. To use this mode, check **Stop simulation**. The block attempts to run the simulation until it detects the number of errors the **Target number of errors** parameter specifies. However, the simulation stops before detecting enough errors if the time reaches the model's **Stop time** setting (in the **Configuration Parameters** dialog box), if the Error Rate Calculation block makes **Maximum number of symbols** comparisons, or if another block in the model directs the simulation to stop.

To ignore either of the two stopping criteria in this block, set the corresponding parameter (**Target number of errors** or **Maximum number of symbols**) to `Inf`. For example, to reach a target number of errors without stopping the simulation early, set **Maximum number of symbols** to `Inf` and set the model's **Stop time** to `Inf`.

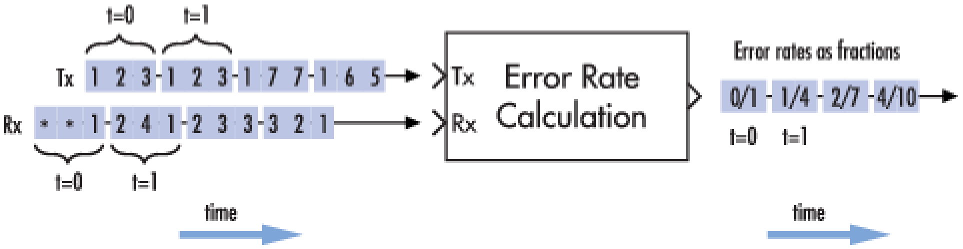
Tuning Parameters in an RSim Executable (Simulink Coder Software)

If you use the Simulink Coder rapid simulation (RSim) target to build an RSim executable, then you can tune the **Target number of errors** and **Maximum number of symbols** parameters without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

Examples

Full Frame Error Calculation

The figure below shows how the block compares pairs of elements and counts the number of error events. The `Tx` and `Rx` inputs are column vectors.



This example assumes that the sample time of each input signal is 1 second and that the block's parameters are as follows:

- **Receive delay** = 2
- **Computation delay** = `0`
- **Computation mode** = `Entire frame`

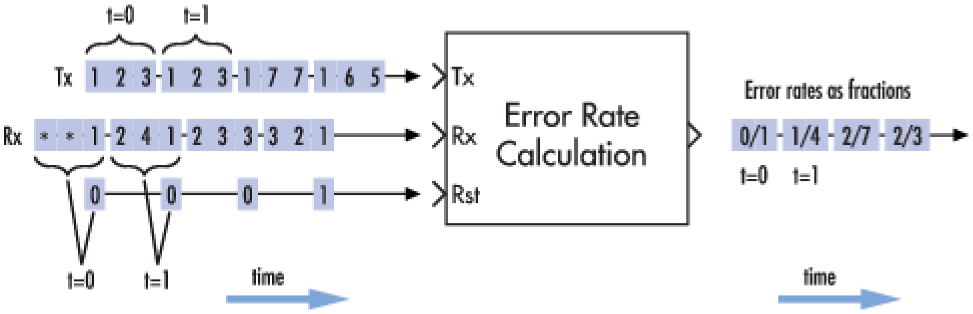
Both input signals are column vectors of length three. However, the schematic arranges each column vector horizontally and aligns pairs of vectors so as to reflect a receive delay of two samples. At each time step, the block compares elements of the Rx signal with those of the Tx signal that appear directly above them in the schematic. For instance, at time 1, the block compares 2, 4, and 1 from the Rx signal with 2, 3, and 1 from the Tx signal.

The values of the first two elements of Rx appear as asterisks because they do not influence the output. Similarly, the 6 and 5 in the Tx signal do not influence the output up to time 3, though they *would* influence the output at time 4.

In the error rates on the right side of the figure, each numerator at time *t* reflects the number of errors when considering the elements of Rx up through time *t*.

Full Frame Error Calculation with Reset

If the block's **Reset port** box had been checked and a reset had occurred at time = 3 seconds, then the last error rate would have been 2/3 instead of 4/10. This value 2/3 would reflect the comparison of 3, 2, and 1 from the Rx signal with 7, 7, and 1 from the Tx signal. The figure below illustrates this scenario. The Tx and Rx inputs are column vectors.



Error Calculation on Selected Samples in the Frame

If using the **Sel** port to calculate errors on a delayed signal, the delay must be added to the **Sel** indices. For more information, see “Calculate Errors for Delayed Selected Samples”.

Parameters

Receive delay

Number of samples by which the received data lags behind the transmitted data. (If Tx or Rx is a vector, then each entry represents a sample.)

Computation delay

Number of samples that the block should ignore at the beginning of the comparison.

Computation mode

Either `Entire frame`, `Select samples from mask`, or `Select samples from port`, depending on whether the block should consider all or only part of the input frames.

Selected samples from frame

A vector that lists the indices of the elements of the Rx frame vector that the block should consider when making comparisons. This field appears only if **Computation mode** is set to `Select samples from mask`.

Output data

Either `Workspace` or `Port`, depending on where you want to send the output data.

Variable name

Name of variable for the output data vector in the base MATLAB workspace. This field appears only if **Output data** is set to `Workspace`.

Reset port

If you check this box, then an additional input port appears, labeled `Rst`.

Stop simulation

If you check this box, then the simulation runs only until this block detects a specified number of errors or performs a specified number of comparisons, whichever comes first.

Target number of errors

The simulation stops after detecting this number of errors. This field is active only if **Stop simulation** is checked.

Maximum number of symbols

The simulation stops after making this number of comparisons. This field is active only if **Stop simulation** is checked.

Supported Data Types

Port	Supported Data Types
Tx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Rx	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Sel	• Double-precision floating point
Reset	• Double-precision floating point • Boolean

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

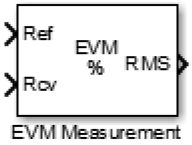
See Also

Blocks
Delay | Find Delay

Introduced before R2006a

EVM Measurement

Measure error vector magnitude



Library

Utility Blocks

Description

The EVM Measurement block measures the error vector magnitude (EVM), which is an indication of modulator or demodulator performance.

The block has one or two input signals: a received signal and, optionally, a reference signal. You must select if the block uses a reference from an input port or from a reference constellation.

The block normalizes to the average reference signal power, average constellation power, or peak constellation power. For RMS EVM, maximum EVM, and *X*-percentile EVM, the output computations reflect the normalization method.

The default EVM output is the RMS EVM in percent, with an option of maximum EVM or *X*-percentile EVM values. The maximum EVM represents the worst-case EVM value per burst. For the *X*-percentile option, you can enable an output port that returns the number of symbols processed in the percentile computations.

The table shows the output type, the parameter that selects the output type, the computation units, and the corresponding measurement interval.

Output	Activation Parameter	Units	Measurement Interval
RMS EVM	None (output by default)	Percentage	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Maximum EVM	**Output maximum EVM**	Percentage	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Percentile EVM	**Output X-percentile EVM**	Percentage	`Entire history`

Output	Activation Parameter	Units	Measurement Interval
Number of symbols	**Output X-percentile EVM** and **Output the number of symbols processed**	None	`Entire history`

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always `double`.

Parameters

Normalize RMS error vector by

Selects the method by which the block normalizes measurements:

- `Average reference signal power`
- `Average constellation power`
- `Peak constellation power`

The default is `Average reference signal power`.

Average constellation power

Normalizes EVM measurement by the average constellation power. This parameter is available only when you set **Normalize RMS error vector** to `Average constellation power`.

Peak constellation power

Normalizes EVM measurement by the peak constellation power. This parameter only is available if you set **Normalize RMS error vector** to `Peak constellation power`.

Reference signal

Specifies the reference signal source as either `Input port` or `Estimated from reference constellation`.

Reference constellation

Specifies the reference constellation points as a vector. This parameter is available only when **Reference signal** is `Estimated from reference constellation`. The default is `constellation(comm.QPSKModulator)`.

Measurement interval

Specify the measurement interval as: `Input length`, `Entire history`, `Custom`, or `Custom with periodic reset`. This parameter affects the RMS and maximum EVM outputs only.

- To calculate EVM using only the current samples, set this parameter to `'Input length'`.
- To calculate EVM for all samples, set this parameter to `'Entire history'`.
- To calculate EVM over an interval you specify and to use a sliding window, set this parameter to `'Custom'`.
- To calculate EVM over an interval you specify and to reset the object each time the measurement interval is filled, set this parameter to `'Custom with periodic reset'`.

Custom measurement interval

Specify the custom measurement interval in samples as a real positive integer. This is the interval over which the EVM is calculated. This parameter is available when **Measurement interval** is `Custom` or `Custom with periodic reset`. The default is `100`.

Averaging dimensions

Averaging dimensions over which to average the EVM measurements, specified as an integer or row vector of integers with element values in the range [1, 3]. For example, to average across the rows, set this parameter to `2`. The default is `1`.

This block supports var-size inputs of the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must be constant. For example, if the input size is `[1000 3 2]` and **Averaging dimensions** is `[1 3]`, then the output size is `[1 3 1]`. The number of elements in the second dimension is fixed at 3.

Output maximum EVM

Outputs the maximum EVM of an input vector or frame.

Output X-percentile EVM

Enables an output *X*-percentile EVM measurement. When you select this option, specify **X-percentile value (%)**.

X-percentile value (%)

This parameter is available only when you select **Output X-percentile EVM**. The *X*th percentile is the EVM value below which X% of all the computed EVM values lie. The parameter defaults to the 95th percentile. That is, 95% of all EVM values are below this value.

Output the number of symbols processed

Outputs the number of symbols that the block uses to compute the *X*-percentile value. This parameter is available only when you select **Output X-percentile EVM**.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

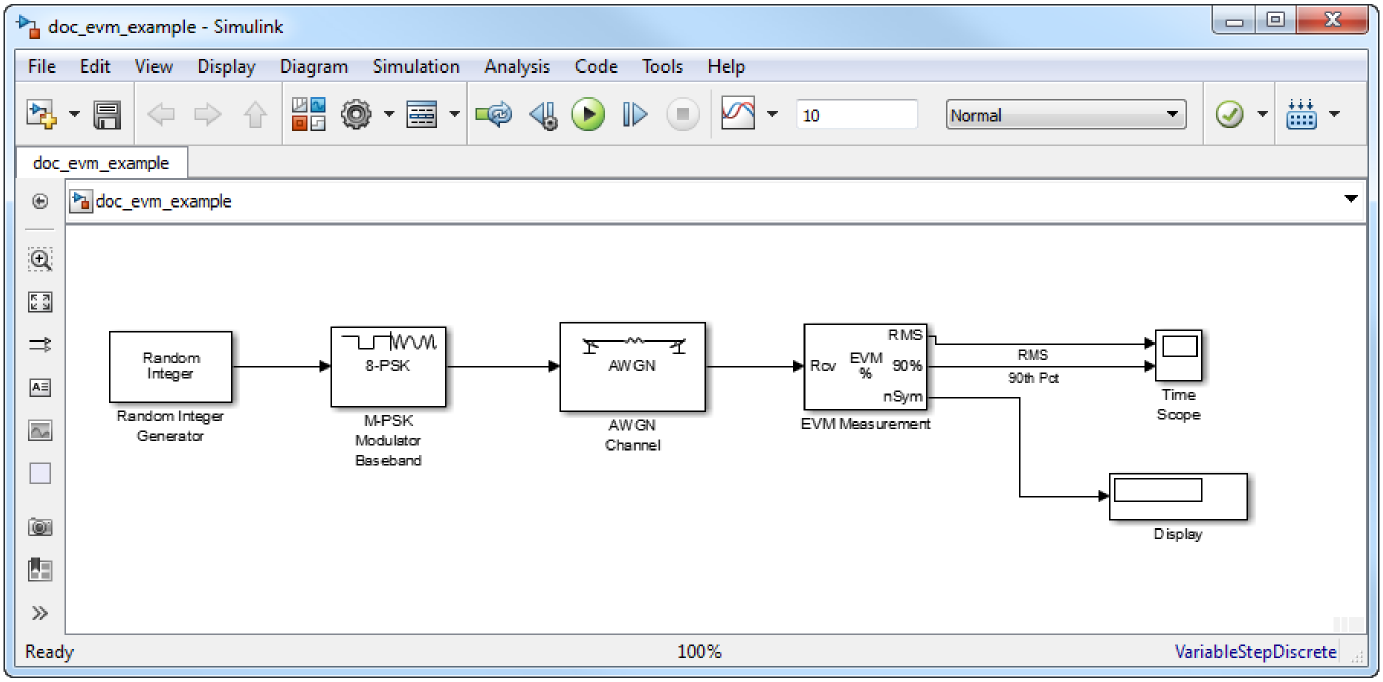
Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Examples

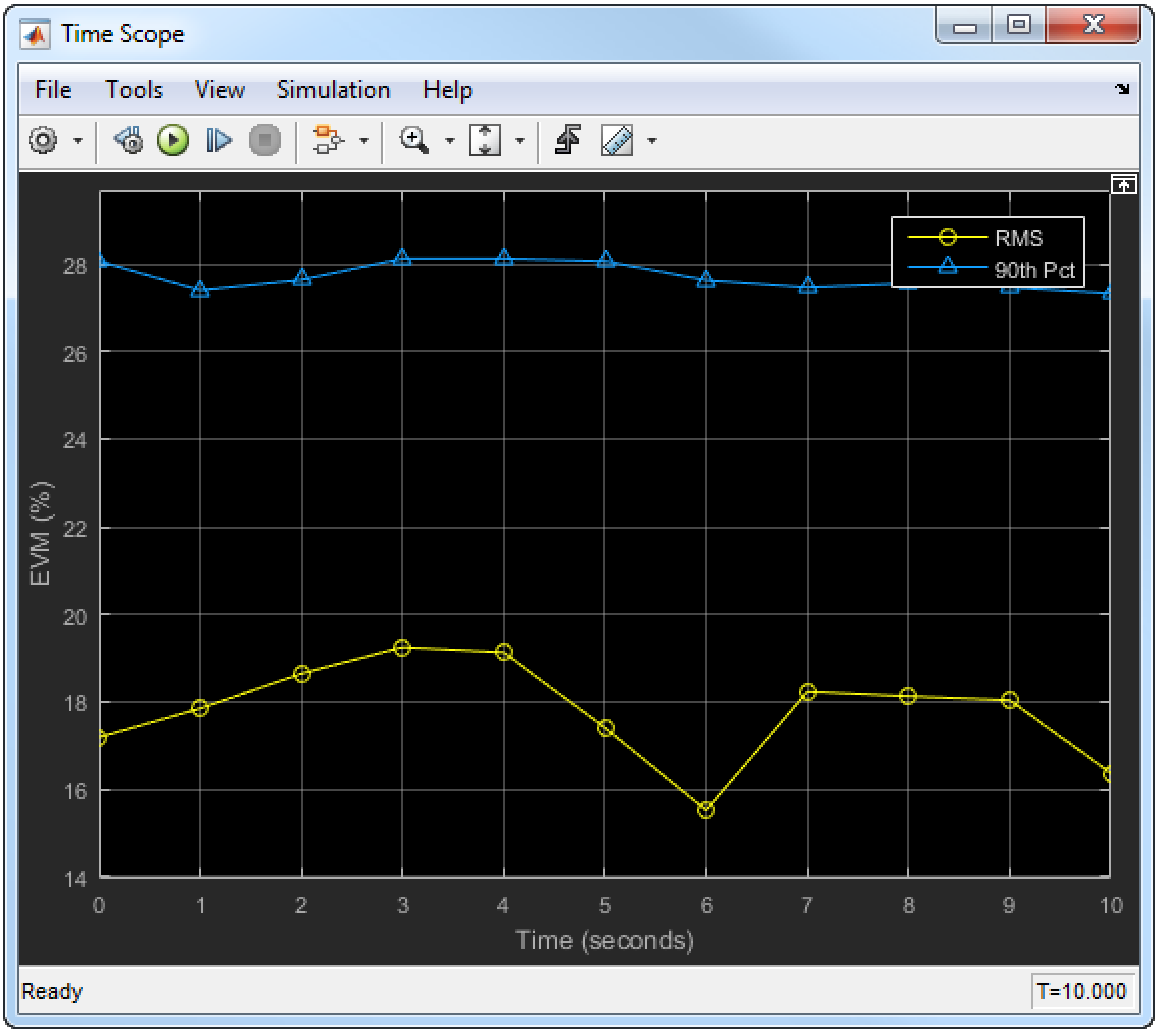
Measure RMS and 90th Percentile EVM

Measure the RMS and 90th percentile EVM for an 8-PSK signal in an AWGN channel.

Open the model by typing doc_evm_example on the command line.



Run the model. The Display block shows the number of symbols used to estimate the EVM. The Time Scope shows the RMS and 90th percentile EVM values.



Observe that 90% of the symbols had an EVM value of less than 28% and that the average EVM is approximately 17%.

Experiment with the model by changing the signal-to-noise ratio in the AWGN Channel block. Examine its effect on the EVM values.

Algorithms

Both the EVM block and the EVM object provide three normalization methods. You can normalize measurements according to the average power of the reference signal, average constellation power, or peak constellation power. Different industry standards follow one of these normalization methods.

The block or object calculates the RMS EVM value differently for each normalization method.

EVM Normalization Method	Algorithm
Reference signal	$EVM_{RMS} = \sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{avg}}}}$
Peak power	$\mathrm{EVM}_{\mathrm{RMS}}(\%) = 100\sqrt{\dfrac{\frac{1}{N}\sum_{k=1}^{N}(e_k)}{P_{\mathrm{max}}}}$

Where:

- $e_k = e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k = In-phase measurement of the kth symbol in the burst
- Q_k = Quadrature phase measurement of the kth symbol in the burst
- N = Input vector length
- P_{avg} = The value for **Average constellation power**
- P_{max} = The value for **Peak constellation power**
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The max EVM is the maximum EVM value in a frame or $\mathrm{EVM}_{\mathrm{max}} = \max\limits_{k \in [1,\ldots,N]} \{\mathrm{EVM}_k\}$, where k is the kth symbol in a burst of length N.

The definition for EVM_k varies depending upon which normalization method you select for computing measurements. The block or object supports these algorithms.

EVM Normalization	Algorithm
Reference signal	$EVM_k = \sqrt{\dfrac{e_k}{\frac{1}{N}\sum_{k=1}^{N}(I_k^2 + Q_k^2)}} * 100$
Average power	$\mathrm{EVM}_k = 100\sqrt{\dfrac{e_k}{P_{\mathrm{avg}}}}$
Peak power	$\mathrm{EVM}_k = 100\sqrt{\dfrac{e_k}{P_{\mathrm{max}}}}$

The block or object computes the X-percentile EVM by creating a histogram of all the incoming EVM_k values. The output provides the EVM value below which X% of the EVM values fall.

References

[1] IEEE Standard 802.16-2004. "Part 16: Air interface for fixed broadband wireless access systems." October 2004.

[2] 3 GPP TS 45.005 V8.1.0 (2008-05). "Radio Access Network: Radio transmission and reception".

[3] IEEE Standard 802.11a-1999. "Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: High-speed Physical Layer in the 5 GHz Band." 1999.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To generate code in a model using this block, you must enable **Dynamic Memory Allocation in MATLAB Functions**. For more information, see “Dynamic memory allocation in MATLAB functions” (Simulink).

See Also

Blocks

MER Measurement

Objects

`comm.EVM`

Topics

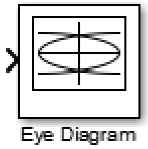
“EVM and MER Measurements with Simulink”
“Error Vector Magnitude (EVM)”

Introduced in R2009b

Eye Diagram Scope

Display eye diagram of time-domain signal

Library: Communications Toolbox / Comm Sinks
Communications Toolbox HDL Support / Comm Sinks
Mixed-Signal Blockset / Utilities
SerDes Toolbox / Utilities

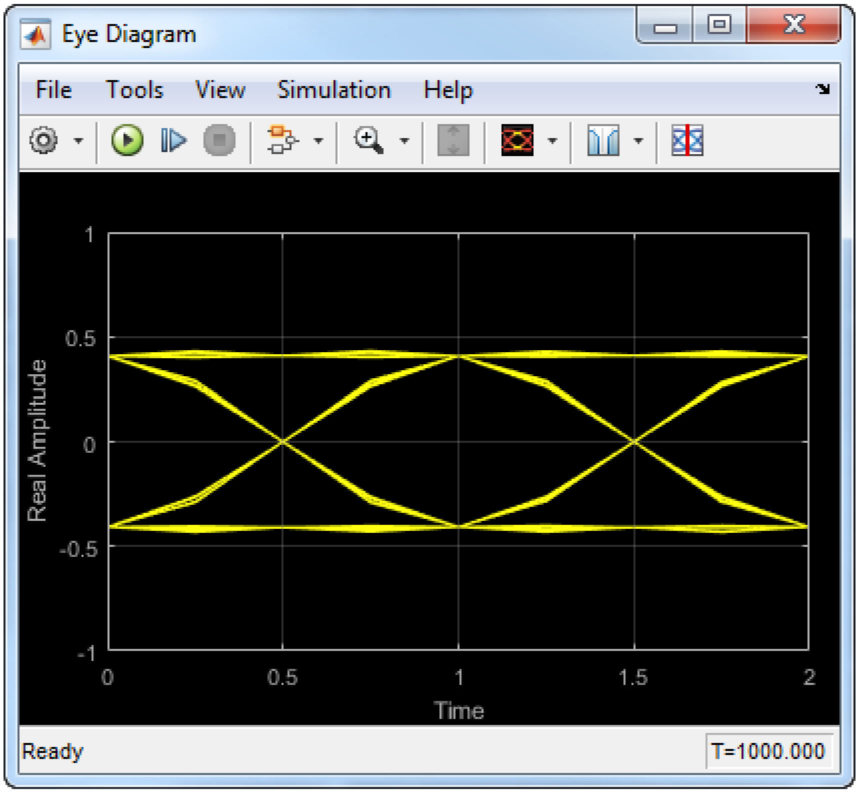


Description

The Eye Diagram block displays multiple traces of a modulated signal to produce an eye diagram. You can use the block to reveal the modulation characteristics of the signal, such as the effects of pulse shaping or channel distortions. For more information, see “Eye Diagram Analysis”.

The Eye Diagram block has one input port. This block accepts a column vector or scalar input signal. The block accepts a signal with the following data types: double, single, base integer, and fixed point. All data types are cast as double before the block displays results.

To modify the eye diagram display, select **View > Configuration Properties** or click the **Configuration Properties** button (⚙). Then select the **Main**, **2D color histogram**, **Axes**, or **Export** tabs and modify the settings.



Ports

Input

In — Input signal
scalar | column vector

Input signal, specified as a scalar or column vector.

Data Types: `double`

Parameters

Main Tab

`Display mode` — Display mode
`Line plot` (default) | `2D color Histogram`

Display mode of the eye diagram, specified as `Line plot` or `2D color histogram`. Selecting `2D color histogram` makes the histogram tab available.

Tunable: Yes

`Enable measurements` — Enable measurements
`off` (default) | `on`

Select this check box to enable eye measurements of the input signal.

`Show horizontal (jitter) histogram` — Display jitter histogram

`off` (default) | `on`

Select this radio button to display the jitter histogram. This can also be accessed by using the histogram button drop down on the toolbar.

Dependencies

This parameter is available when **Display mode** is `2D color histogram` and **Enable measurements** is selected.

`Show vertical (noise) histogram` — Display noise histogram

`off` (default) | `on`

Select this radio button to display the noise histogram. This can also be accessed by using the histogram button drop down on the toolbar.

Dependencies

This parameter is available when **Display mode** is `2D color histogram` and **Enable measurements** is selected.

`Do not show horizontal or vertical histogram` — Do not show horizontal or vertical histogram

`on` (default) | `off`

Select this radio button to display neither the histogram noise nor the histogram jitter.

Dependencies

This parameter is available when **Display mode** is `2D color histogram` and **Enable measurements** is selected.

`Show horizontal bathtub curve` — Show horizontal bathtub curve

`off` (default) | `on`

Select this check box to display the horizontal bathtub curve. This can also be accessed by using the bathtub curve button on the toolbar.

Dependencies

This parameter is available when **Enable measurements** is selected.

`Show vertical bathtub curve` — Show vertical bathtub curve

`off` (default) | `on`

Select this check box to display the vertical bathtub curve. This can also be accessed by using the bathtub curve button on the toolbar.

Dependencies

This parameter is available when **Enable measurements** is selected.

`Eye diagram to display` — Eye diagram to display

`Real only` (default) | `Real and imaginary`

Select either `Real only` or `Real and imaginary` to display one or both eye diagrams. To make eye measurements, this parameter must be `Real only`.

Tunable: Yes

`Color fading` — Color fading

`off` (default) | `on`

Select this check box to fade the points in the display as the interval of time after they are first plotted increases.

Tunable: Yes

Dependencies

This parameter is available only when the **Display mode** is `Line plot`.

`Samples per symbol` — Samples per symbol

8 (default) | positive integer

Number of samples per symbol, specified as a positive integer. Use with **Symbols per trace** to determine the number of samples per trace.

Tunable: Yes

`Sample offset` — Sample offset

0 (default) | nonnegative integer

Sample offset, specified as a nonnegative integer smaller than the product of **Samples per symbol** and **Symbols per trace**. The offset provides the number of samples to omit before plotting the first point.

Tunable: Yes

`Symbols per trace` — Symbols per trace

2 (default) | positive integer

Number of symbols plotted per trace, specified as a positive integer.

Tunable: Yes

`Traces to display` — Number of traces to display

40 (default) | positive integer

Number of traces plotted, specified as a positive integer.

Tunable: Yes

Dependencies

This parameter is available only when the **Display mode** is `Line plot`

Axes Tab

`Title` — Title label

None (default)

Label that appears above the eye diagram plot.

Tunable: Yes

`Show grid` — Toggle scope grid

`on` (default) | `off`

Toggle this check box to turn the grid on and off.

Tunable: Yes

`Y-limits (Minimum)` — Lower limit of *y*-axis

`-1.1` (default) | scalar

Minimum value of the *y*-axis.

Tunable: Yes

`Y-limits (Maximum)` — Upper limit of *y*-axis

`1.1` (default) | scalar

Maximum value of the *y*-axis.

Tunable: Yes

`Real axis label` — Real axis label

`Real Amplitude` (default)

Text that the scope displays along the real axis.

Tunable: Yes

`Imaginary axis label` — Imaginary axis label

`Imaginary Amplitude` (default)

Text that the scope displays along the imaginary axis.

Tunable: Yes

2D Histogram Tab

The 2D histogram tab is available when you click the histogram button or when the display mode is set to `2D color histogram`.

`Oversampling method` — Oversampling method

`None` (default) | `Input interpolation` | `Histogram interpolation`

Oversampling method, specified as `None`, `Input interpolation`, or `Histogram interpolation`.

To plot eye diagrams as quickly as possible, set the **Oversampling method** to `None`. The drawback to not oversampling is that the plots look pixelated when the number of samples per trace is small. To create smoother, less-pixelated plots using a small number of samples per trace, set the **Oversampling method** to `Input interpolation` or `Histogram interpolation`. `Input interpolation` is the faster of the two interpolation methods and produces good results when the signal-to-noise ratio (SNR) is high. With a lower SNR, this oversampling method is not recommended because it introduces a bias to the centers of the histogram ranges. `Histogram interpolation` is not as fast as the other techniques, but it provides good results even when the SNR is low.

Tunable: Yes

`Color scale` — Color scale
`Linear` (default) | `Logarithmic`

Color scale of the histogram plot, specified as either `Linear` or `Logarithmic`. Set **Color scale** to `Logarithmic` if certain areas of the eye diagram include a disproportionate number of points.

Tunable: Yes

The toolbar contains a histogram reset button ↻, which resets the internal histogram buffers and clears the display. This button is not available when the display mode is set to `Line plot`.

Export Tab

`Export measurements, histograms and bathtub curves` — Export measurements, histograms and bathtub curves
`Off` (default) | `off`

Select this check box export the eye diagram measurements to the MATLAB workspace.

Tunable: Yes

`Variable name` — Variable name
`EyeData` (default)

Specify the name of the variable to which the eye diagram measurements are saved. The data is saved as a structure having these fields:

- `MeasurementSettings`
- `Measurements`
- `JitterHistogram`
- `NoiseHistogram`
- `HorizontalBathtub`
- `VerticalBathtub`
- `BlockName`

Tunable: Yes

Style Dialog Box

In the **Style** dialog box, you can customize the style of the active display. You can change the color of the figure containing the displays, the background and foreground colors of display axes, and properties of lines in a display. To open this dialog box, select **View** > **Style**.

`Figure color` — Figure color
`black` (default)

Specify the background color of the scope figure.

`Axes colors` — Axes colors
`black` | `gray` (default)

Specify the fill and line colors for the axes.

`Line` — Line style, thickness and color for line plots

`continuous` | `0.5` | `yellow` (default)

Specify the line style, line width, and line color for the displayed signal.

Dependencies

This parameter is available only when the **Display mode** is `Line plot`.

`Marker` — Data point marker

None (default) | ...

Data point marker for the selected signal, specified as one of the choices in this table data point markers. This parameter is similar to the `Marker` property for MATLAB Handle Graphics® plot objects.

Specifier	Marker Type
none	No marker (default)
○	Circle
□	Square
×	Cross
•	Point
+	Plus sign
*	Asterisk
◇	Diamond
▽	Downward-pointing triangle
△	Upward-pointing triangle
◁	Left-pointing triangle
▷	Right-pointing triangle
☆	Five-pointed star (pentagram)
✡	Six-pointed star (hexagram)

Dependencies

This parameter is available only when the **Display mode** is `Line plot`.

`Colormap` — Colormap for histograms

`Hot` (default) | `Parula` | `Jet` | `HSV` | `Cool` | `SpringSummer` | `Autumn` | `Winter` | `Gray` | `Bone` | `Copper` | `Pink` | `Lines` | `Custom`

Specify the colormap of the histogram plots as one of these schemes: `Parula`, `Jet`, `HSV`, `Hot`, `Cool`, `Spring`, `Summer`, `Autumn`, `Winter`, `Gray`, `Bone`, `Copper`, `Pink`, `Lines`, or `Custom`. If you select `Custom`, a dialog box pops up from which you can enter code to specify your own colormap.

Dependencies

This parameter is available only when the **Display mode** is `2D color histogram`.

Measurement Settings Pane

To change measurement settings, first select **Enable measurements**. Then, in the **Eye Measurements** pane, click the arrow next to **Settings**. You can control these measurement settings.

`Eye level boundaries` — Time range for calculating eye levels
`[40 60]` (default) | two-element vector

Time range for calculating eye levels, specified as a two-element vector. These values are expressed as a percentage of the symbol duration.

Tunable: Yes

`Decision boundary` — Amplitude level threshold
`0` (default) | scalar

Amplitude level threshold in V, specified as a scalar. This parameter separates the different signaling regions for horizontal (jitter) histograms. This parameter is tunable, but the jitter histograms reset when the parameter changes.

For non-return-to-zero (NRZ) signals, set **Decision boundary** to 0. For return-to-zero (RZ) signals, set **Decision boundary** to half the maximum amplitude.

Tunable: Yes

`Rise/Fall thresholds` — Amplitude levels of the rise and fall transitions
`[10 90]` (default) | two-element vector

Amplitude levels of the rise and fall transitions, specified as a two-element vector. These values are expressed as a percentage of the eye amplitude. This parameter is tunable, but the crossing histograms of the rise and fall thresholds reset when the parameter changes.

Tunable: Yes

`Hysteresis` — Amplitude tolerance of the horizontal crossings
`0` (default) | scalar

Amplitude tolerance of the horizontal crossings in V, specified as a scalar. Increase hysteresis to provide more tolerance to spurious crossings due to noise. This parameter is tunable, but the jitter and the rise and fall histograms reset when the parameter changes.

Tunable: Yes

`BER threshold` — BER used for eye measurements
`1e-12` (default) | nonnegative scalar from 0 to 0.5

BER used for eye measurements, specified as a nonnegative scalar from 0 to 0.5. The value is used to make measurements of random jitter, total jitter, horizontal eye openings, and vertical eye openings.

Tunable: Yes

`Bathtub BERs` — BER values used to calculate openings of bathtub curves
`[0.5 0.1 0.01 0.001 0.0001 1e-05 1e-06 1e-07 1e-08 1e-09 1e-10 1e-11 1e-12]` (default) | vector

BER values used to calculate openings of bathtub curves, specified as a vector whose elements range from 0 to 0.5. Horizontal and vertical eye openings are calculated for each of the values specified by this parameter.

Tunable: Yes

Dependencies

To enable this parameter, select **Show horizontal bathtub curve**, **Show vertical bathtub curve**, or both.

`Measurement delay` — Duration of initial data discarded from measurements

`0` (default) | nonnegative scalar

Duration of initial data discarded from measurements, in seconds, specified as a nonnegative scalar.

Block Characteristics

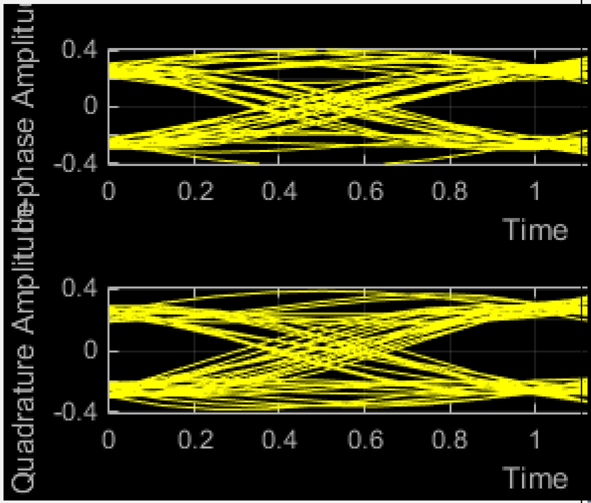
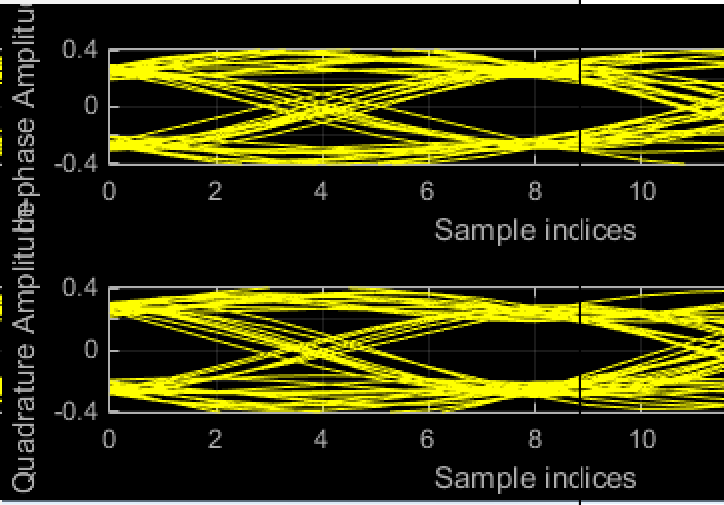
Data Types	`Boolean` | `double` | `enumerated` | `fixed point` | `integer` | `single`
Direct Feedthrough	`no`
Multidimensional Signals	`no`
Variable-Size Signals	`no`
Zero-Crossing Detection	`no`

More About

Using Eye Diagram in Conditionally Executed Subsystems

When an Eye Diagram block is placed in a conditionally executed subsystem, for example in a triggered or enabled subsystem:

- Input size must be an integer multiple of `SamplesPerSymbol` * `SymbolsPerTrace`
- Sample offset must be zero
- The rightmost part of the display is intentionally omitted. This figure compares typical eye diagram display when placed in a normal system versus one placed in a conditionally executed subsystem.

Eye Diagram Plot in Normal System	Eye Diagram Plot in Conditionally Executed Subsystem
	
In a regular Eye Diagram, the rightmost part is a line between the last sample of a trace and the first sample of the next trace.	In conditionally executed subsystems, these traces may be non-contiguous, thus this rightmost segment could corrupt the display and is omitted.

Measurements

Measurements assume that the eye diagram object has valid data. A valid eye diagram has two distinct eye crossing points and two distinct eye levels.

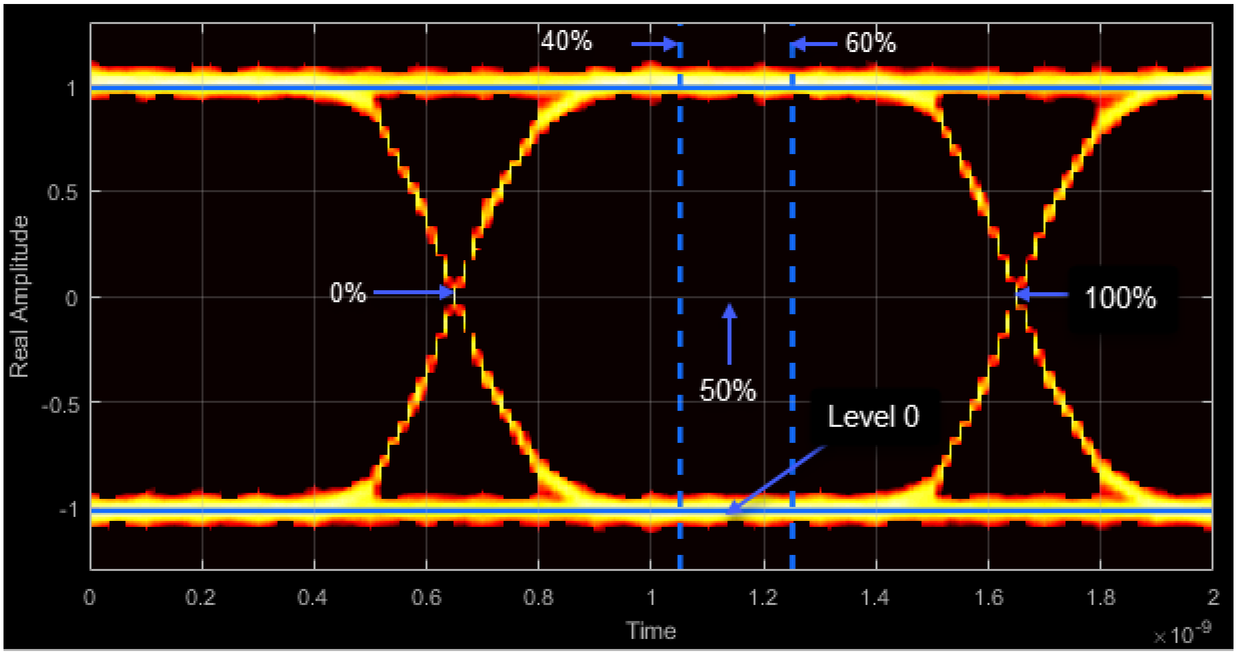
To open the measurements pane, click on the **Eye Measurements** button or select **Tools > Measurements > Eye Measurements** from the toolbar menu.

Note

- For amplitude measurements, at least one bin per vertical histogram must reach 10 hits before the measurement is taken, ensuring higher accuracy.
- For time measurements, at least one bin per horizontal histogram must reach 10 hits before the measurement is taken.
- When an eye crossing time measurement falls within the [-0.5/Fs, 0) seconds interval, the time measurement wraps to the end of the eye diagram, i.e., the measurement wraps by 2×T_s seconds (where T_s is the symbol time). For a complex signal case, the analyze method issues a warning if the crossing time measurement of the in-phase branch wraps while that of the quadrature branch does not (or vice versa). To avoid time-wrapping or a warning, add a half-symbol duration delay to the current value in the `MeasurementDelay` property of the eye diagram object. This additional delay repositions the eye in the approximate center of the scope.

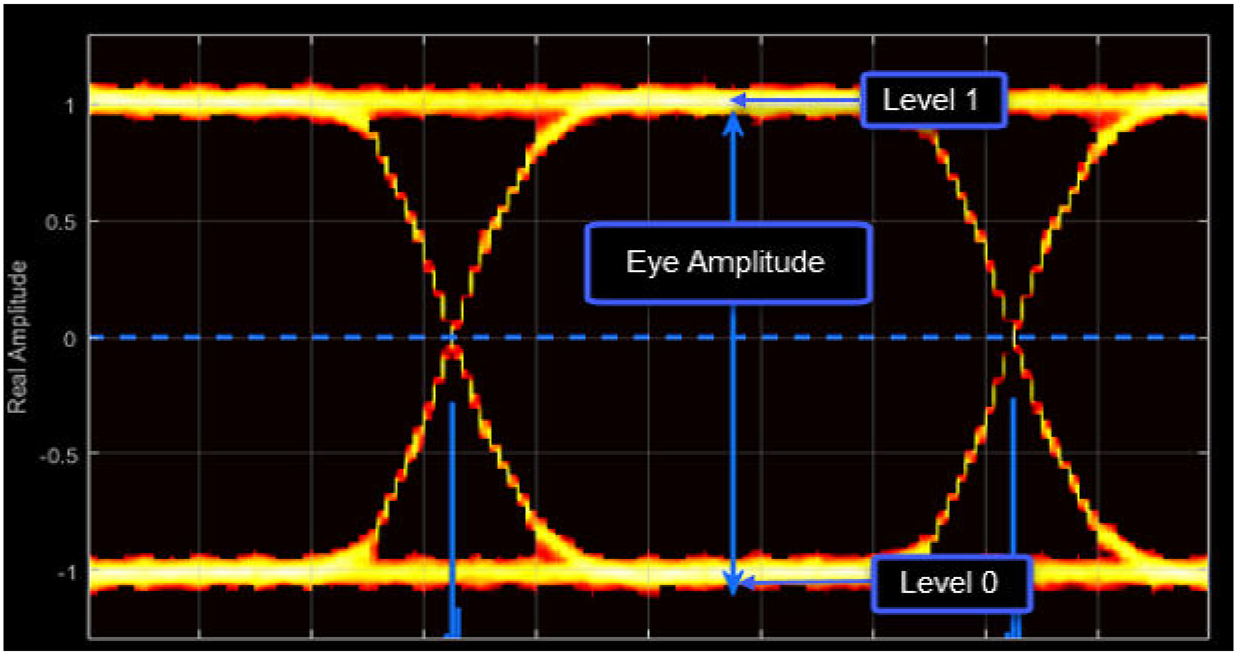
Eye Levels - Amplitude level used to represent data bits

Eye level is the amplitude level used to represent data bits. For the displayed NRZ signal, the levels are –1 V and +1 V. The eye levels are calculated by averaging the 2-D histogram within the eye level boundaries. For example, when the EyeLevelBoundaries property is set to `[40 60]`, that is, 40% and 60% of the symbol duration, the eye levels are calculated by estimating the mean value of the vertical histogram in this window marked by the eye level boundaries.



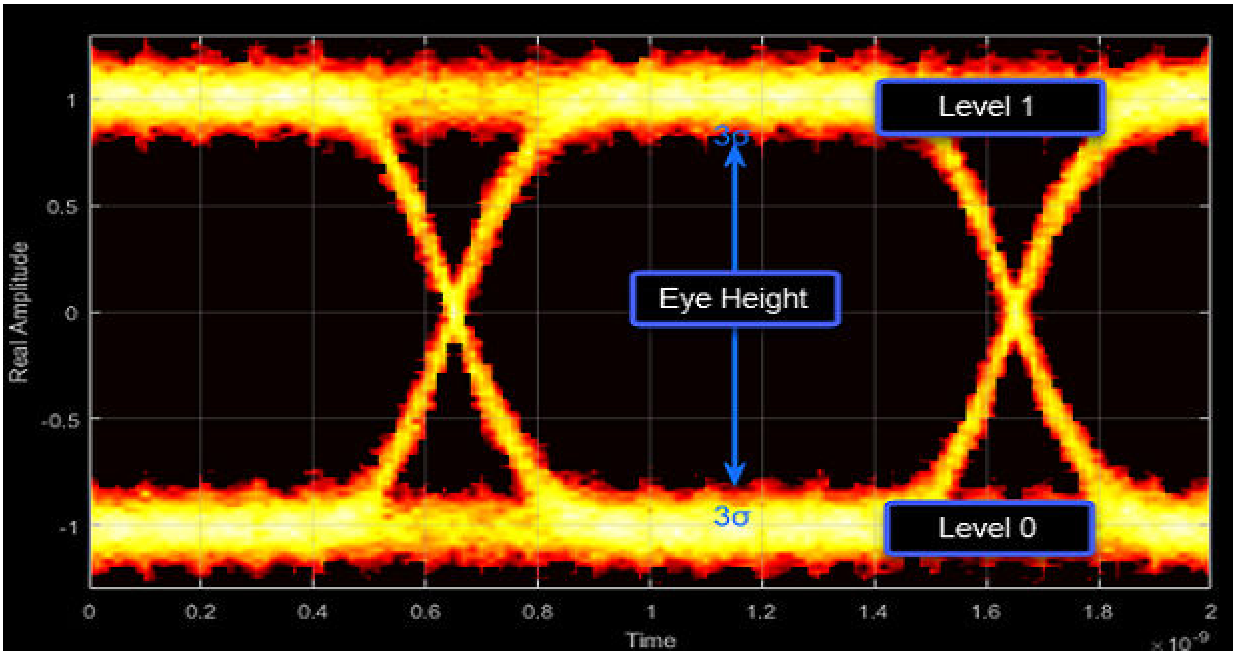
Eye Amplitude - Distance between eye levels

Eye amplitude is the distance in V between the mean value of two eye levels.



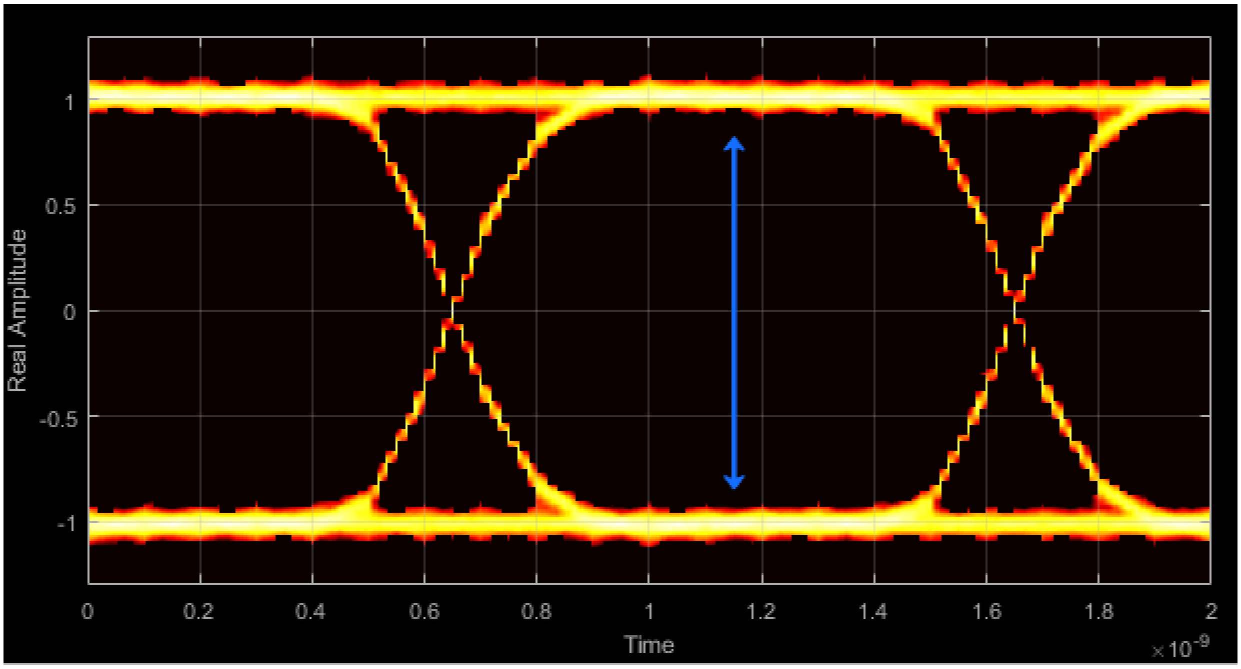
Eye Height - Statistical minimum distance between eye levels

Eye height is the distance between μ – 3σ of the upper eye level and μ + 3σ of the lower eye level. μ is the mean of the eye level, and σ is the standard deviation.



Vertical Opening - Distance between BER threshold points

The vertical opening is the distance between the two points that correspond to the BERThreshold property. For example, for a BER threshold of 10^{-12}, these points correspond to the 7σ distance from each eye level.



Eye SNR - Signal-to-noise ratio

The eye SNR is the ratio of the eye level difference to the difference of the vertical standard deviations corresponding to each eye level:

$$\text{SNR} = \frac{L_1 - L_0}{\sigma_1 - \sigma_0},$$

where L_1 and L_0 represent the means of the upper and lower eye levels and σ_1 and σ_0 represent their standard deviations.

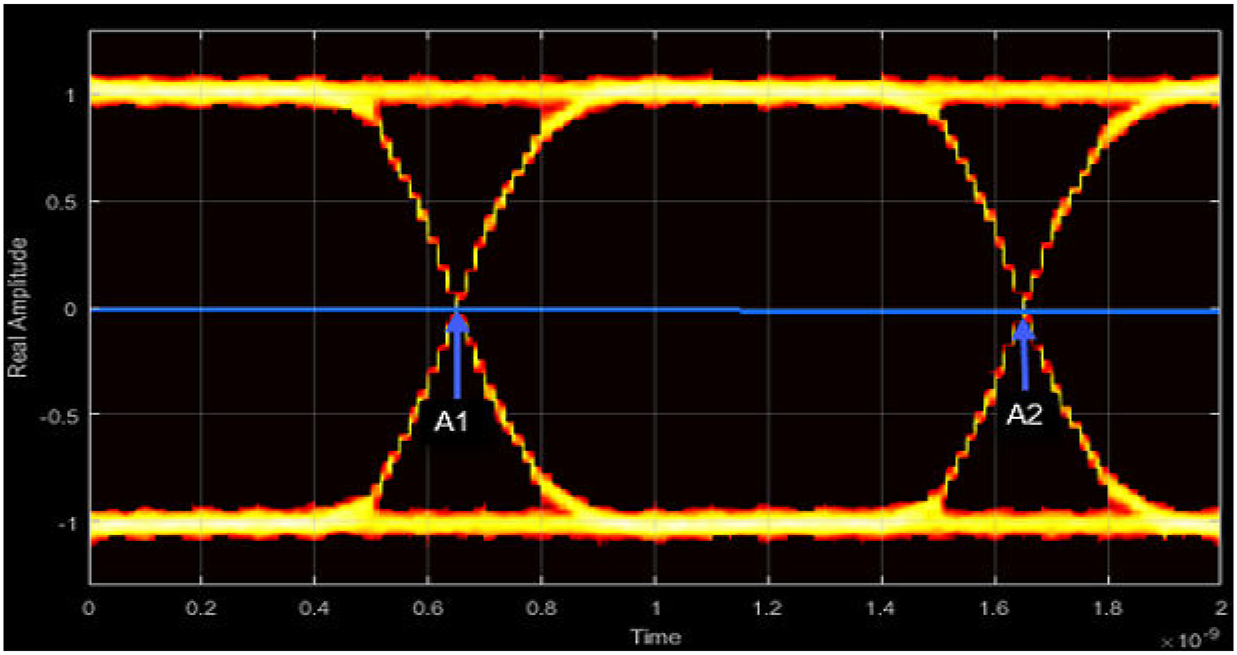
Q Factor - Quality factor

The Q factor is the quality factor and is calculated using the same formula as the eye SNR. However, the standard deviations of the vertical histograms are replaced with those computed with the dual-Dirac analysis.

Crossing Levels - Amplitude levels for eye crossings

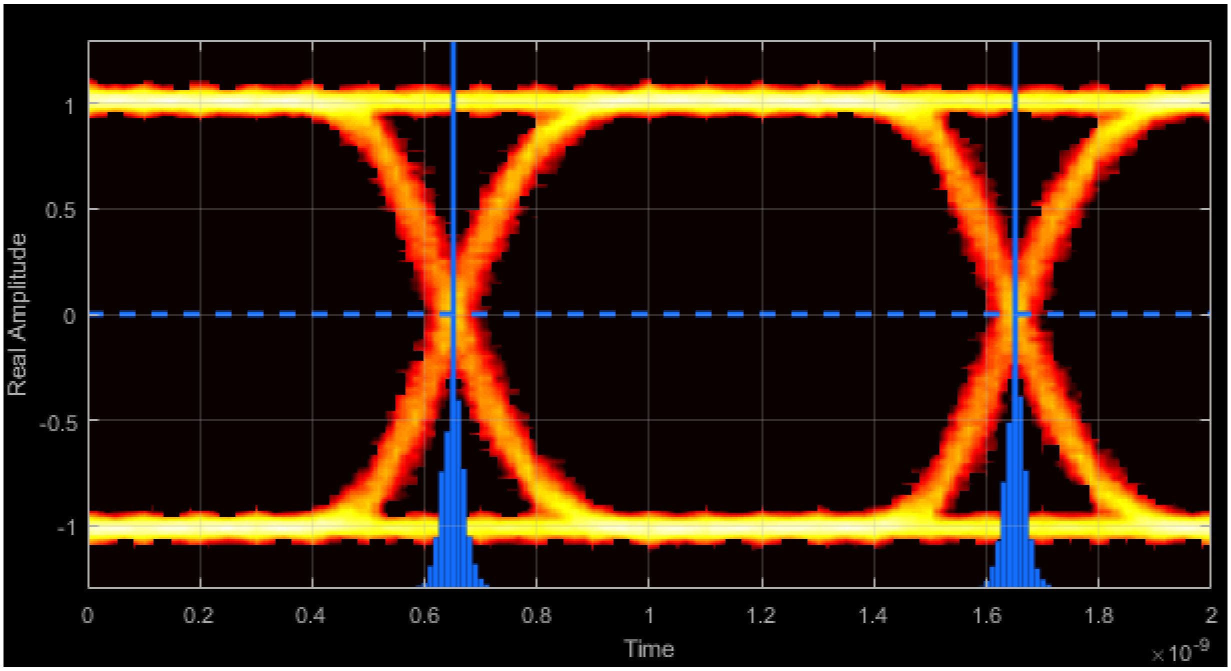
The crossing levels are the amplitude levels at which the eye crossings occur.

The level at which the input signal crosses the amplitude value is specified by the DecisionBoundary property.



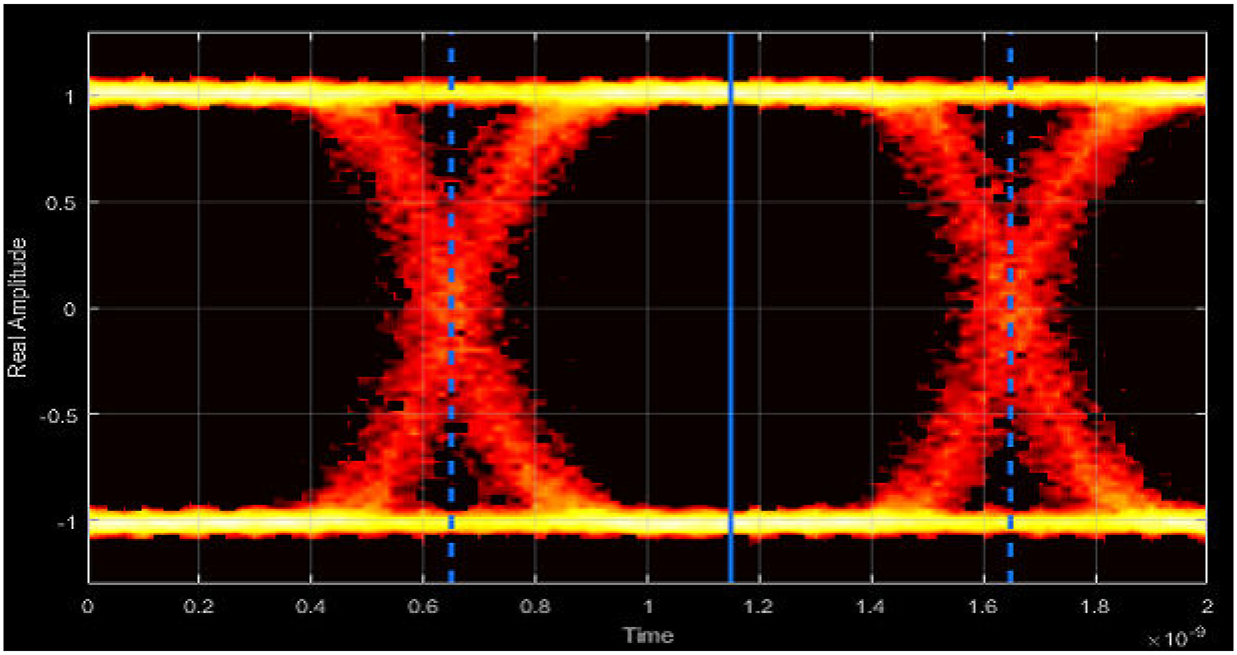
Crossing Times - Times for which crossings occur

The crossing times are the times at which the crossings occur. The times are computed as the mean values of the horizontal (jitter) histograms.



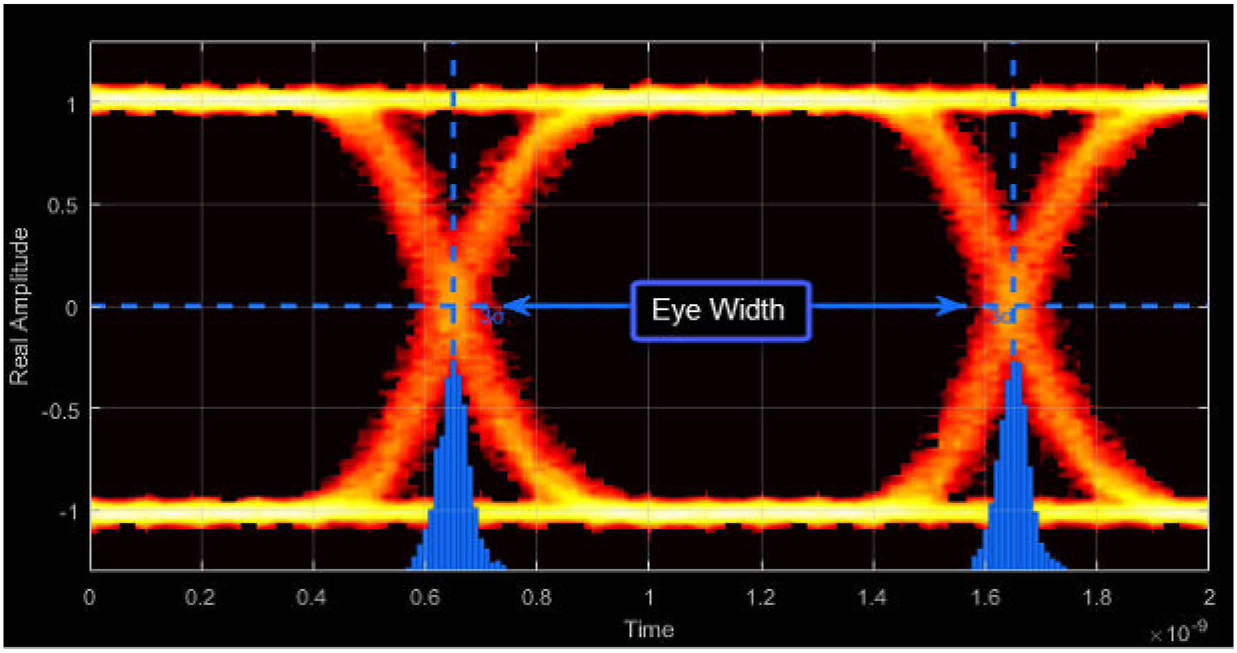
Eye Delay - Mean time between eye crossings

Eye delay is the midpoint between the two crossing times.



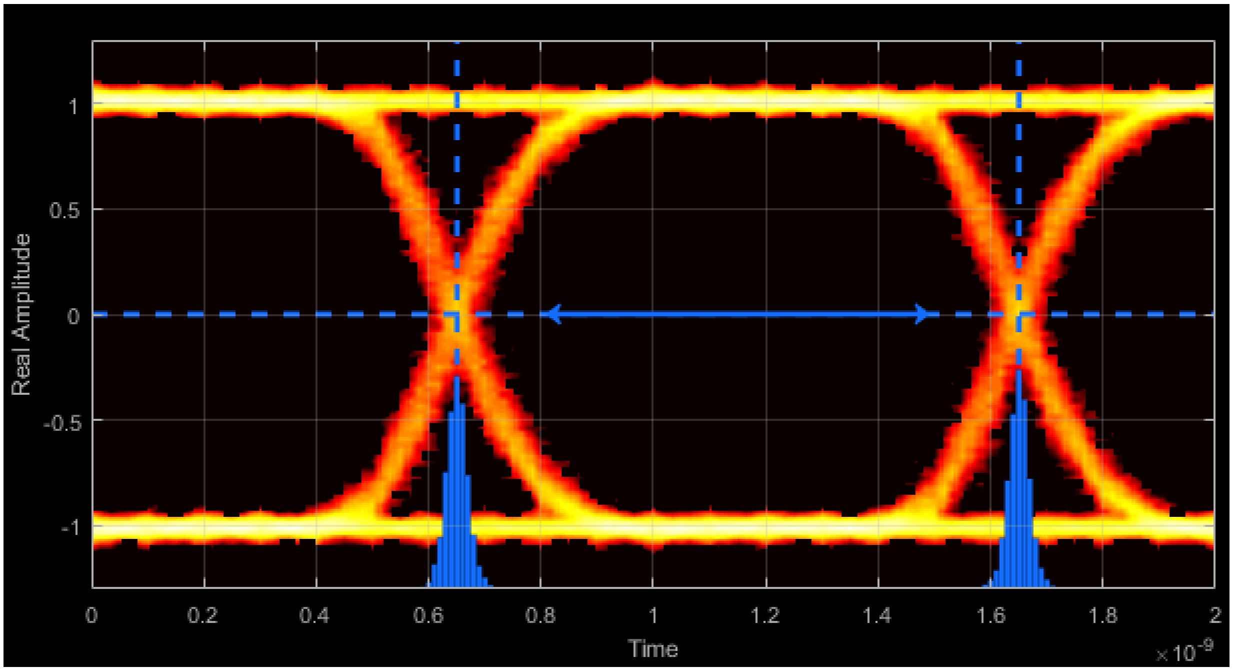
Eye Width - Statistical minimum time between eye crossings

Eye width is the horizontal distance between μ + 3σ of the left crossing time and μ – 3σ of the right crossing time. μ is the mean of the jitter histogram, and σ is the standard deviation.



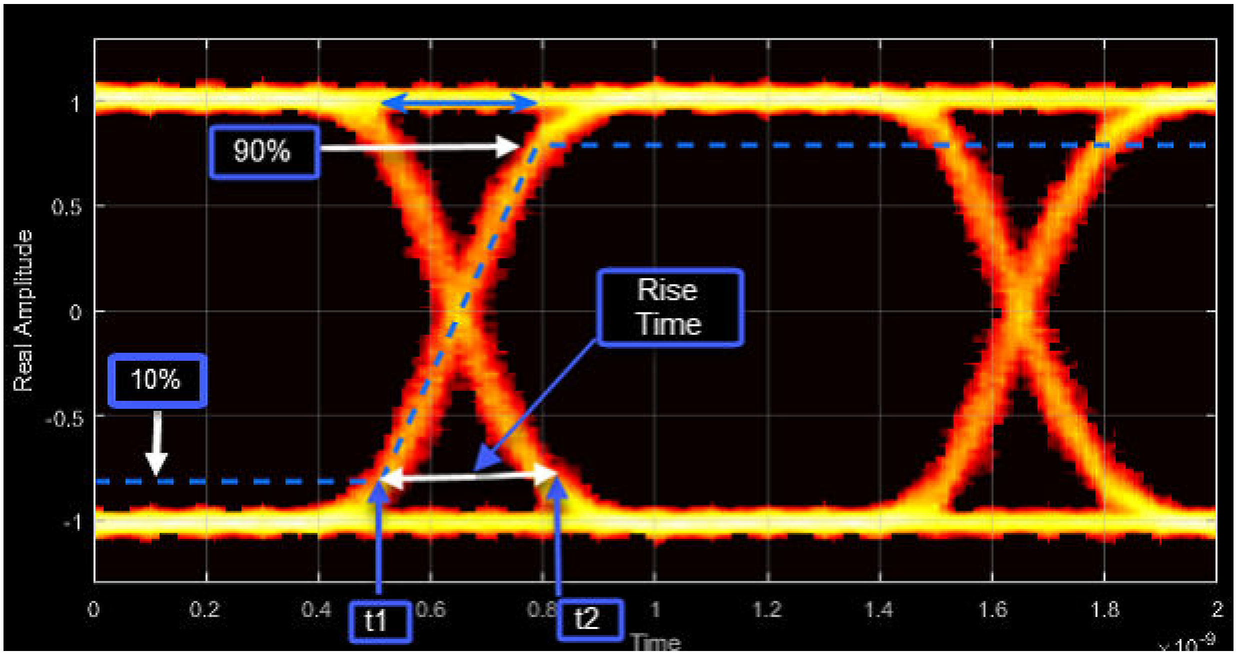
Horizontal Opening - Time between BER threshold points

The horizontal opening is the distance between the two points that correspond to the BERThreshold property. For example, for a 10^{-12} BER, these two points correspond to the 7σ distance from each crossing time.



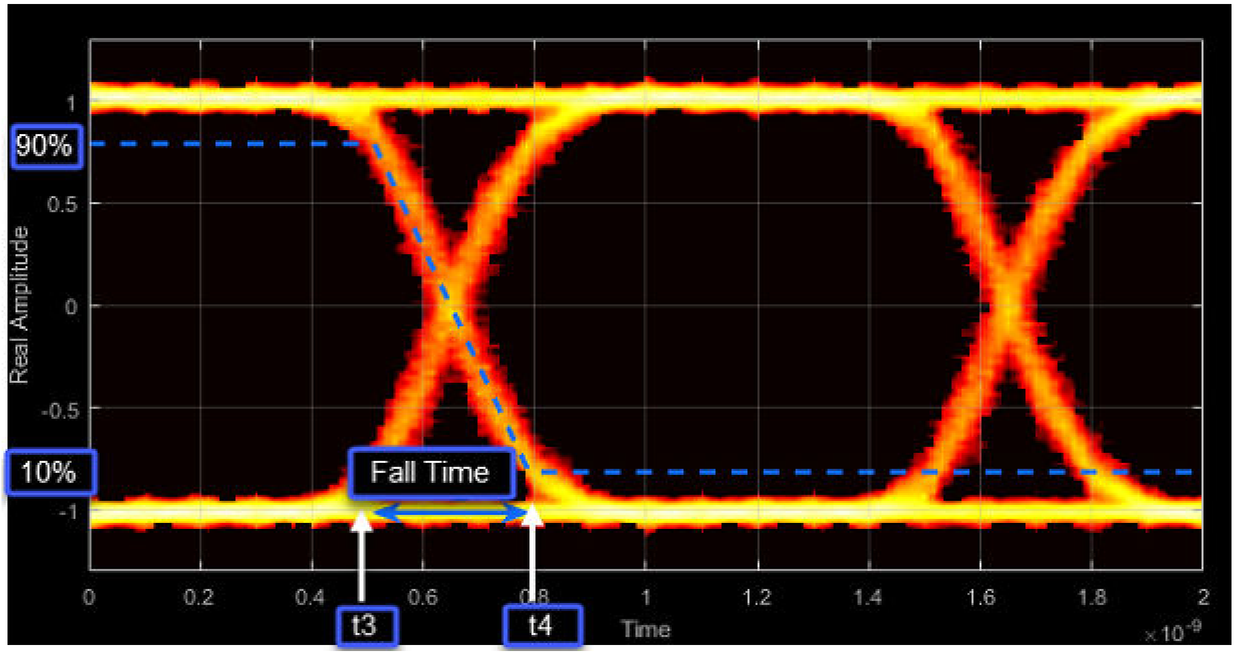
Rise Time - Time to transition from low to high

Rise time is the mean time between the low and high rise/fall thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



Fall Time - Time to transition from high to low

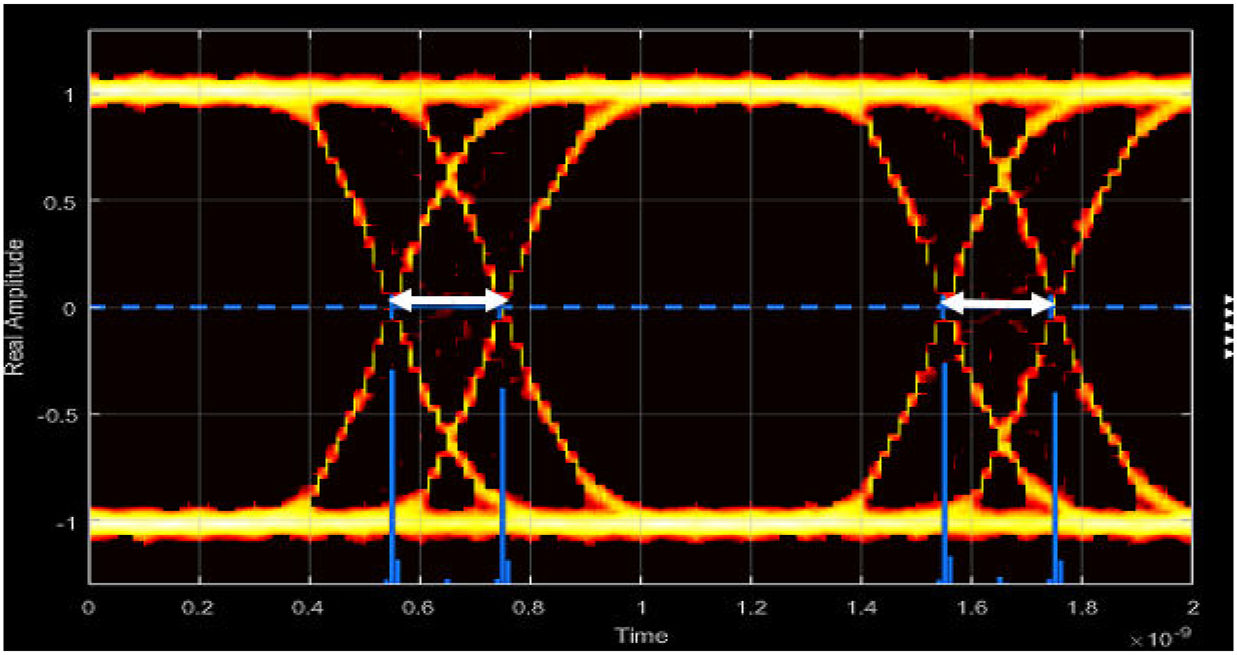
Fall time is the mean time between the high and low rise/fall thresholds defined in the eye diagram. The default thresholds are 10% and 90% of the eye amplitude.



Deterministic Jitter - Deterministic deviation from ideal signal timing

Jitter is the deviation of a signal’s timing event from its intended (ideal) occurrence in time [2]. Jitter can be represented with a dual-Dirac model. A dual-Dirac model assumes that the jitter has two components: deterministic jitter (DJ) and random jitter (RJ).

DJ is the distance between the two peaks of the dual-Dirac histograms. The probability density function (PDF) of DJ is composed of two delta functions.



Random Jitter - Random deviation from ideal signal timing

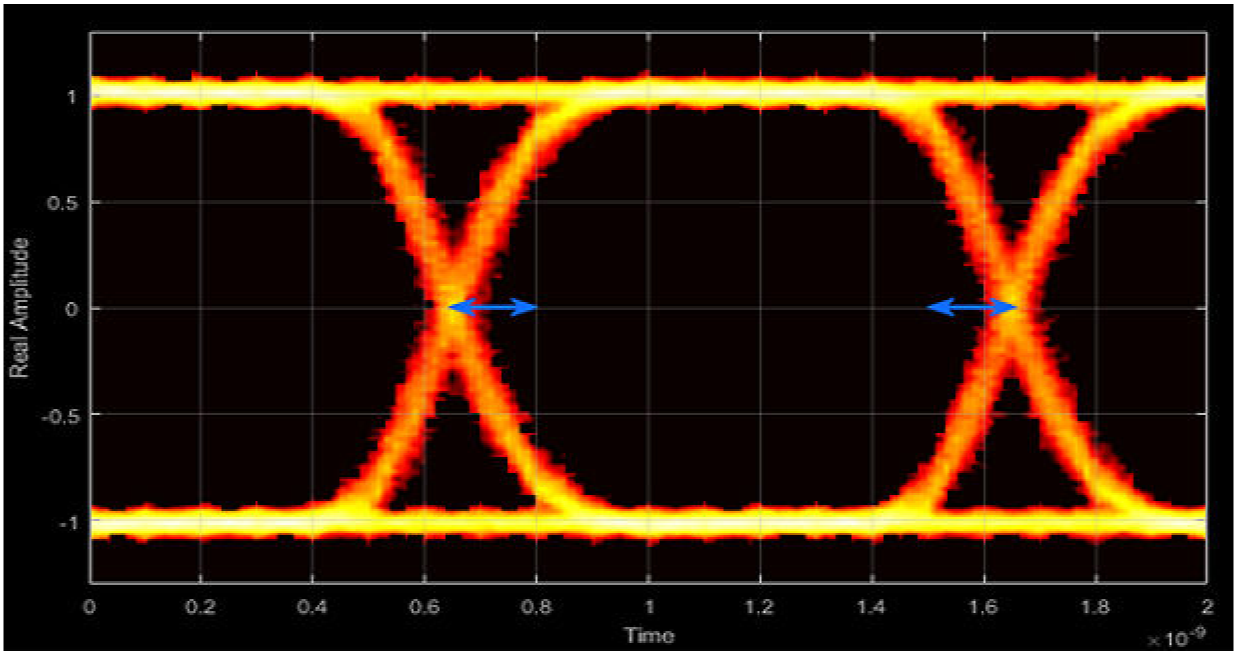
RJ is the Gaussian unbounded jitter component. The random component of jitter is modeled as a zero-mean Gaussian random variable with a specified standard-deviation of σ. The RJ is computed as:

$$\mathrm{RJ} = (Q_L + Q_R)\sigma\,,$$

where

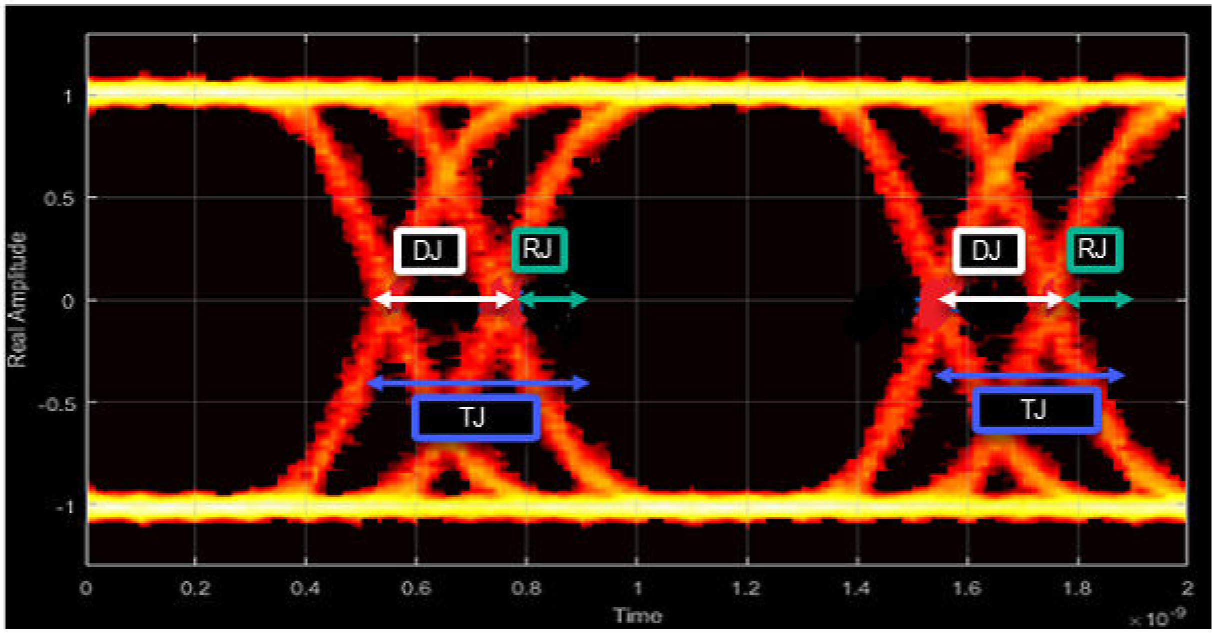
$$Q = \sqrt{2}\mathrm{erfc}^{-1}\left(2\frac{BER}{\rho}\right)\,.$$

BER is the specified BER threshold. ρ is the amplitude of the left and right Dirac function, which is determined from the bin counts of the jitter histograms.

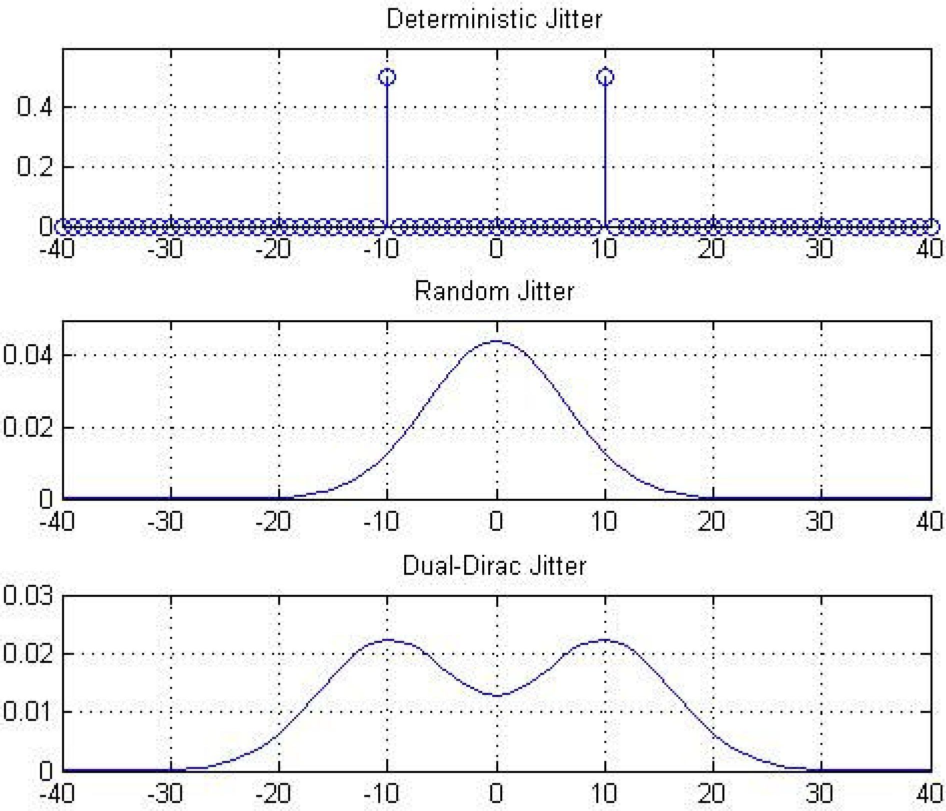


Total Jitter - Deviation from ideal signal timing

Total jitter (TJ) is the sum of the deterministic and random jitter, such that TJ = DJ + RJ.

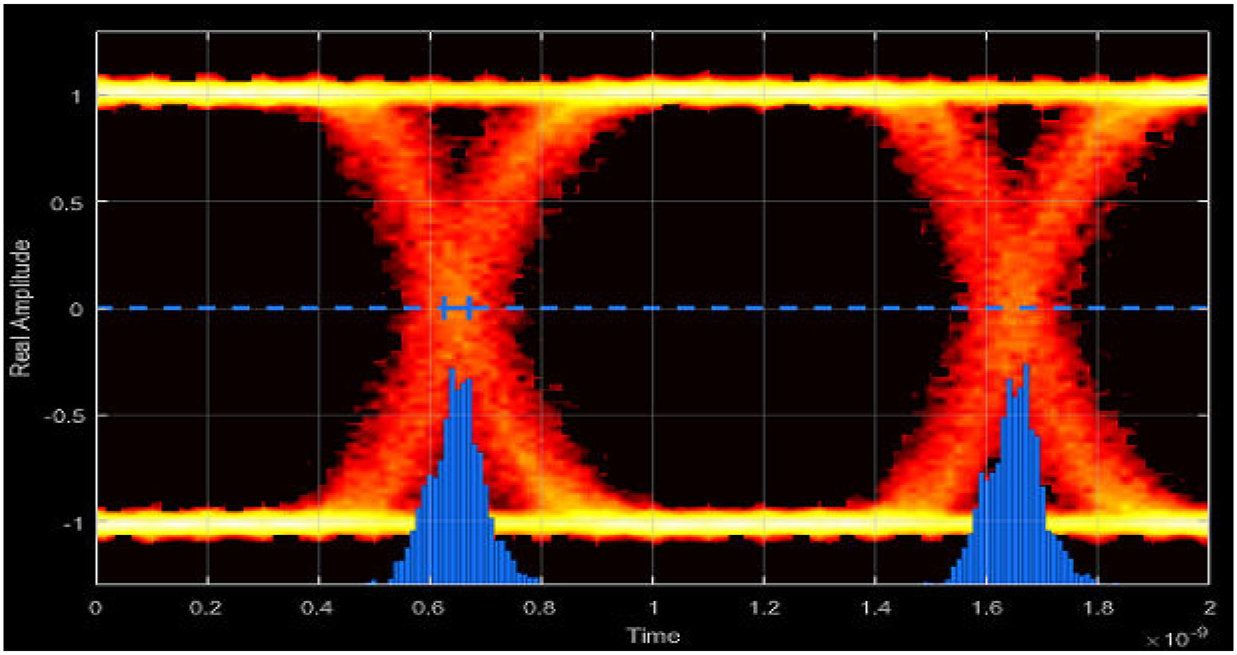


The total jitter PDF is the convolution of the DJ PDF and the RJ PDF.



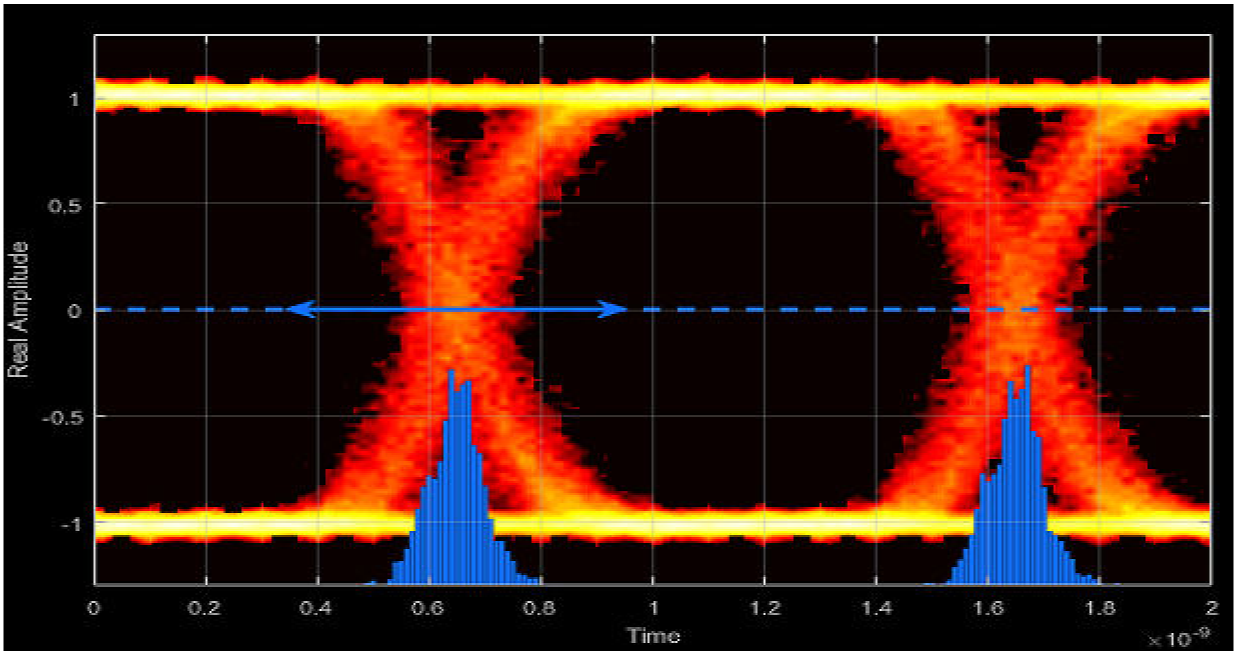
RMS Jitter - Standard deviation of jitter

RMS jitter is the standard deviation of the jitter calculated in the horizontal (jitter) histogram at the decision boundary.



Peak-to-Peak Jitter - Distance between extreme data points of histogram

Peak-to-peak jitter is the maximum horizontal distance between the left and right nonzero values in the horizontal histogram of each crossing time.

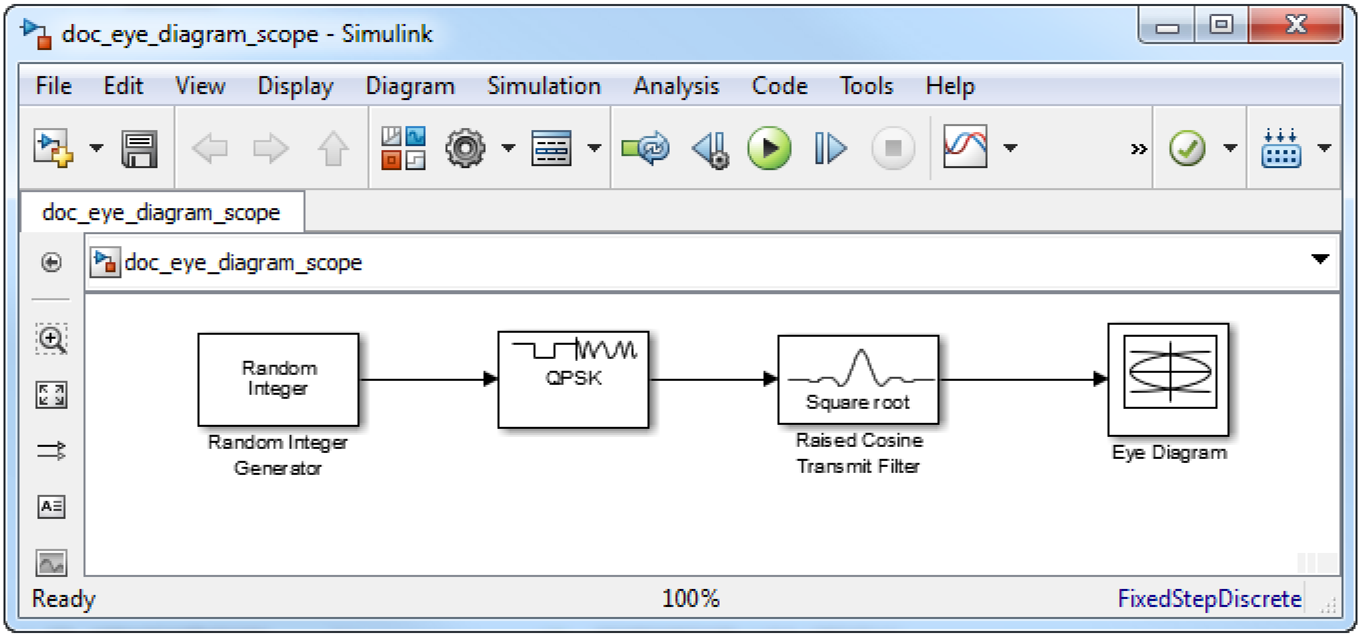


View Eye Diagram

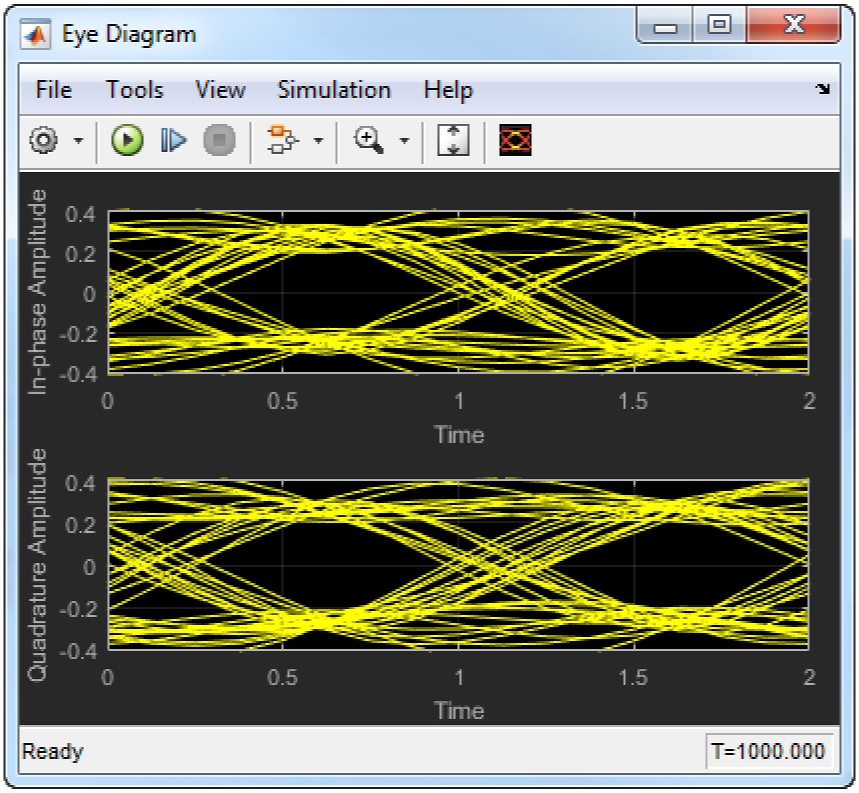
Display the eye diagram of a filtered QPSK signal using the Eye Diagram block.

Load the doc_eye_diagram_scope model from the MATLAB command prompt.

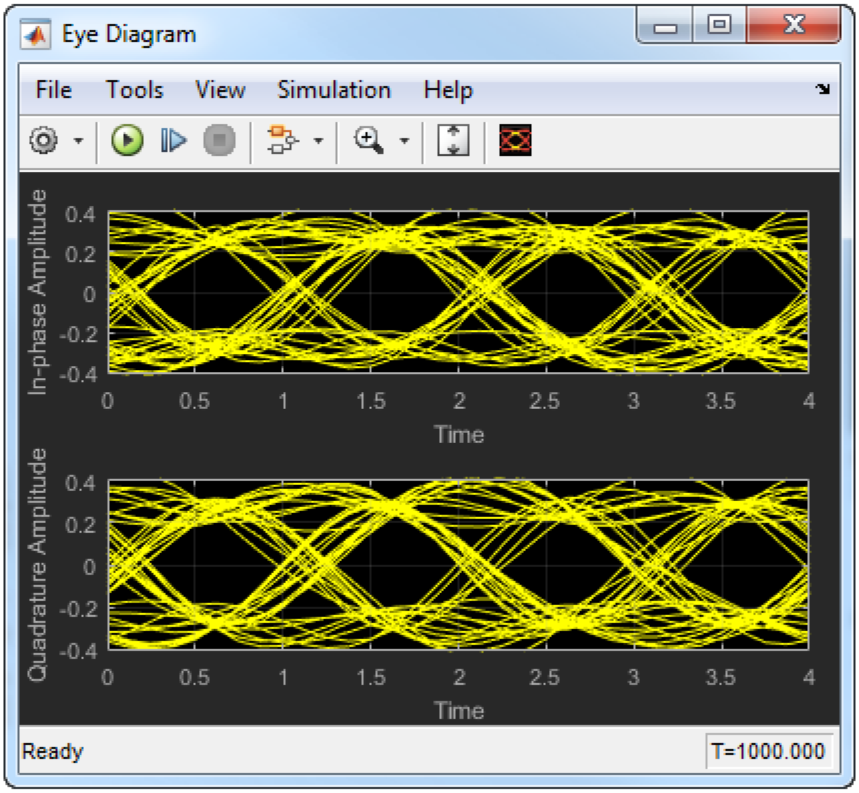
```
doc_eye_diagram_scope
```



Run the model and observe that two symbols are displayed.



Open the configuration parameters dialog box. Change the **Symbols per trace** parameter to 4. Run the simulation and observe that four symbols are displayed.



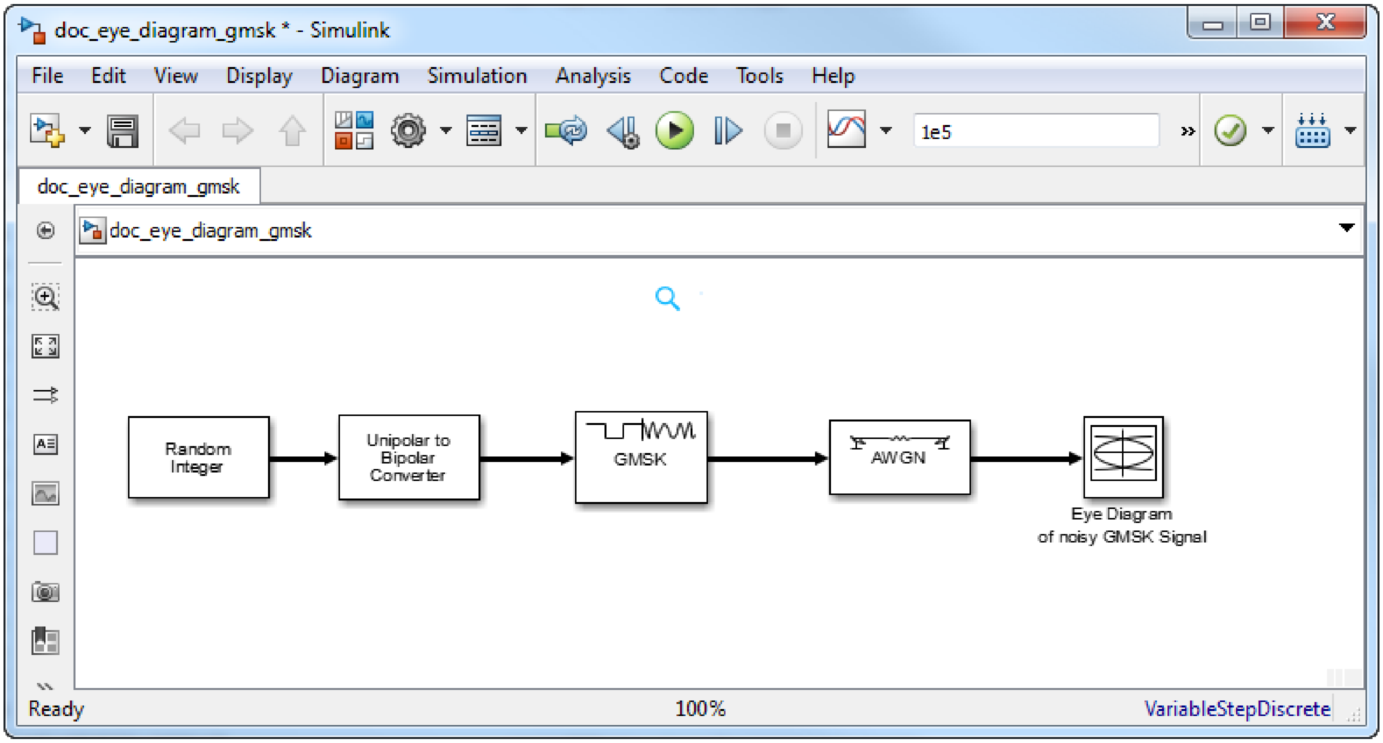
Try changing the Raised Cosine Transmit Filter parameters or changing additional Eye Diagram parameters to see their effects on the eye diagram.

Histogram Plots

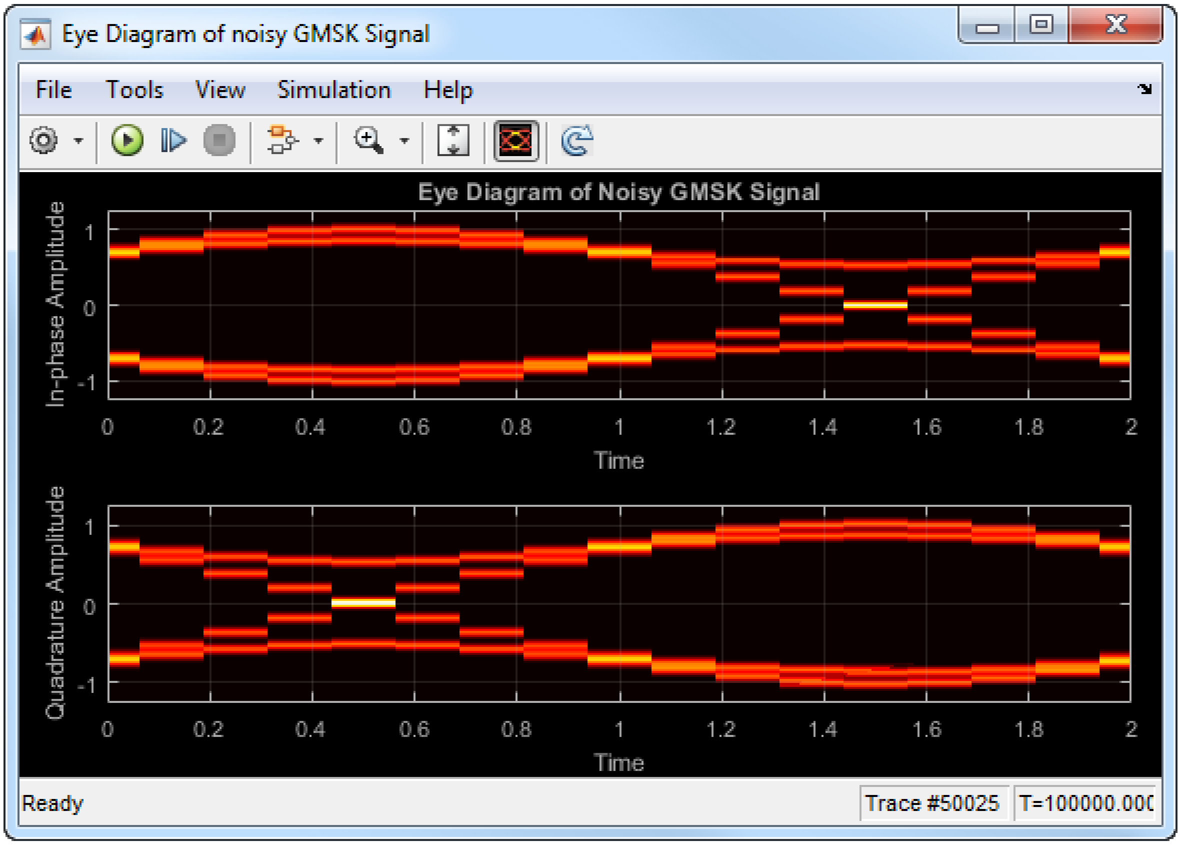
Display histogram plots of a noisy GMSK signal.

Load the doc_eye_diagram_gmsk model from the MATLAB command prompt.

```
doc_eye_diagram_gmsk
```

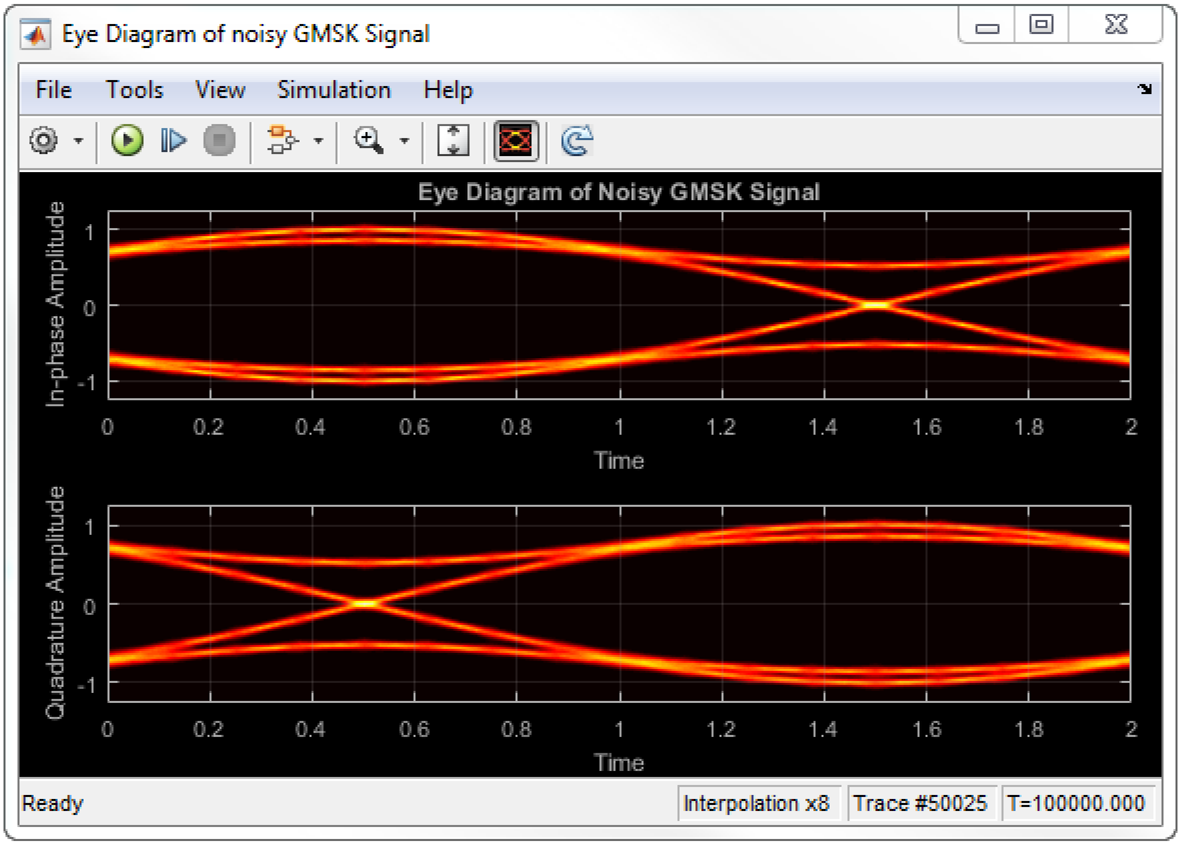

doc_eye_diagram_gmsk * - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
1e5
doc_eye_diagram_gmsk
doc_eye_diagram_gmsk
Random Integer
Unipolar to Bipolar Converter
GMSK
AWGN
Eye Diagram of noisy GMSK Signal
Ready
100%
VariableStepDiscrete

Run the model. The eye diagram is configured to show a histogram without interpolation.



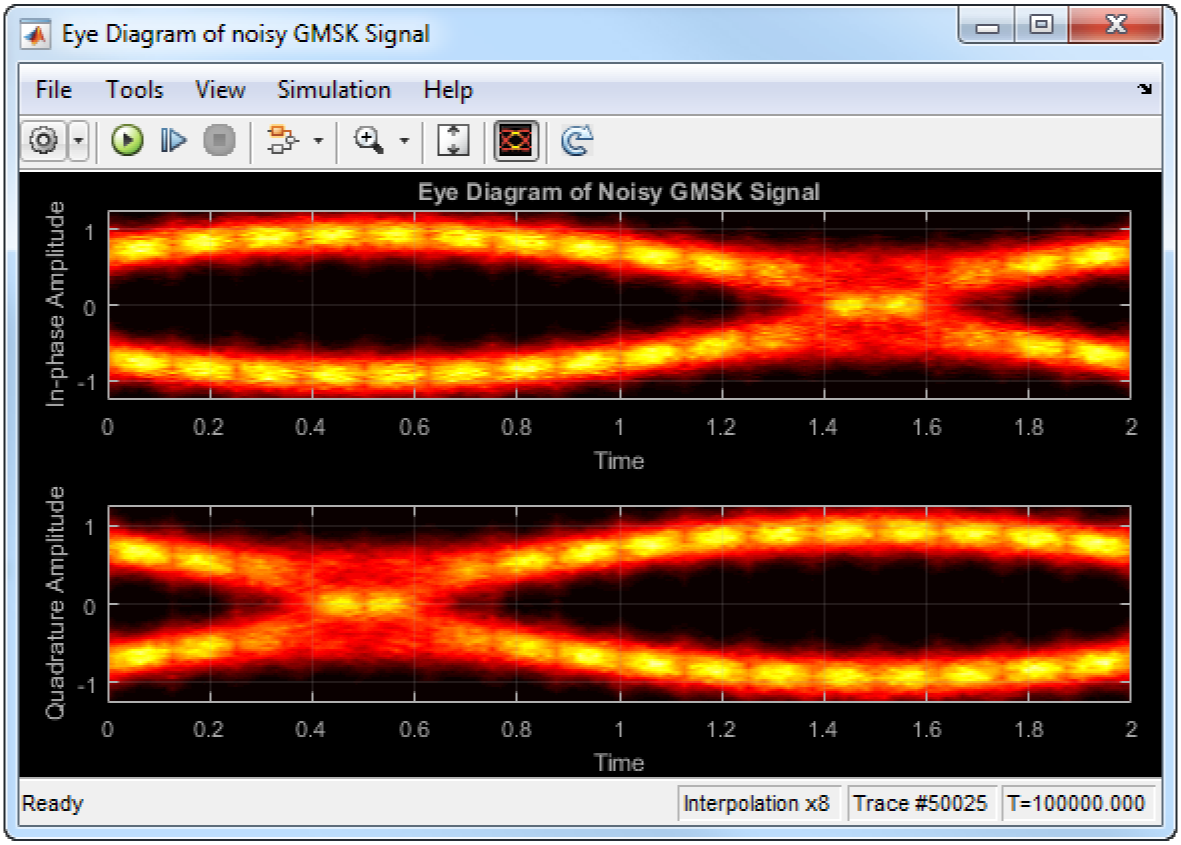
The lack of interpolation results in a plot having piecewise-continuous behavior.

Open the **2D Histogram** tab of the Configuration Properties dialog box. Set the **Oversampling method** to `Input interpolation`. Run the model.



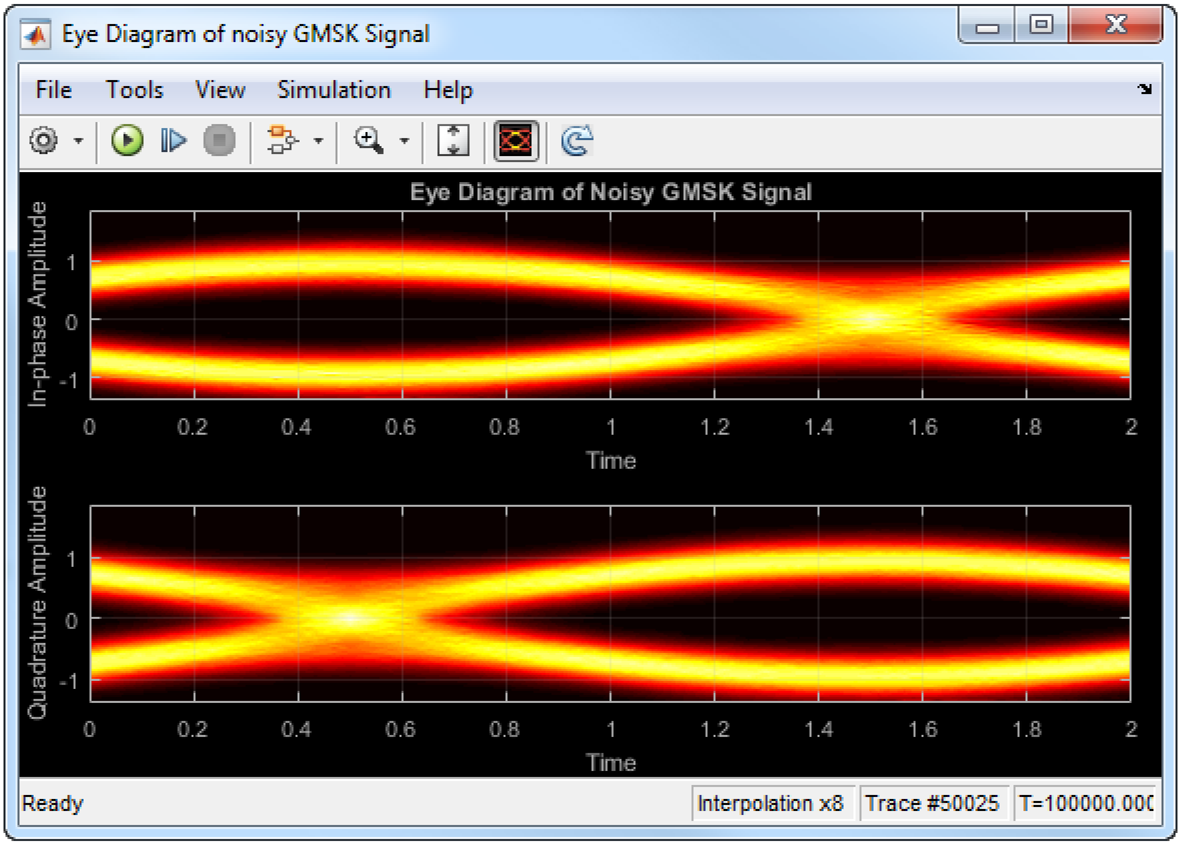
The interpolation smooths the eye diagram.

On the AWGN Channel block, change **SNR (dB)** from `25` to `10`. Run the model.



Observe that vertical striping is present in the eye diagram. This striping is the result of input interpolation, which has limited accuracy in low-SNR conditions.

Set the **Oversampling method** to `Histogram interpolation`. Run the model.



The eye diagram plot now renders accurately because the histogram interpolation method works for all SNR values. This method is not as fast as the other techniques and results in increased execution time.

Programmatic Configuration

You can programmatically configure the scope properties with callbacks or within scripts by using a scope configuration object as describe in “Control Scope Blocks Programmatically” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

This block is excluded from the generated code when code generation is performed on a system containing this block.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

This block can be used for simulation visibility in subsystems that generate HDL code, but is not included in the hardware implementation.

See Also

Blocks
Constellation Diagram

Functions
`eyediagram`

Topics
“Eye Diagram Analysis”

Introduced in R2014a

Find Delay

Find delay between two signals
Library: Communications Toolbox / Utility Blocks

Description

The Find Delay block finds the delay between a signal and a delayed, and possibly distorted, version of itself. This is useful when you want to compare a transmitted and received signal to find the bit error rate, but do not know the delay in the received signal. This block accepts a column vector or matrix input signal. For a matrix input, the block outputs a row vector, and finds the delay in each channel of the matrix independently. See “Delays” for more information about signal delays.

Ports

Input

`sRef` — Reference signal
vector | matrix

Original reference signal, specified as a vector or matrix. Dimensions and sample times of **sRef** and **sDel** must match.

Data Types: `double`

`sDel` — Delayed signal
vector | matrix

Delayed or distorted version of reference signal, specified as a vector or matrix. Dimensions and sample times of **sRef** and **sDel** must match.

Data Types: `double`

Output

`delay` — Delay output
scalar | vector

The output port labeled **delay** outputs the delay in units of samples.

For a matrix input, the output is a row vector, and finds the delay in each channel of the matrix independently

The delay output is a nonnegative integer less than the **Correlation window length**.

Data Types: `double`

`chg` — Delay flag
`0` | `1`

The **chg** output port outputs `1` when there is a change from the delay computed at the previous sample, and `0` when there is no change

Dependencies

This output port becomes visible only when `Include "change signal" output port` is selected.

Data Types: `Boolean`

Parameters

`Correlation window length (samples)` — Number of samples

`200` (default) | positive integer

The number of samples the block uses to calculate the cross-correlations of the two signals.

As the **Correlation window length** is increased, the reliability of the computed delay also increases. However, the processing time to compute the delay increases as well.

`Include "change signal" output port` — Enable chg port

`off` (default) | `on`

If you select this option, then the block has an extra output port that emits a value of 1 when the current computed delay differs from the previous computed delay and emits a value of 0 when there is no delay.

`Disable recurring updates` — Disable recurring updates

`off` (default) | `on`

Selecting this option causes the block to stop computing the delay after it computes the same delay value for a specified number of samples.

`Number of constant delay outputs to disable updates` — Number of constant delay outputs to disable updates

`3` (default) | positive integer

A positive integer specifying how many times the block must compute the same delay before ceasing to update.

Dependencies

This field becomes visible only when **Disable recurring updates** is selected.

Block Characteristics

Data Types	`double` | `enumerated` | `integer`[a] | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

[a] Signed integers only.

More About

Finding the Delay Before Calculating an Error Rate

A typical use of this block is to determine the correct **Receive delay** parameter in the Error Rate Calculation block. This is illustrated in “Use the Find Delay Block”. In that example, the modulation/ demodulation operation introduces a computational delay into the received signal and the Find Delay block determines that the delay is 6 samples. This value of 6 becomes a parameter in the Error Rate Calculation block, which computes the bit error rate of the system.

Another example of this usage is in “Delays”.

Finding the Delay to Help Align Words

Another typical use of this block is to determine how to align the boundaries of frames with the boundaries of codewords or other types of data blocks. “Delays” describes when such alignment is necessary and also illustrates, in the “Aligning Words of a Block Code” discussion, how to use the Find Delay block to solve the problem.

Tips for Using the Block Effectively

- Set **Correlation window length** sufficiently large so that the computed delay eventually stabilizes at a constant value. When this occurs, the signal from the optional `chg` output port stabilizes at the constant value of zero. If the computed delay is not constant, you should increase **Correlation window length**. If the increased value of **Correlation window length** exceeds the duration of the simulation, then you should also increase the duration of the simulation accordingly. If you can roughly estimate the delay, then the **Correlation window length** will produce a stable delay estimate at four times that value.
- If the cross-correlation between the two signals is broad, then the **Correlation window length** value should be much larger than the expected delay, or else the algorithm might stabilize at an incorrect value. For example, a CPM signal has a broad autocorrelation, so it has a broad cross-correlation with a delayed version of itself. In this case, the **Correlation window length** value should be much larger than the expected delay.
- If the block calculates a delay that is greater than 75 percent of the **Correlation window length**, the signal `sRef` is probably delayed relative to the signal `sDel`. In this case, you should switch the signal lines leading into the two input ports.
- You can make the Find Delay block stop updating the delay after it computes the same delay value for a specified number of samples. To do so, select **Disable recurring updates**, and enter a positive integer in the **Number of constant delay outputs to disable updates** field. For example, if you set **Number of constant delay outputs to disable updates** to `20`, the block will stop recalculating and updating the delay after it calculates the same value `20` times in succession. Disabling recurring updates causes the simulation to run faster after the target number of constant delays occurs.

Algorithms

The Find Delay block finds the delay by calculating the cross-correlations of the first signal with time-shifted versions of the second signal, and then finding the index at which the cross-correlation is maximized.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Error Rate Calculation

Functions

`finddelay`

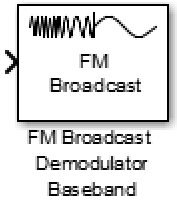
Topics

“Use the Find Delay Block”

Introduced before R2006a

FM Broadcast Demodulator Baseband

Demodulate using broadcast FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Broadcast Demodulator Baseband block demodulates a complex baseband FM signal by using the conjugate delay method, and filters the signal by using a de-emphasis filter. To demodulate stereo audio using 38 kHz, enable stereo demodulation. To demodulate RBDS signals from the 57 kHz band, enable RBDS demodulation.

Parameters

Sample rate (Hz)

Specify the input signal sample rate as a positive real scalar.

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe.

De-emphasis filter time constant (s)

Specify the de-emphasis lowpass filter time constant in seconds as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe.

Output audio sample rate (Hz)

Specify the output audio sample rate as a positive real scalar.

Play audio device

Select this check box to play sound from a default audio device.

Buffer size (samples)

Specify the buffer size the block uses to communicate with an audio device as a positive integer scalar. This parameter is available only when the **Play audio device** check box is selected.

Stereo audio

Select this check box to enable demodulation of a stereo audio signal. If not selected, the audio signal is assumed to be monophonic.

RBDS demodulation

Select this check box to demodulate the RBDS signal from the input complex baseband FM signal. By default, this check box is not selected.

Number of samples per RBDS symbol

Specify the number of samples of the RBDS output as a positive integer. The RBDS sample rate is given by **Number of samples per RBDS symbol** × `1187.5` Hz. According to the RBDS standard, the sample rate of each bit is 1187.5 Hz.

This parameter appears when you select the **RBDS demodulation** check box.

The default is 10.

RBDS Costas loop

Specify whether a Costas loop is used to recover the phase of the RBDS signal. Select this check box for radio stations that do not lock the `57` kHz RBDS signal in phase with the third harmonic of the `19` kHz pilot tone.

This parameter appears when you select the **RBDS demodulation** check box.

By default, this check box is not selected.

Simulate using

Select the type of simulation to run.

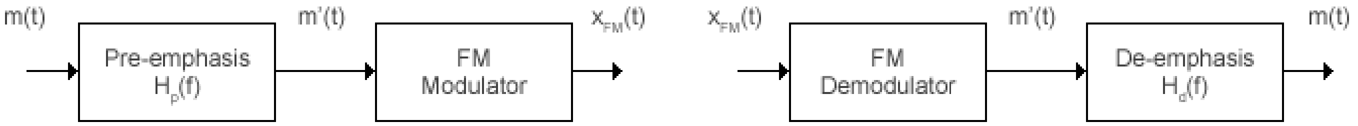
- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

The FM Broadcast demodulator includes the functionality of the baseband FM demodulator, de-emphasis filtering, and the ability to receive stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMDemodulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



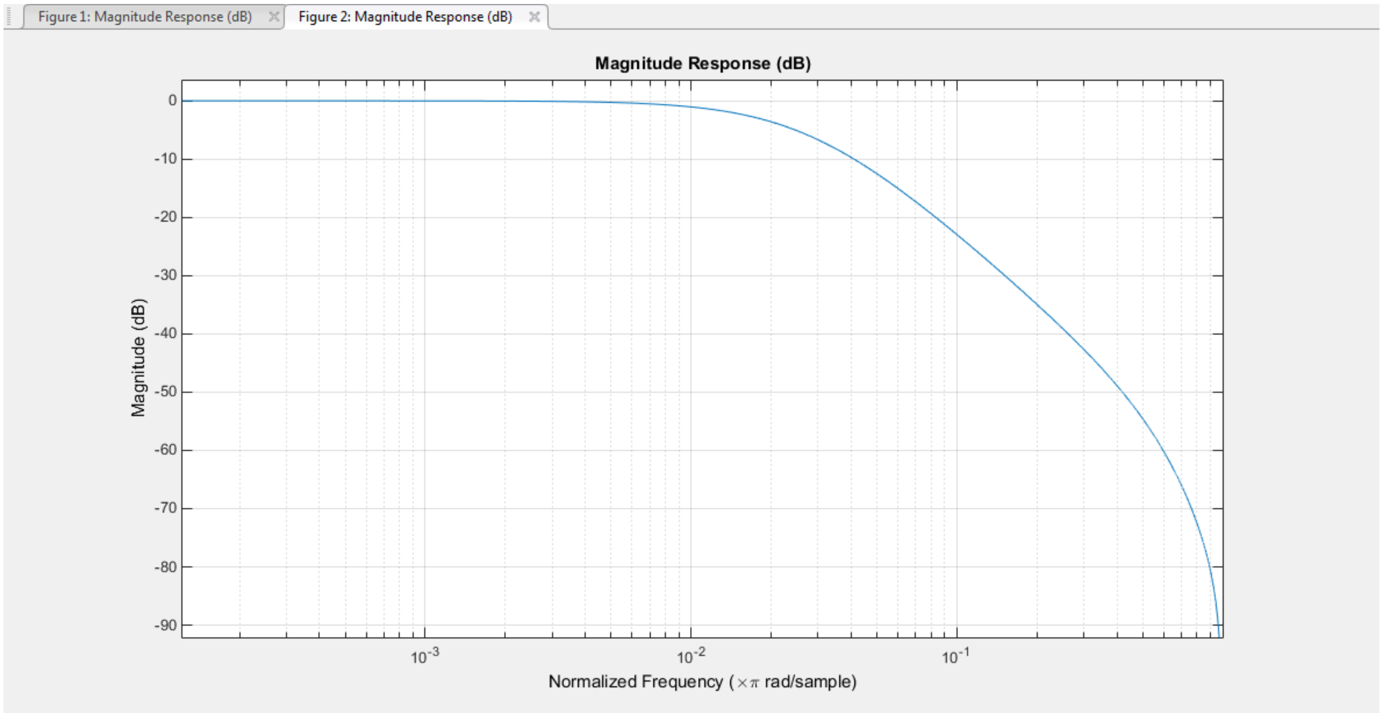
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s\,,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s}\;.$$

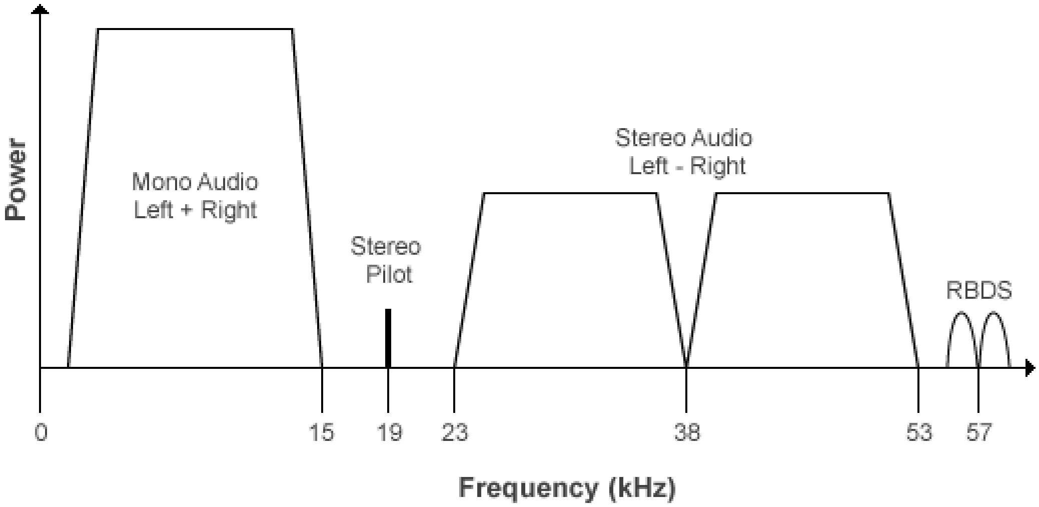
For an audio sample rate of 44.1 kHz, the de-emphasis filter has the following response.



Stereo and RDS/RBDS FM — Multiplex Signal

The FM broadcast demodulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals.

Here is the spectrum of the multiplex baseband signal, *m(t)*.



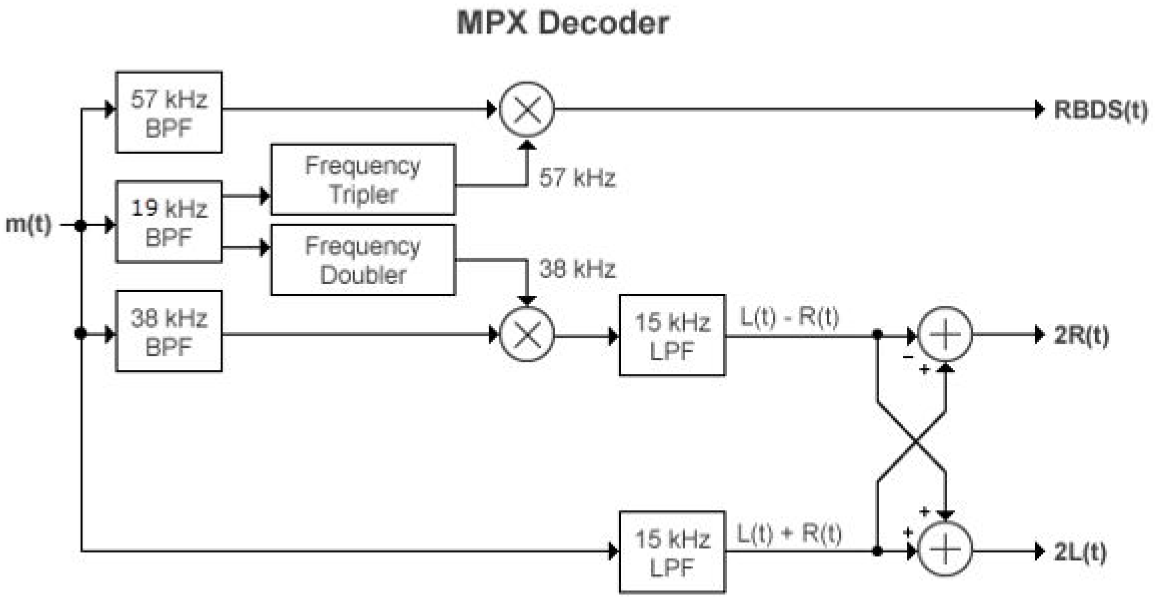
m(t) is given by

$$m(t) = C_0[L(t) + R(t)] + C_1\cos(2\pi \times 19kHz \times t) + C_0[L(t) - R(t)]\cos(2\pi \times 38kHz \times t) + C_2RBDS(t)\cos(2\pi \times 57kHz \times t)\,,$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

The demodulator applies *m(t)* to three bandpass filters with center frequencies at 19, 38, and 57 kHz, and to a lowpass filter with a 3-dB cutoff frequency of 15 kHz. The 19 kHz bandpass filter extracts the pilot tone from the modulated signal. The recovered pilot tone is doubled and tripled in frequency to produce the 38 kHz and 57 kHz signals, which demodulate the (*L* – *R*) and RDS/RBDS signals, respectively. To generate a scaled version of the left and right channels that produce the stereo sound, the (*L* + *R*) and (*L* – *R*) signals are added and subtracted. The RDS/RBDS signal is recovered by mixing with the 57 kHz signal.

Here is the block diagram of the FM broadcast demodulator.

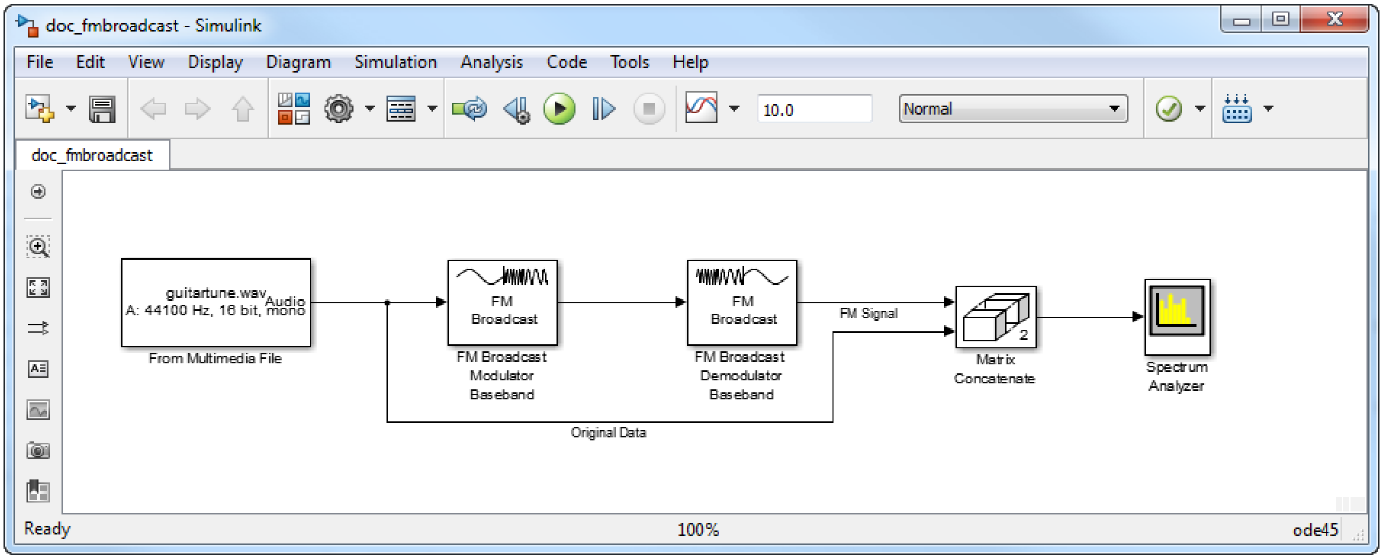


Examples

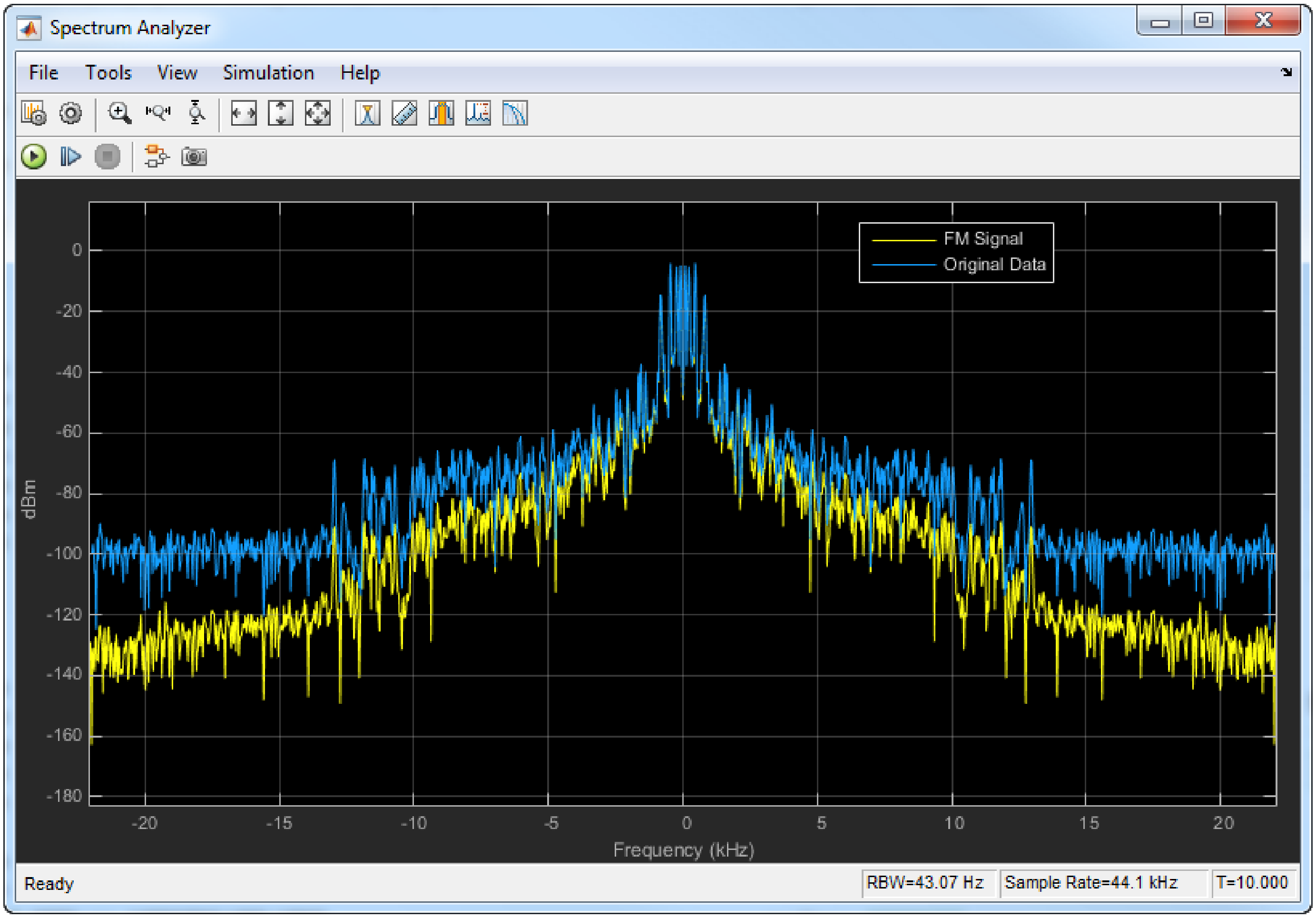
Modulate and Demodulate an Audio Signal

Load an audio input file, modulate and demodulate using the FM broadcast blocks. Compare the input signal spectrum with the demodulated signal spectrum.

Open the doc_fmbroadcast model.


doc_fmbroadcast - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Normal
doc_fmbroadcast
guitartune.wav
A: 44100 Hz, 16 bit, mono
Audio
From Multimedia File
FM Broadcast
FM Broadcast Modulator Baseband
FM Broadcast
FM Broadcast Demodulator Baseband
FM Signal
Original Data
2
Matrix Concatenate
Spectrum Analyzer
Ready
100%
ode45

Run the model. The spectrum of the baseband FM signal is attenuated at the higher frequencies relative to the original waveform.



Experiment with the model by changing the **Frequency deviation (Hz)** and the **Pre-emphasis filter time constant (s)** parameters on the modulator and demodulator and observe the impact on the FM signal spectrum.

Limitations

The input length must be an integer multiple of the audio decimation factor. If the **RBDS demodulation** check box is selected, the input length in addition must be an integer multiple of the RBDS decimation factor.

Supported Data Types

Port	Supported Data Types
Signal Input	• Double-precision floating point • Single-precision floating point
Signal Output	• Double-precision floating point • Single-precision floating point

References

[1] Chakrabarti, I. H., and Hatai, I. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems*. New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. “Frequency Modulation (FM) Tutorial”. FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

FM Broadcast Modulator Baseband

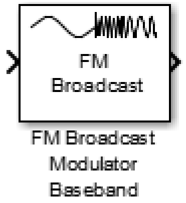
Objects

`comm.FMBroadcastDemodulator` | `comm.FMDemodulator` | `comm.RBDSWaveformGenerator`

Introduced in R2015a

FM Broadcast Modulator Baseband

Modulate using broadcast FM method



Library

Modulation > Analog Baseband Modulation

Description

The FM Broadcast Modulator Baseband block pre-emphasizes an audio signal and modulates it onto a baseband FM signal. If you select the **Stereo audio** check box, the block modulates the stereo audio (*L*–*R*) at the 38 kHz band, in addition to the baseband (*L*+*R*). If you select the **RBDS modulation** check box, the block also modulates a baseband RBDS signal at 57 kHz. For more details, see "Algorithms" on page 5-283.

Parameters

Sample rate (Hz)

Specify the output signal sample rate as a positive real scalar.

Frequency deviation (Hz)

Specify the frequency deviation of the modulator in Hz as a positive real scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth. FM broadcast standards specify a value of 75 kHz in the United States and 50 kHz in Europe.

Pre-emphasis filter time constant (s)

Specify the pre-emphasis highpass filter time constant as a positive real scalar. FM broadcast standards specify a value of 75 μs in the United States and 50 μs in Europe.

Sample rate of audio input signal (Hz)

Specify the input audio sample rate as a positive real scalar.

Stereo audio

Select this check box if the input signal is a stereophonic audio signal.

RBDS modulation

Select this check box to modulate a baseband RBDS signal at 57 kHz. By default, this check box is not selected.

Oversampling factor of RBDS input

Specify the number of samples per RBDS symbol as a positive integer. The RBDS sample rate is given by **Oversampling factor of RBDS input** × `1187.5` Hz. According to the RBDS standard, the sample rate of each bit is 1187.5 Hz.

This parameter appears when you select the **RBDS modulation** check box.

The default is 10.

Simulate using

Select the type of simulation to run.

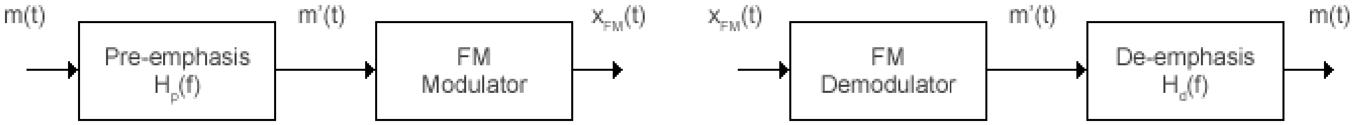
- `Code generation`. Simulate model using generate C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.
- `Interpreted execution`. Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.

Algorithms

The FM Broadcast modulator includes the functionality of the baseband FM modulator, pre-emphasis filtering, and the ability to transmit stereophonic signals. The algorithms which govern basic FM modulation and demodulation are covered in `comm.FMModulator`.

Filtering

FM amplifies high-frequency noise and degrades the overall signal-to-noise ratio. To compensate, FM broadcasters insert a pre-emphasis filter prior to FM modulation to amplify the high-frequency content. The FM receiver has a reciprocal de-emphasis filter after the FM demodulator to attenuate high-frequency noise and restore a flat signal spectrum.



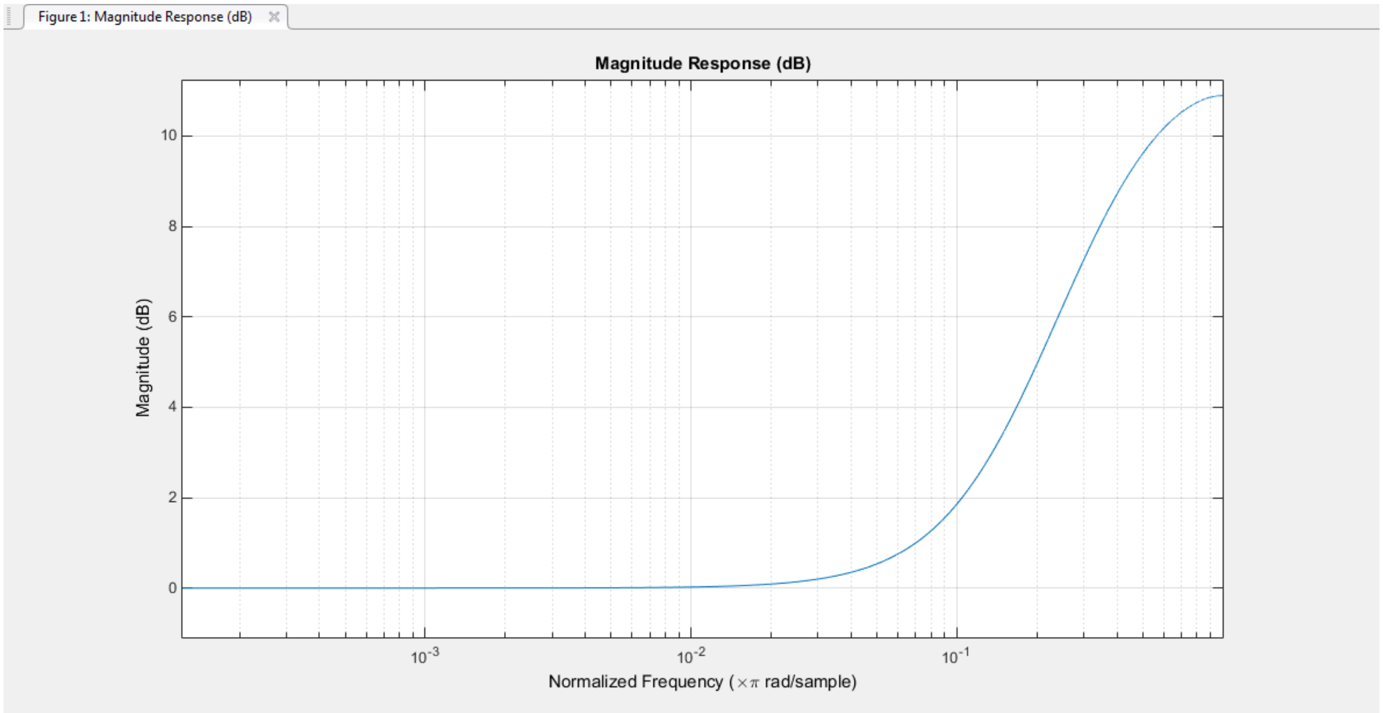
The pre-emphasis filter has a highpass characteristic transfer function given by

$$H_p(f) = 1 + j2\pi f\tau_s\,,$$

where τ_s is the filter time constant. The time constant is 50 µs in Europe and 75 µs in the United States. Similarly, the transfer function for the lowpass de-emphasis filter is given by

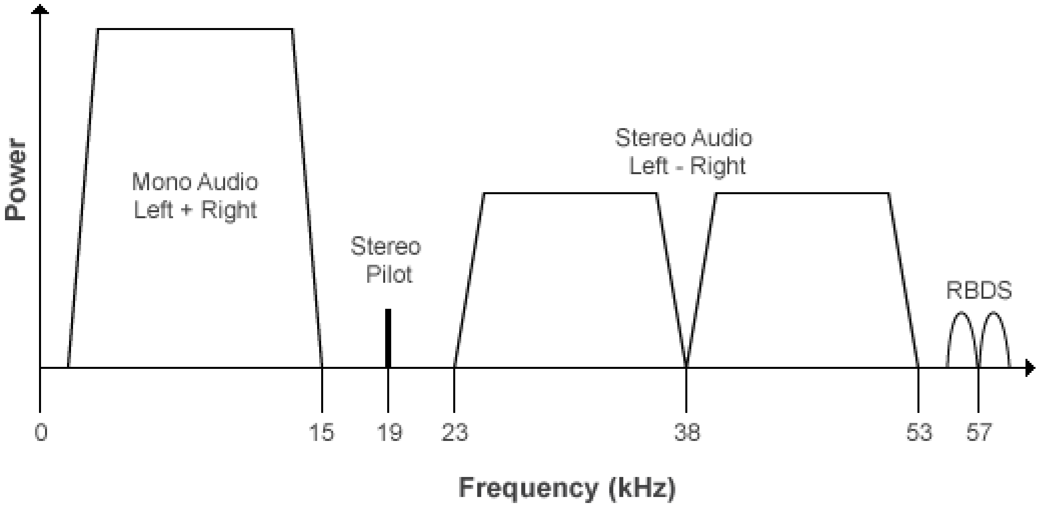
$$H_d(f) = \frac{1}{1 + j2\pi f\tau_s}\ .$$

Irrespective of the audio sampling rate, the signal is converted to a 152 kHz output sampling rate. For an audio sample rate of 44.1 kHz, the pre-emphasis filter has the following response.

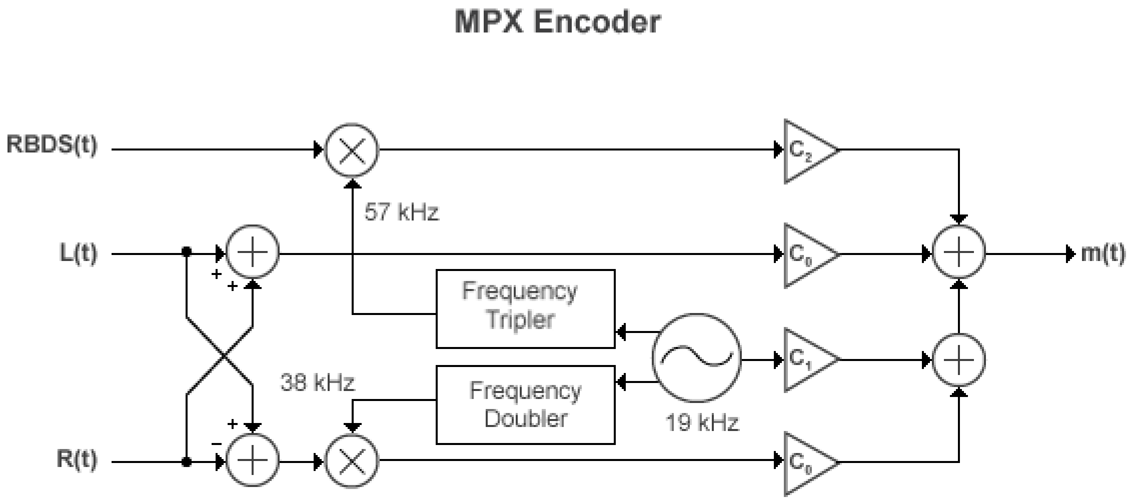


Stereo and RDS/RBDS FM – Multiplex Signal

The FM broadcast modulator supports stereophonic and monophonic operations. To support stereo transmission, the left (L) and right (R) channel information (L+R) is assigned to the mono portion of the spectrum (0 to 15 kHz). The (L-R) information is amplitude modulated onto the 23 to 53 kHz region of the baseband spectrum using a 38 kHz subcarrier signal. A pilot tone at 19 kHz in the multiplexed signal enables the FM receiver to coherently demodulate the stereo and RDS/RBDS signals. Here is the spectrum of the multiplex baseband signal.



Here is the block diagram of the FM broadcast modulator, which is used to generate the multiplex baseband signal. *L(t)* and *R(t)* denote the time-domain waveforms from the left and right channels. *RBDS(t)* denotes the time-domain waveform of the RDS/RBDS signal.



The multiplex message signal, *m(t)* is given by

$$m(t) = C_0[L(t) + R(t)] + C_1\cos(2\pi \times 19kHz \times t) + C_0[L(t) - R(t)]\cos(2\pi \times 38kHz \times t) + C_2RBDS(t)\cos(2\pi \times 57kHz \times t)\,,$$

where C_0, C_1, and C_2 are gains. To generate the appropriate modulation level, these gains scale the amplitudes of the (*L(t)*±*R(t)*) signals, the 19 kHz pilot tone, and the RDS/RBDS subcarrier, respectively.

Limitations

- If you select the **RBDS modulation** check box, both the audio and RBDS inputs must satisfy the following equation:

$$\frac{audioLength}{audioSampleRate} = \frac{RBDSLength}{RBDSSampleRate}$$

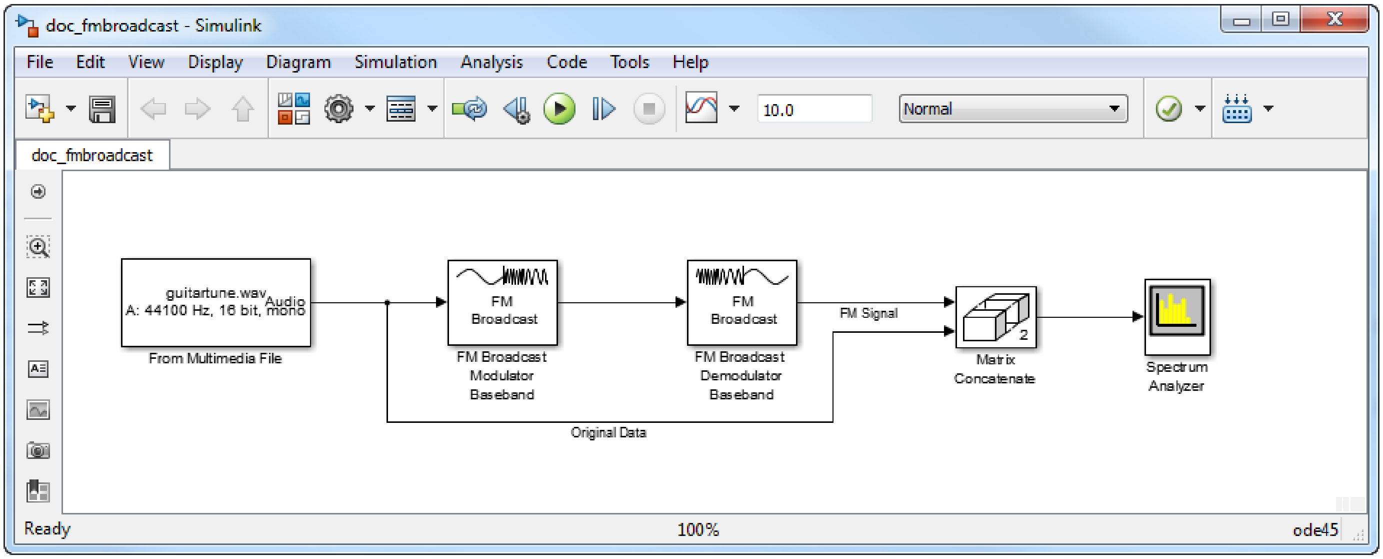
- The input length of the audio signal must be an integer multiple of the audio decimation factor. The input length of the RBDS signal must be an integer multiple of the RBDS decimation factor.

Examples

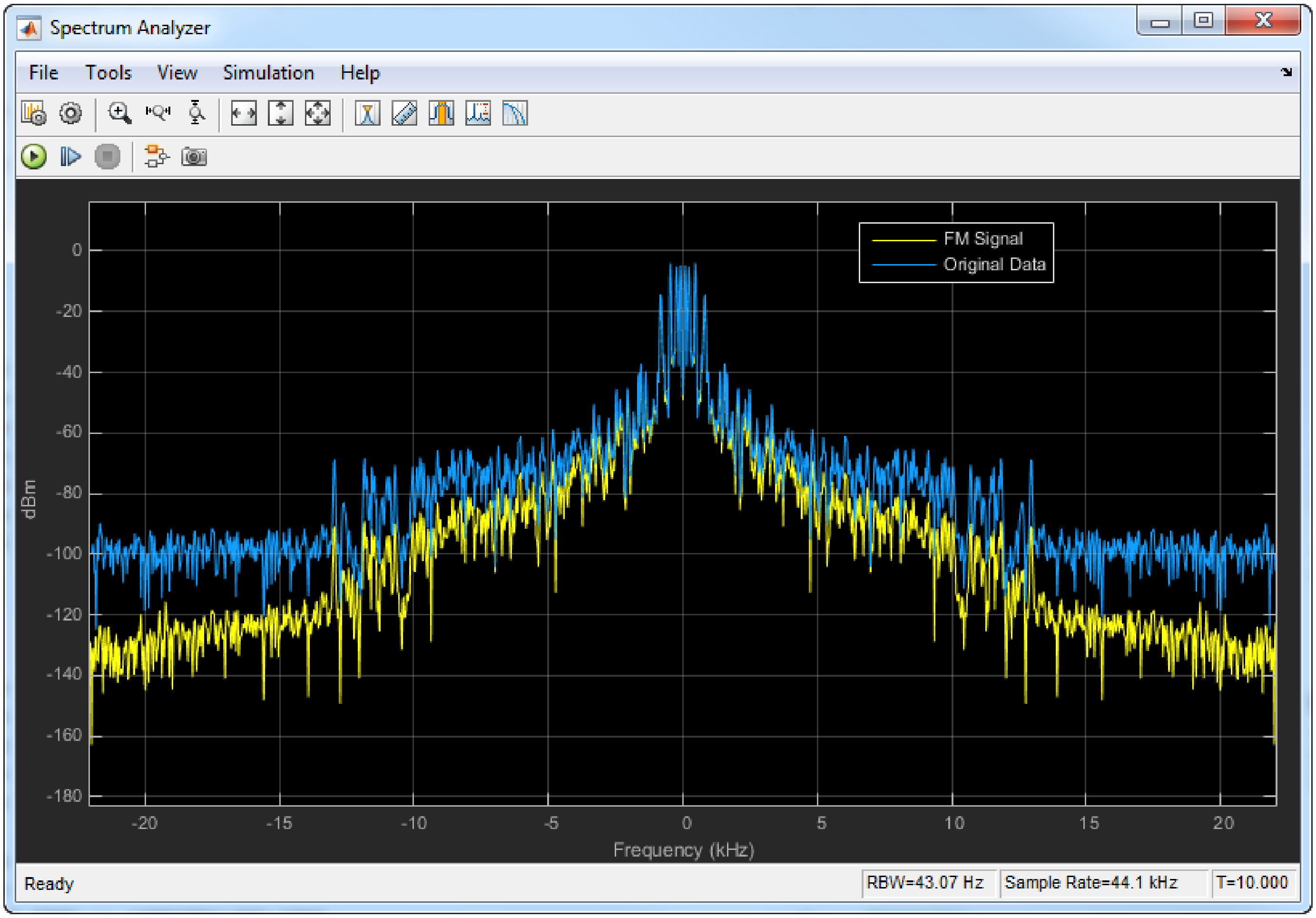
Modulate and Demodulate an Audio Signal

Load an audio input file, modulate and demodulate using the FM broadcast blocks. Compare the input signal spectrum with the demodulated signal spectrum.

Open the doc_fmbroadcast model.


doc_fmbroadcast - Simulink
File Edit View Display Diagram Simulation Analysis Code Tools Help
10.0
Normal
doc_fmbroadcast
guitartune.wav
A: 44100 Hz, 16 bit, mono
Audio
From Multimedia File
FM Broadcast
FM Broadcast Modulator Baseband
FM Broadcast
FM Broadcast Demodulator Baseband
FM Signal
Original Data
2
Matrix Concatenate
Spectrum Analyzer
Ready
100%
ode45

Run the model. The spectrum of the baseband FM signal is attenuated at the higher frequencies relative to the original waveform.



Experiment with the model by changing the **Frequency deviation (Hz)** and the **Pre-emphasis filter time constant (s)** parameters on the modulator and demodulator and observe the impact on the FM signal spectrum.

Supported Data Types

Port	Supported Data Types
Signal Input	• Double-precision floating point • Single-precision floating point
Signal Output	• Double-precision floating point • Single-precision floating point

References

[1] Chakrabarti, I. H., and Hatai, I. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

[3] Der, Lawrence. "Frequency Modulation (FM) Tutorial". FM Tutorial. Silicon Laboratories Inc., pp. 4–8.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

FM Broadcast Demodulator Baseband

Objects

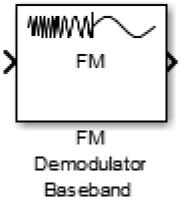
`comm.FMBroadcastModulator` | `comm.FMModulator` | `comm.RBDSWaveformGenerator`

Introduced in R2015a

FM Demodulator Baseband

Demodulate using FM method

Library: Communications Toolbox / Modulation / Analog Baseband Modulation



Description

The FM Demodulator Baseband block demodulates a complex input signal and returns a real output signal.

Ports

Input

In — Input data signal
scalar | vector | matrix

Input signal, specified as a real scalar, vector, or matrix.

Data Types: double | single

Output

Out — Output data signal
scalar | vector | matrix

Output signal, returned as a real scalar, vector, or matrix. The data at this port has the same data type and size as the input signal.

Data Types: double | single

Parameters

Frequency deviation (Hz) — Frequency deviation of demodulator
75e3 (default) | positive scalar

Frequency deviation of the demodulator, in Hz, specified as a positive scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth.

Simulate using — Type of simulation to run
Code generation (default) | Interpreted execution

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Algorithms

Represent a frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where *A* is the carrier amplitude, f_c is the carrier frequency, *x(τ)* is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming |*x(t)*| ≤ 1.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$\begin{aligned} y_s(t) = Y(t)e^{-j2\pi f_c t} &= \frac{A}{2}\left[e^{j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t} \\ &= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t - j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right]. \end{aligned}$$

Removing the component at -2f_c from $y_s(t)$ leaves the baseband signal representation, *y(t)*, which is expressed as

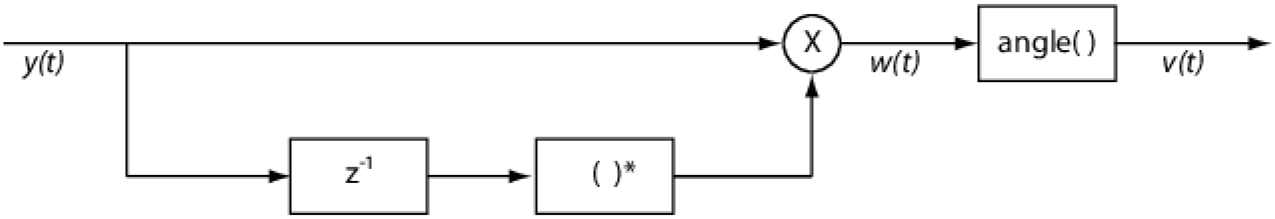
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)},$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, *ϕ(t)*.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself,

$$w(t) = \frac{A^2}{4} e^{j\phi(t)} e^{-j\phi(t-T)} = \frac{A^2}{4} e^{j[\phi(t) - \phi(t-T)]},$$

where T is the sample period. In discrete terms, w_n=$w(nT)$, and

$$w_n = \frac{A^2}{4} e^{j[\phi_n - \phi_{n-1}]},$$

$$v_n = \phi_n - \phi_{n-1} .$$

The signal v_n is the approximate derivative of ϕ_n, such that $v_n \approx x_n$.

References

[1] Chakrabarti, I. H., and I, Hatai. "A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation." *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

FM Modulator Baseband

Objects

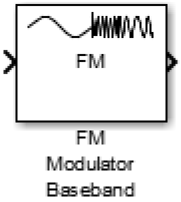
`comm.FMDemodulator`

Introduced in R2015a

FM Modulator Baseband

Modulate using FM method

Library: Communications Toolbox / Modulation / Analog Baseband Modulation



Description

The FM Modulator Baseband block applies frequency modulation to a real input signal and returns a complex output signal.

Ports

Input

`In` — Input data signal
`scalar` | `vector` | `matrix`

Input signal, specified as a real scalar, vector, or matrix.

Data Types: `double` | `single`

Output

`Out` — Output data signal
`scalar` | `vector` | `matrix`

Output signal, returned as a real scalar, vector, or matrix. The data at this port has the same data type and size as the input signal.

Data Types: `double` | `single`

Parameters

`Frequency deviation (Hz)` — Frequency deviation of modulator
`75e3` (default) | `positive scalar`

Frequency deviation of the modulator, in Hz, specified as a positive scalar. The system bandwidth is equal to twice the sum of the frequency deviation and the message bandwidth.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Algorithms

Represent a frequency modulated passband signal, *Y(t)*, as

$$Y(t) = A\cos\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right),$$

where *A* is the carrier amplitude, f_c is the carrier frequency, *x(τ)* is the baseband input signal, and f_Δ is the frequency deviation in Hz. The frequency deviation is the maximum shift from f_c in one direction, assuming |*x(t)*| ≤ 1.

A baseband FM signal can be derived from the passband representation by downconverting it by f_c such that

$$\begin{aligned} y_s(t) = Y(t)e^{-j2\pi f_c t} &= \frac{A}{2}\left[e^{j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)} + e^{-j\left(2\pi f_c t + 2\pi f_\Delta \int_0^t x(\tau)d\tau\right)}\right]e^{-j2\pi f_c t} \\ &= \frac{A}{2}\left[e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau} + e^{-j4\pi f_c t - j2\pi f_\Delta \int_0^t x(\tau)d\tau}\right]. \end{aligned}$$

Removing the component at -2f_c from $y_s(t)$ leaves the baseband signal representation, *y(t)*, which is expressed as

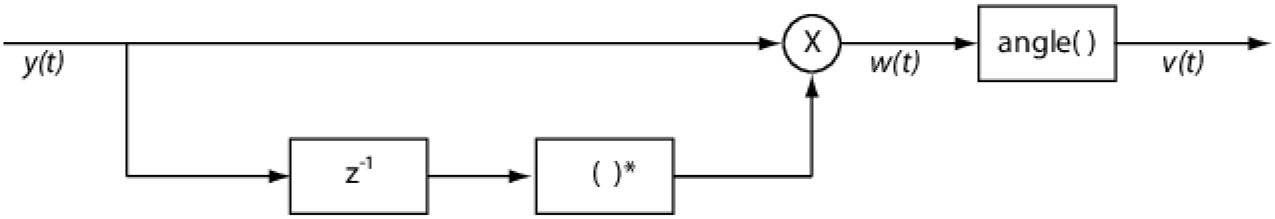
$$y(t) = \frac{A}{2}e^{j2\pi f_\Delta \int_0^t x(\tau)d\tau}.$$

The expression for *y(t)* is rewritten as

$$y(t) = \frac{A}{2}e^{j\phi(t)},$$

where $\phi(t) = 2\pi f_\Delta \int_0^t x(\tau)d\tau$, which implies that the input signal is a scaled version of the derivative of the phase, *ϕ(t)*.

A baseband delay demodulator is used to recover the input signal from *y(t)*.



A delayed and conjugated copy of the received signal is subtracted from the signal itself,

$$w(t) = \frac{A^2}{4}e^{j\phi(t)}e^{-j\phi(t-T)} = \frac{A^2}{4}e^{j[\phi(t)-\phi(t-T)]},$$

where T is the sample period. In discrete terms, w_n=$w(nT)$, and

$$w_n = \frac{A^2}{4}e^{j[\phi_n - \phi_{n-1}]},$$

$$v_n = \phi_n - \phi_{n-1}\ .$$

The signal v_n is the approximate derivative of ϕ_n, such that $v_n \approx x_n$.

References

[1] Chakrabarti, I. H., and I, Hatai. “A New High-Performance Digital FM Modulator and Demodulator for Software-Defined Radio and Its FPGA Implementation.” *International Journal of Reconfigurable Computing*. Vol. 2011, No. 10.1155/2011, 2011, p. 10.

[2] Taub, Herbert, and Donald L. Schilling. *Principles of Communication Systems.* New York: McGraw-Hill, 1971, pp. 142–155.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

FM Demodulator Baseband

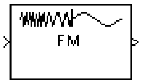
Objects

`comm.FMModulator`

Introduced in R2015a

FM Demodulator Passband

Demodulate FM-modulated data



Library

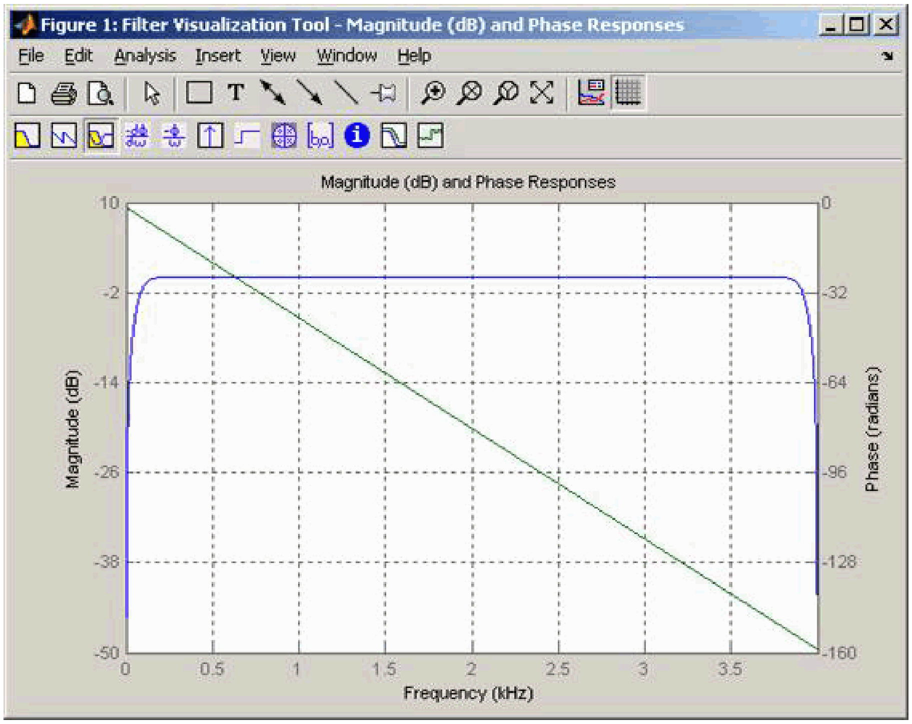
Analog Passband Modulation, in Modulation

Description

The FM Demodulator Passband block demodulates a signal that was modulated using frequency modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

For best results, use a carrier frequency which is estimated to be larger than 10% of the reciprocal of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.



Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the reciprocal of your input signal's sample time (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier in radians.

Frequency deviation (Hz)

The frequency deviation of the carrier frequency in Hertz. Sometimes it is referred to as the "variation" in the frequency.

Hilbert transform filter order

The length of the FIR filter used to compute the Hilbert transform.

Pair Block

FM Modulator Passband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

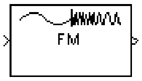
Blocks

FM Modulator Passband

Introduced before R2006a

FM Modulator Passband

Modulate using frequency modulation



Library

Analog Passband Modulation, in Modulation

Description

The FM Modulator Passband block modulates using frequency modulation. The output is a passband representation of the modulated signal. The output signal's frequency varies with the input signal's amplitude. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$\cos\left(2\pi f_c t + 2\pi K_c \int_0^t u(\tau) d\tau + \theta\right)$$

where:

- f_c represents the **Carrier frequency** parameter.
- θ represents the **Initial phase** parameter.
- K_c represents the **Frequency deviation** parameter.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal.

By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier in radians.

Frequency deviation (Hz)

The frequency deviation of the carrier frequency in Hertz. Sometimes it is referred to as the "variation" in the frequency.

Pair Block

FM Demodulator Passband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

FM Demodulator Passband

Introduced before R2006a

Free Space Path Loss

Apply free space path loss to complex signal
Library: Communications Toolbox / RF Impairments

Description

The Free Space Path Loss block applies a free space path loss to a complex signal. The block simulates the loss of signal power due to the distance between the transmitter and receiver. The **Mode** parameter indicates whether you specify the loss in decibels or as a computation that is based on distance and the RF signal frequency.

Ports

Input

`In1` — Complex signal
scalar | column vector

Complex signal, specified as a scalar or column vector.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Out1` — Output signal
scalar | vector

Output signal, returned as a scalar or column vector. This output is the same dimension and data type as the input signal.

Parameters

`Mode` — Loss calculation mode
`Decibels` (default) | `Distance and Frequency`

Loss calculation mode, specified as one of these options.

- `Decibels` — The loss is specified using the **Loss (dB)** parameter.
- `Distance and Frequency` — The loss is computed using the **Distance (km)** and **Carrier frequency (MHz)** parameters.

`Loss (dB)` — Power loss
`10` (default) | scalar

Power loss in decibels, specified as a scalar. The decibel amount shown on the block icon is rounded for display purposes only.

Dependencies

To enable this parameter, set the **Mode** parameter to `Decibels`.

`Distance (km)` — Distance between transmitter and receiver

`0.100` (default) | scalar

Distance between the transmitter and receiver in kilometers, specified as a scalar.

Dependencies

To enable this parameter, set the **Mode** parameter to `Distance and Frequency`.

`Carrier frequency (MHz)` — Carrier frequency

`1920` (default) | scalar

Carrier frequency in megahertz, specified as a scalar.

Dependencies

To enable this parameter, set the **Mode** parameter to `Distance and Frequency`.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Algorithms

The free-space path loss, L, in decibels is:

$L = 20\log_{10}(4\pi R/\lambda)$.

This formula assumes that the target is in the far-field of the transmitting element or array. In the near-field, the free-space path loss formula is not valid and can result in a loss smaller than 0 dB, equivalent to a signal gain. For this reason, the loss is set to 0 dB for range values $R \leq \lambda/4\pi$.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Functions

`fspl`

Blocks

I/Q Imbalance | Memoryless Nonlinearity | Phase Noise | Receiver Thermal Noise

Introduced before R2006a

Gaussian Noise Generator

(To be removed) Generate Gaussian distributed noise with given mean and variance values

Note Gaussian Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

Library

Noise Generators sublibrary of Comm Sources

Description

The Gaussian Noise Generator block generates discrete-time white Gaussian noise. You must specify the **Initial seed** vector in the simulation.

The **Mean Value** and the **Variance** can be either scalars or vectors. If either of these is a scalar, then the block applies the same value to each element of a sample-based output or each column of a frame-based output. Individual elements or columns, respectively, are uncorrelated with each other.

When the **Variance** is a vector, its length must be the same as that of the **Initial seed** vector. In this case, the covariance matrix is a diagonal matrix whose diagonal elements come from the **Variance** vector. Since the off-diagonal elements are zero, the output Gaussian random variables are uncorrelated.

When the **Variance** is a square matrix, it represents the covariance matrix. Its off-diagonal elements are the correlations between pairs of output Gaussian random variables. In this case, the **Variance** matrix must be positive definite, and it must be `N`-by-`N`, where `N` is the length of the **Initial seed**.

The probability density function of *n*-dimensional Gaussian noise is

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

where *x* is a length-*n* vector, *K* is the *n*-by-*n* covariance matrix, μ is the mean value vector, and the superscript *T* indicates matrix transpose.

Initial Seed

The **Initial seed** parameter initializes the random number generator that the Gaussian Noise Generator block uses to add noise to the input signal. When multiple blocks in a model have the **Initial seed** parameter, you can choose different initial seeds for each block to ensure different random streams are used in each block. Set **Initial seed** to an integer value for repeatable results or use the `randi` function to randomize your results.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. For additional information, see “Sources and Sinks”.

If the **Initial seed** parameter is a vector, then its length becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. In this case, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal. If the **Initial seed** parameter is a scalar but either the **Mean value** or **Variance** parameter is a vector, then the vector length determines the output attributes mentioned above.

Parameters

Mean value

The mean value of the random variable output.

Variance

The covariance among the output random variables.

Initial seed

The initial seed value for the random number generator.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame

The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D

If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type

The output can be set to `double` or `single` data types.

See Also

Blocks

AWGN Channel | MATLAB Function | Random Source

Functions

`isprime` | `randi` | `randn` | `rng`

Introduced before R2006a

General Block Deinterleaver

Restore ordering of symbols in input vector

Library

Block sublibrary of Interleaving

Description

The General Block Deinterleaver block rearranges the elements of its input vector without repeating or omitting any elements. If the input contains *N* elements, then the **Permutation vector** parameter is a column vector of length *N*. The column vector indicates the indices, in order, of the output elements that came from the input vector. That is, for each integer *k* between 1 and *N*,

Output(**Permutation vector**(*k*)) = Input(*k*)

The **Permutation vector** parameter must contain unique integers between 1 and *N*.

Both the input and the **Permutation vector** parameter must be column vector signals.

This block can output sequences that vary in length during simulation. For more information about sequences that vary in length, or variable-size signals, see “Variable-Size Signal Basics” (Simulink).

This block accept the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

To use this block as an inverse of the General Block Interleaver block, use the same **Permutation vector** parameter in both blocks. In that case, the two blocks are inverses in the sense that applying the General Block Interleaver block followed by the General Block Deinterleaver block leaves data unchanged.

Parameters

Permutation vector source

A selection that specifies the source of the permutation vector. The source can be either `Dialog` or `Input port`. The default value is `Dialog`.

Permutation vector

A vector of length N that lists the indices of the output elements that came from the input vector. This parameter is available only when **Permutation vector source** is set to `Dialog`.

Examples

This example reverses the operation in the example on the General Block Interleaver block reference page. If you set **Permutation vector** to `[4,1,3,2]'` and you set the General Block Deinterleaver block input to `[1;40;59;32]`, then the output of the General Block Deinterleaver block is `[40;32;59;1]`.

Pair Block

General Block Interleaver

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Block Interleaver

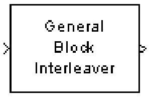
Functions

`perms`

Introduced before R2006a

General Block Interleaver

Reorder symbols in input vector



Library

Block sublibrary of Interleaving

Description

The General Block Interleaver block rearranges the elements of its input vector without repeating or omitting any elements. If the input contains *N* elements, then the **Permutation vector** parameter is a column vector of length *N*. The column vector indicates the indices, in order, of the input elements that form the length-*N* output vector; that is,

Output(*k*) = Input(**Permutation vector**(k))

for each integer *k* between 1 and *N*. The contents of **Permutation vector** must be integers between 1 and N, and must have no repetitions.

Both the input and the **Permutation vector** parameter must be column vector signals.

This block can output sequences that vary in length during simulation. For more information about sequences that vary in length, or variable-size signals, see “Variable-Size Signal Basics” (Simulink).

This block accept the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Permutation vector source

A selection that specifies the source of the permutation vector. The source can be either `Dialog` or `Input port`. The default value is `Dialog`.

Permutation vector

A vector of length N that lists the indices of the output elements that came from the input vector. This parameter is available only when **Permutation vector source** is set to `Dialog`.

Examples

If **Permutation vector** is `[4;1;3;2]` and the input vector is `[40;32;59;1]`, then the output vector is `[1;40;59;32]`. Notice that all of these vectors have the same length and that the vector **Permutation vector** is a permutation of the vector `[1:4]'`.

Pair Block

General Block Deinterleaver

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Block Deinterleaver

Functions

`perms`

Introduced before R2006a

General CRC Generator

Generate CRC code bits according to generator polynomial and append to input data frames

Library: Communications Toolbox / Error Detection and Correction / CRC

Description

The General CRC Generator block generates cyclic redundancy check (CRC) code bits for each input data frame and appends them to the frame. For more information, see “CRC Generator Operation” on page 5-309.

Ports

Input

`In` — Input signal
binary column vector

Input signal, specified as a binary column vector. The length of the input frame must be a multiple of the value of the **Checksums per frame** parameter.

Data Types: `double` | `Boolean`

Output

`Out` — Output codeword frame
binary column vector

Output codeword frame, returned as a binary column vector that inherits the data type of the input signal. The output contains the input data frames with the CRC bit sequences appended to them.

The length of the output frame is $m + k * r$, where m is the size of the input frame, k is the number of checksums per frame, and r is the degree of the generator polynomial.

Parameters

`Generator polynomial` — Generator polynomial
`'z^16 + z^12 + z^5 + 1'` (default) | polynomial character vector | binary row vector | integer row vector

Generator polynomial for the CRC algorithm, specified as one of the following:

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial x^3+ z^2+ 1.

- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, [3 2 0] represents the polynomial $z^3 + z^2 + 1$.

For more information, see "Character Representation of Polynomials".

Some commonly used generator polynomials include:

CRC method	Generator polynomial
CRC-32	'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'
CRC-24	'z^24 + z^23 + z^14 + z^12 + z^8 + 1'
CRC-16	'z^16 + z^15 + z^2 + 1'
Reversed CRC-16	'z^16 + z^14 + z + 1'
CRC-8	'z^8 + z^7 + z^6 + z^4 + z^2 + 1'
CRC-4	'z^4 + z^3 + z^2 + z + 1'

Example: 'z^7 + z^2 + 1', [1 0 0 0 0 1 0 1], and [7 2 0] represent the same polynomial, $p(z) = z^7 + z^2 + 1$.

Initial states — Initial states of internal shift register
0 (default) | 1 | binary row vector

Initial states of the internal shift register, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial.

Direct method — Use direct algorithm for CRC checksum calculations
off (default) | on

Select to use the direct algorithm for CRC checksum calculations. When cleared, the block uses the non-direct algorithm for CRC checksum calculations.

For more information on direct and non-direct algorithms, see "Error Detection and Correction".

Reflect input bytes — Reflect input bytes
off (default) | on

Select to flip the input data on a bytewise basis before entering the data into the shift register. When **Reflect input bytes** is selected, the input frame length divided by the value of the **Checksums per frame** parameter must be an integer and a multiple of 8. When **Reflect input bytes** is cleared, the block does not flip the input data.

Reflect checksums before final XOR — Reflect checksums before final XOR
off (default) | on

Select to flip the CRC checksums around their centers after the input data are completely through the shift register. When **Reflect checksums before final XOR** is cleared, the block does not flip the CRC checksums.

Final XOR — Final XOR
0 (default) | 1 | binary row vector

Final XOR, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. The XOR operation runs using the value of the **Final XOR** parameter the CRC checksum before appending the CRC to the input data. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial. A setting of `0` is equivalent to no XOR operation.

`Checksums per frame` — Number of checksums calculated for each frame
1 (default) | positive integer

Number of checksums calculated for each frame, specified as a positive integer.

Block Characteristics

Data Types	`Boolean` | `double`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

Cyclic Redundancy Check Coding

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

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

When the number of checksums per frame is greater than 1, the input data frame is divided into subframes, the rule is applied to each data subframe, and individual checksums are appended to each subframe. The subframe codewords are concatenated to output one frame.

For a discussion of the supported CRC algorithms, see “Cyclic Redundancy Check Codes”.

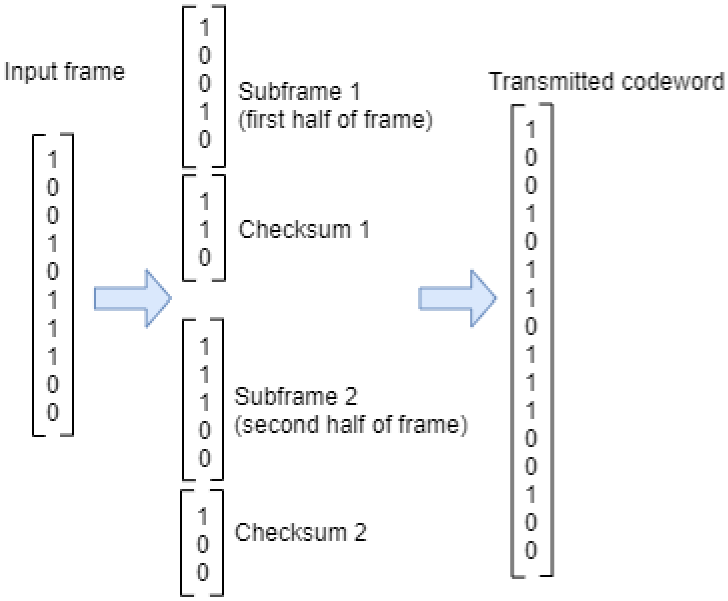
CRC Generator Operation

The CRC generator appends CRC checksums to the input frame according to the specified generator polynomial and number of checksums per frame.

For a specific initial state of the internal shift register and *k* checksums per input frame:

1. The input signal is divided into *k* subframes of equal size.
2. Each of the *k* subframes are prefixed with the initial states vector.
3. The CRC algorithm is applied to each subframe.
4. The resulting checksums are appended to the end of each subframe.
5. The subframes are concatenated and output as a column vector.

For the scenario shown here, a 10-bit frame is input, a third degree generator polynomial computes the CRC checksum, the initial state is 0, and the number of checksums per frame is 2.



The input frame is divided into two subframes of size 5 and checksums of size 3 are computed and appended to each subframe. The initial states are not shown, because an initial state of `[0]` does not affect the output of the CRC algorithm. The output transmitted codeword frame has the size 5 + 3 + 5 + 3 = 16.

References

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Objects

`comm.CRCGenerator`

Blocks

General CRC Generator HDL Optimized | General CRC Syndrome Detector

Topics

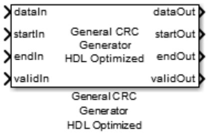
“Cyclic Redundancy Check Codes”

Introduced before R2006a

General CRC Generator HDL Optimized

Generate CRC code bits and append them to input data

Library: Communications Toolbox HDL Support / Error Detection and Correction / CRC
Communications Toolbox / Error Detection and Correction / CRC



Description

The General CRC Generator HDL Optimized block, which is similar to the General CRC Generator block, generates a cyclic redundancy check (CRC) checksum and appends it to the input message. The General CRC Generator HDL Optimized block processing is optimized for HDL code generation. Instead of processing an entire frame at once, the block accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame. To achieve higher throughput, the block accepts vector data up to the CRC length and implements a parallel architecture.

Ports

Input

`dataIn` — Input data
scalar | vector

Input data, specified as one of these options.

- Scalar – Specify an integer representing several bits. For this case, the block supports an unsigned integer (`uint8`, `uint16`, or `uint32`) or `fixdt(0,N,0)` data type.
- Vector – Specify a vector of binary values. For this case, the block supports a `double` or `Boolean` data type.

The data width must be less than or equal to the CRC length, and the CRC length must be divisible by the data width. For CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.

Example: The `uint8` vector input `[0 0 0 1 0 0 1 1]` is equivalent to 19.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fixed point` | `Boolean`

`startIn` — Start of input frame indicator
Boolean scalar

Start of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`endIn` — End of input frame indicator
Boolean scalar

End of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`validIn` — Valid input data indicator
Boolean scalar

Valid input data indicator, specified as a Boolean scalar.

This is a control signal that indicates if the data on the **dataIn** port is valid.

Data Types: `Boolean`

Output

`dataOut` — Output data
scalar | vector

Output data with appended checksum, returned as a scalar or vector. The output data type and size are the same as the input data.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`startOut` — Start of output frame indicator
scalar

Start of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`endOut` — End of output frame indicator
scalar

End of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`validOut` — Valid output data indicator
scalar

Valid output data indicator, returned as a Boolean scalar.

This port is a control signal that indicates if the data on the **dataOut** port is valid.

Data Types: `Boolean`

Parameters

`Polynomial` — Generator polynomial
`[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]` (default) | binary vector

Specify the generator polynomial as a binary vector with coefficients in descending order of powers. The vector length is equal to the degree of the polynomial plus 1.

`Initial state` — Initial conditions of shift register
`0` (default) | binary scalar | binary vector

Specify initial conditions of the internal shift register as a binary, double-precision, or single-precision scalar or vector. For vector inputs, the length of the initial state must be equal to the degree of the generator polynomial.

`Direct method` — Method of calculating checksum
`off` (default) | `on`

Specify the direct or indirect method for calculating the checksum.

- Select this parameter for the block to use the direct algorithm for CRC checksum calculations.
- Clear this parameter for the block to use the nondirect algorithm for CRC checksum calculations.

For more information about direct and nondirect algorithms, see "Cyclic Redundancy Check Codes".

`Reflect input` — Input byte order
`off` (default) | `on`

Specify the input byte order.

- Select this parameter for the block to flip each input byte before it enters the shift register.
- Clear this parameter for the block to pass the message data to the shift register unchanged.

The input data width must be a multiple of 8.

`Reflect CRC checksum` — Checksum byte order
`off` (default) | `on`

Specify the checksum byte order.

- Select this parameter for the block to flip each checksum byte before passing it to the final XOR stage.
- Clear this parameter for the block to pass the checksum byte to the final XOR stage unchanged.

The input data width must be a multiple of 8.

`Final XOR value` — Checksum
`0` (default) | binary scalar | binary vector

Specify the checksum as a binary, double-precision, or single-precision data type scalar or vector. The block performs XOR operation on the CRC checksum with this value before appending it to the input data.

If you specify a vector input, the vector length must be equal to the degree of the generator polynomial.

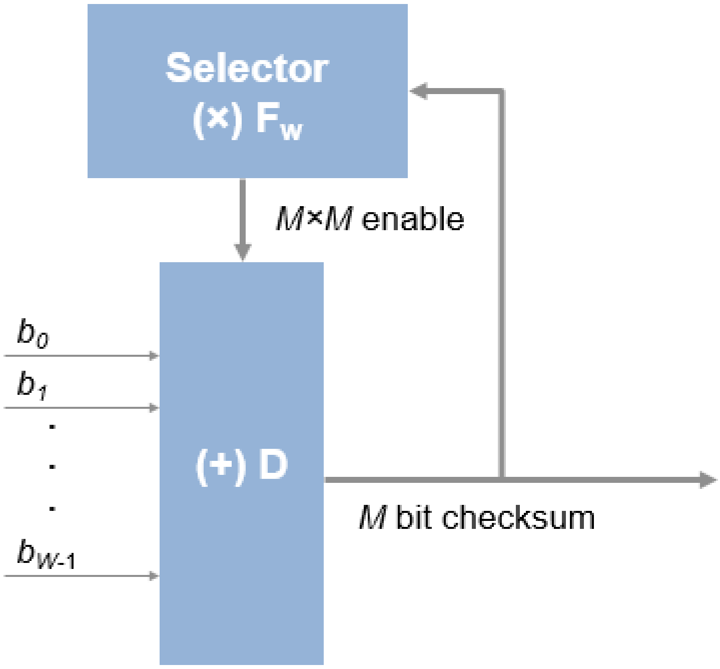
Algorithms

When you use a vector or integer input, the block implements a parallel CRC algorithm [1].

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates M bits of a CRC checksum for each W input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of M, the recursive checksum calculation for W bits in parallel is

$$X' = F_W(\times)X(+)D.$$

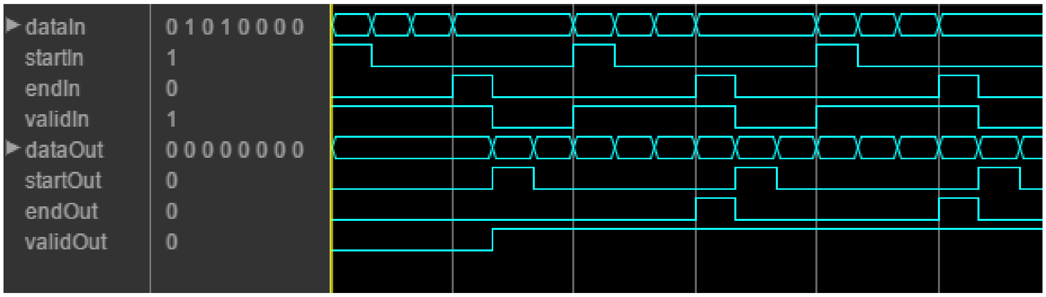
F_W is an M-by-M matrix that selects elements of the current state for the polynomial calculation with the new input bits. D is an M-element vector that provides the new input bits, ordered in relation to the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with an 8-bit binary vector input. To insert the checksum word, input frames must have enough space between the.

This waveform diagram shows continuous input data. The block also supports noncontinuous data. The output valid signal matches the input valid pattern.



```
▶ dataIn    0 1 0 1 0 0 0 0
  startIn   1
  endIn     0
  validIn   1
▶ dataOut   0 0 0 0 0 0 0 0
  startOut  0
  endOut    0
  validOut  0
```

Initial Delay

The General CRC Generator HDL Optimized block introduces a latency on the output. Assuming the input data is continuous, you can compute the latency by using the equation.

```
initialdelay = (CRC length/input data width) + 2.
```

References

[1] Campobello, G., G. Patane, and M. Russo. “Parallel Crc Realization.” *IEEE Transactions on Computers* 52, no. 10 (October 2003): 1312–19. https://doi.org/10.1109/TC.2003.1234528.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

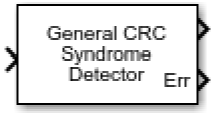
General CRC Generator | General CRC Syndrome Detector HDL Optimized | `comm.HDLCRCGenerator`

Introduced in R2012a

General CRC Syndrome Detector

Detect errors in received codeword frames according to generator polynomial

Library: Communications Toolbox / Error Detection and Correction / CRC



Description

The General CRC Syndrome Detector block computes cyclic redundancy check (CRC) checksums for received codeword frames. For successful CRC detection in a communications system link, you must align the parameter settings of the General CRC Syndrome Detector block with the paired General CRC Generator block.

For more information, see “CRC Syndrome Detector Operation” on page 5-320.

Ports

Input

`In` — Received codeword
binary column vector

Received codeword, specified as a binary column vector.

Data Types: `double` | `Boolean`

Output

`Out` — Output frame
binary column vector

Output frame, returned as a binary column vector that inherits the data type of the input signal. The output frame contains the received codeword with the checksums removed.

The length of the output frame is n - k * r bits, where n is the size of the received codeword, k is the number of checksums per frame, and r is the degree of the generator polynomial.

`Err` — Checksum error signal
binary column vector

Checksum error signal, returned as a binary column vector that inherits the data type of the input signal. The length of `Err` equals the value of **Checksums per frame**. For each checksum computation, an element value of 0 in `Err` indicates no checksum error, and an element value of 1 in `Err` indicates a checksum error.

Parameters

`Generator polynomial` — Generator polynomial

`'z^16 + z^12 + z^5 + 1'` (default) | polynomial character vector | binary row vector | integer row vector

Generator polynomial for the CRC algorithm, specified as one of the following:

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial $x^3+ z^2+ 1$.
- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, `[3 2 0]` represents the polynomial $z^3 + z^2 + 1$.

For more information, see “Character Representation of Polynomials”.

Some commonly used generator polynomials include:

CRC method	Generator polynomial
CRC-32	`'z^32 + z^26 + z^23 + z^22 + z^16 + z^12 + z^11 + z^10 + z^8 + z^7 + z^5 + z^4 + z^2 + z + 1'`
CRC-24	`'z^24 + z^23 + z^14 + z^12 + z^8 + 1'`
CRC-16	`'z^16 + z^15 + z^2 + 1'`
Reversed CRC-16	`'z^16 + z^14 + z + 1'`
CRC-8	`'z^8 + z^7 + z^6 + z^4 + z^2 + 1'`
CRC-4	`'z^4 + z^3 + z^2 + z + 1'`

Example: `'z^7 + z^2 + 1'`, `[1 0 0 0 0 1 0 1]`, and `[7 2 0]` represent the same polynomial, $p(z) = z^7 + z^2 + 1$.

`Initial states` — Initial states of internal shift register

`0` (default) | `1` | binary row vector

Initial states of the internal shift register, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial.

`Direct method` — Use direct algorithm for CRC checksum calculations

`off` (default) | `on`

Select to use the direct algorithm for CRC checksum calculations. When cleared, the block uses the non-direct algorithm for CRC checksum calculations.

For more information on direct and non-direct algorithms, see “Error Detection and Correction”.

`Reflect input bytes` — Reflect input bytes

`off` (default) | `on`

Select to flip the received codeword on a bytewise basis before entering the data into the shift register. When **Reflect input bytes** is selected, the received codeword length divided by the value of the **Checksums per frame** parameter must be an integer and a multiple of 8. When **Reflect input bytes** is cleared, the block does not flip the input data.

`Reflect checksums before final XOR` — Reflect checksums before final XOR

`off` (default) | `on`

Select **Reflect checksums before final XOR** to flip the CRC checksums around their centers after the input data are completely through the shift register. When **Reflect checksums before final XOR** is cleared, the block does not flip the CRC checksums.

`Final XOR` — Final XOR

`0` (default) | `1` | binary row vector

Final XOR, specified as a binary scalar or a binary row vector with a length equal to the degree of the generator polynomial. The XOR operation runs using the value of the **Final XOR** parameter and the CRC checksum before comparing with the input checksum. A scalar value is expanded to a row vector of equal length to the degree of the generator polynomial. A setting of `0` is equivalent to no XOR operation.

`Checksums per frame` — Number of checksums calculated for each frame

`1` (default) | positive integer

Number of checksums calculated for each frame, specified as a positive integer.

Block Characteristics

Data Types	`Boolean` | `double`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

Cyclic Redundancy Check Coding

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

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

When the number of checksums per frame is greater than 1, the input data frame is divided into subframes, the rule is applied to each data subframe, and individual checksums are appended to each subframe. The subframe codewords are concatenated to output one frame.

For a discussion of the supported CRC algorithms, see “Cyclic Redundancy Check Codes”.

CRC Syndrome Detector Operation

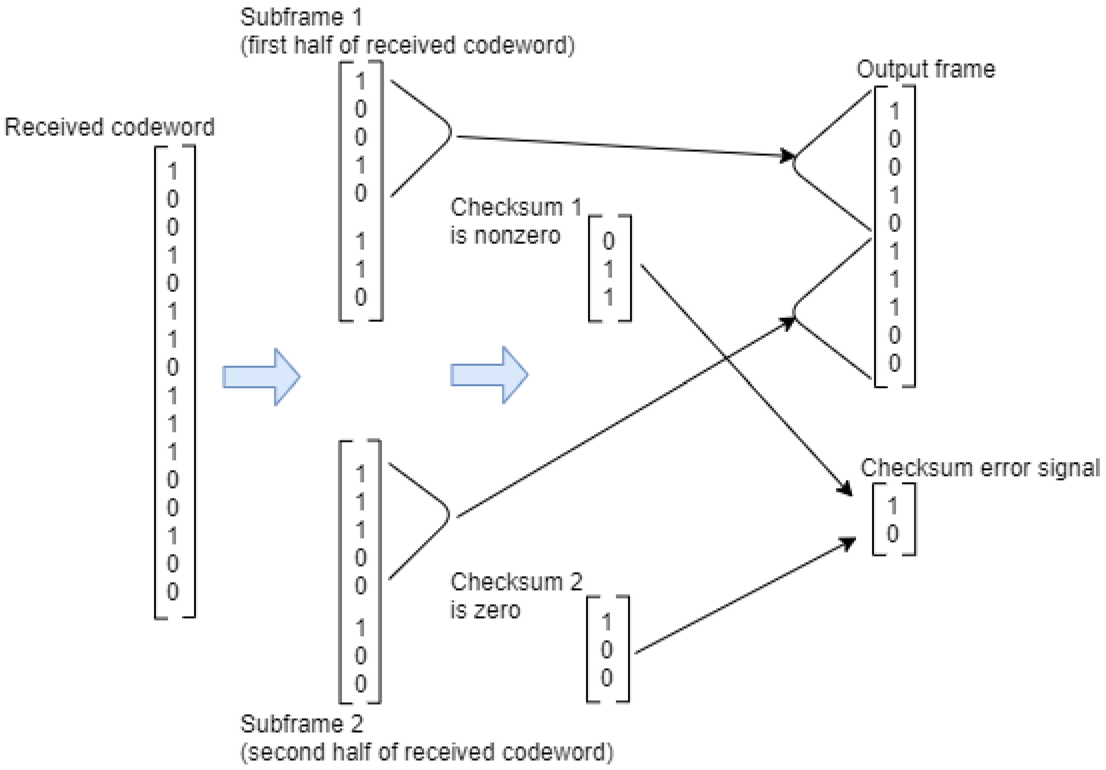
The CRC syndrome detector outputs the received message frame and a checksum error vector according to the specified generator polynomial and number of checksums per frame.

The checksum bits are removed from each subframe, so that the resulting the output frame length is *n* - *k* × *r*, where *n* is the size of the received codeword, *k* is the number of checksums per frame, and *r* is the degree of the generator polynomial. The input frame must be evenly divisible by *k*.

For a specific initial state of the internal shift register:

1. The received codeword is divided into *k* equal sized subframes.
2. The CRC is removed from each of the *k* subframes and compared to the checksum calculated on the received codeword subframes.
3. The output frame is assembled by concatenating the subframe bits of the *k* subframes and then output as a column vector.
4. The checksum error is output as a binary column vector of length *k*. An element value of 0 indicates an error-free received subframe, and an element value of 1 indicates an error occurred in the received subframe.

For the scenario shown here, a 16-bit codeword is received, a third degree generator polynomial computes the CRC checksum, the initial state is 0, and the number of checksums per frame is 2.



Since the number of checksums per frame is 2 and the generator polynomial degree is 3, the received codeword is split in half and two checksums of size 3 are computed, one for each half of the received codeword. The initial states are not shown, because an initial state of `[0]` does not affect the output of the CRC algorithm. The output frame contains the concatenation of the two halves of the received codeword as a single vector of size 10. The checksum error signal output contains a 2-by-1 binary frame vector whose entries depend on whether the computed checksums are zero. As shown in the figure, the first checksum is nonzero and the second checksum is zero, indicating an error occurred in reception of the first half of the codeword.

References

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

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Objects

`comm.CRCDetector`

Blocks

General CRC Generator | General CRC Syndrome Detector HDL Optimized

Topics

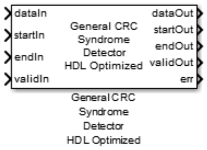
“Cyclic Redundancy Check Codes”

Introduced before R2006a

General CRC Syndrome Detector HDL Optimized

Detect errors in input data using CRC

Library: Communications Toolbox HDL Support / Error Detection and Correction / CRC
Communications Toolbox / Error Detection and Correction / CRC


dataIn
startIn
endIn
validIn
General CRC Syndrome Detector HDL Optimized
dataOut
startOut
endOut
validOut
err
General CRC Syndrome Detector HDL Optimized

Description

The General CRC Syndrome Detector HDL Optimized block performs a cyclic redundancy check (CRC) on data and compares the resulting checksum with the appended checksum. The General CRC Syndrome Detector HDL Optimized block processing is optimized for HDL code generation. If the two checksums do not match, the block reports an error. Instead of processing an entire frame at once, the block accepts and returns a data sample stream with accompanying control signals. The control signals indicate the validity of the samples and the boundaries of the frame. To achieve higher throughput, the block accepts vector data up to the CRC length and implements a parallel architecture.

Ports

Input

`dataIn` — Input data
scalar | vector

Input data, specified as one of these options.

- Scalar – Specify an integer representing several bits. For this case, the block supports an unsigned integer (`uint8`, `uint16`, or `uint32`) or `fixdt(0,N,0)` data type.
- Vector – Specify a vector of binary values. For this case, the block supports a `double` or `Boolean` data type.

The data width must be less than or equal to the CRC length, and the CRC length must be divisible by the data width. For CRC-CCITT/CRC-16, the valid data widths are 16, 8, 4, 2, and 1.

Example: The `uint8` vector input `[0 0 0 1 0 0 1 1]` is equivalent to 19.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `fixed point` | `Boolean`

`startIn` — Start of input frame indicator
scalar

Start of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`endIn` — End of input frame indicator

scalar

End of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`validIn` — Valid input data indicator

scalar

Valid input data indicator, specified as a Boolean scalar.

This a control signal that indicates if the data on the **dataIn** port is valid.

Data Types: `Boolean`

Output

`dataOut` — Output data

scalar | vector

Output data, returned as a scalar or vector. The output data type and size are the same as the input data.

Data Types: `double` | `uint8` | `uint16` | `uint32` | `Boolean` | `fixed point`

`startOut` — Start of output frame indicator

scalar

Start of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`endOut` — End of output frame indicator

scalar

End of output frame indicator, returned as a `Boolean` scalar.

Data Types: `Boolean`

`validOut` — Valid output data indicator

scalar

Valid output data indicator, returned as a Boolean scalar.

This is a control signal that indicates if the data on the **dataOut** port is valid.

Data Types: `Boolean`

`err` — Error indicator

scalar

Error indicator for the corruption of the received data, returned as a Boolean scalar.

When this value is `1`, the message contains at least one error. When this value is `0`, the message contains zero errors.

Data Types: `Boolean`

Parameters

`Polynomial` — Generator polynomial

`[1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 1]` (default) | binary vector

Specify the generator polynomial as a binary vector with coefficients in descending order of powers. The vector length is equal to the degree of the polynomial plus 1.

`Initial state` — Initial conditions of shift register

`0` (default) | binary scalar | binary vector

Specify initial conditions of the internal shift register as a binary, double-precision, or single-precision scalar or vector. For vector inputs, the length of the initial state must be equal to the degree of the generator polynomial.

`Direct method` — Method of calculating checksum

`off` (default) | `on`

Specify the method of calculating checksum as a Boolean scalar.

- Select this parameter to use the direct algorithm for CRC checksum calculations.
- Clear this parameter to use the nondirect algorithm for CRC checksum calculations.

To learn about the direct and non-direct algorithms, see “Cyclic Redundancy Check Codes”.

`Reflect input` — Input byte order

`off` (default) | `on`

Specify the input byte order.

- Select this parameter for the block to flip each input byte before it enters the shift register.
- Clear this parameter for the block to pass the message data to the shift register unchanged.

The input data width must be a multiple of 8.

`Reflect CRC checksum` — Checksum byte order

`off` (default) | `on`

Specify the checksum byte order.

- Select this parameter for the block to flip each checksum byte before passing it to the final XOR stage.
- Clear this parameter for the block to pass the checksum byte to the final XOR stage unchanged.

The input data width must be a multiple of 8.

`Final XOR value` — Checksum

`0` (default) | binary scalar | binary vector

Specify the checksum as a binary, double-precision, or single-precision data type scalar or vector. The block performs XOR operation on the CRC checksum with this value before appending it to the input data.

If you specify a vector input, the vector length must be equal to the degree of the generator polynomial.

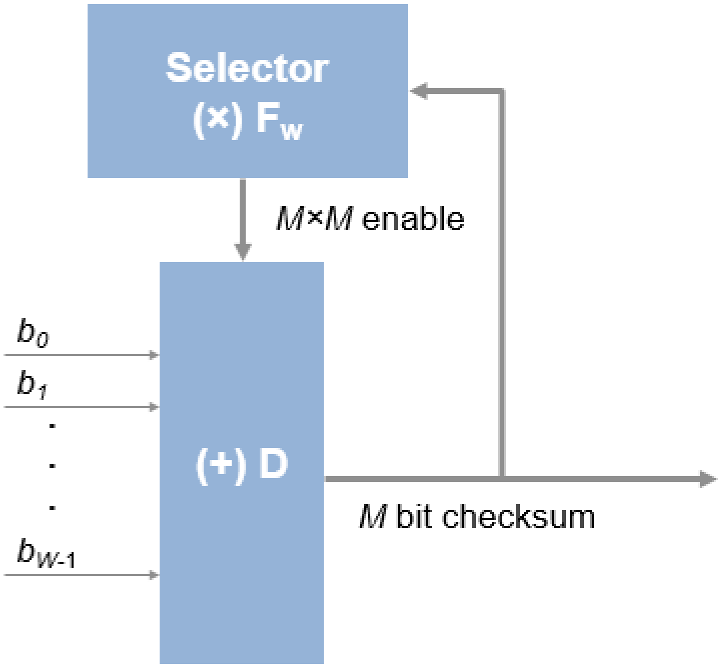
Algorithms

When you use vector or integer input, the block implements a parallel CRC algorithm [1].

To provide high throughput for modern communications systems, the block implements the CRC algorithm with a parallel architecture. This architecture recursively calculates *M* bits of a CRC checksum for each *W* input bits. At the end of the frame, the final checksum result is appended to the message. For a polynomial length of *M*, the recursive checksum calculation for *W* bits in parallel is

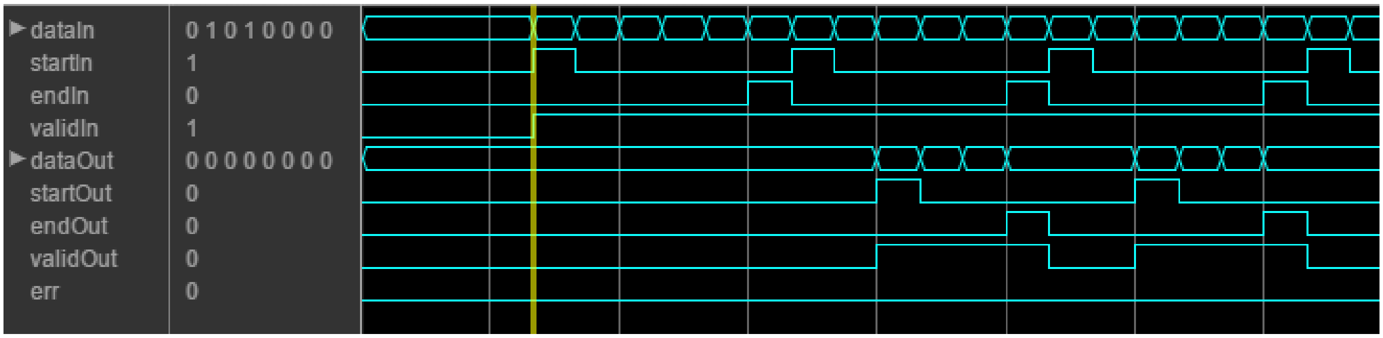
$$X' = F_W(\times)X(+)D .$$

F_W is an *M*-by-*M* matrix that selects elements of the current state for the polynomial calculation with the new input bits. *D* is an *M*-element vector that provides the new input bits, ordered in relation to the generator polynomial and padded with zeros. The block implements the (×) with logical AND and (+) with logical XOR.



Timing Diagram

This waveform shows streaming data and the accompanying control signals for a CRC16 with 8-bit binary vector input. The input frames are contiguous. The output frames include space between them because the detector block removes the checksum word.



This waveform diagram shows continuous input data. Non-continuous data is also supported.

Initial Delay

The General CRC Syndrome Detector HDL Optimized block introduces a latency on the output. This latency can be computed with the following equation, assuming the input data is continuous:

```
initialdelay = 3 * (CRC length/input data width) + 2.
```

References

[1] Campobello, G., G. Patane, and M. Russo. “Parallel Crc Realization.” *IEEE Transactions on Computers* 52, no. 10 (October 2003): 1312–19. https://doi.org/10.1109/TC.2003.1234528.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).

OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

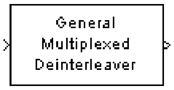
See Also

General CRC Syndrome Detector | General CRC Generator HDL Optimized | `comm.HDLCRCDetector`

Introduced in R2012b

General Multiplexed Deinterleaver

Restore ordering of symbols using specified-delay shift registers



Library

Convolutional sublibrary of Interleaving

Description

The General Multiplexed Deinterleaver block restores the original ordering of a sequence that was interleaved using the General Multiplexed Interleaver block.

In typical usage, the parameters in the two blocks have the same values. As a result, the **Interleaver delay** parameter, *V*, specifies the delays for each shift register in the corresponding *interleaver*, so that the delays of the deinterleaver's shift registers are actually `max(V)-V`.

This block accepts a scalar or column vector input signal, which can be real or complex. The output signal has the same sample time as the input signal.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of the output will be the same as that of the input signal.

Parameters

Interleaver delay (samples)

A vector that lists the number of symbols that fit in each shift register of the corresponding interleaver. The length of this vector is the number of shift registers.

Initial conditions

The values that fill each shift register when the simulation begins.

Pair Block

General Multiplexed Interleaver

References

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

HDL Code Generation
Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

The implementation for the General Multiplexed Deinterleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` tonone.

When you set `ResetType` to `none`, reset is not applied to the shift registers. When registers are not fully loaded, mismatches between Simulink and the generated code occur for some number of samples during the initial phase. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Set the **Ignore output data checking (number of samples)** option accordingly. (If you are using the command-line interface, you can use the `IgnoreDataChecking` property for this purpose.)

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see "OutputPipeline" (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also "ResetType" (HDL Coder).

See Also

Blocks
Convolutional Deinterleaver | General Multiplexed Interleaver | Helical Deinterleaver

Introduced before R2006a

General Multiplexed Interleaver

Permute input symbols using set of shift registers with specified delays

Library

Convolutional sublibrary of Interleaving

Description

The General Multiplexed Interleaver block permutes the symbols in the input signal. Internally, it uses a set of shift registers, each with its own delay value.

This block accepts a scalar or column vector input signal, which can be real or complex. The input and output signals have the same sample time.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal has the same data type as the input signal.

Parameters

Interleaver delay (samples)

A column vector listing the number of symbols that fit into each shift register. The length of this vector is the number of shift registers. (In sample-based mode, it can also be a row vector.)

Initial conditions

The values that fill each shift register at the beginning of the simulation.

If **Initial conditions** is a scalar, then its value fills all shift registers. If **Initial conditions** is a column vector, then each entry fills the corresponding shift register. (In sample-based mode, **Initial conditions** can also be a row vector.) If a given shift register has zero delay, then the value of the corresponding entry in the **Initial conditions** vector is unimportant.

Pair Block

General Multiplexed Deinterleaver

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

The implementation for the General Multiplexed Interleaver block is shift register based. If you want to suppress generation of reset logic, set the implementation parameter `ResetType` to`'none'`.

When you set `ResetType` to`'none'`, reset is not applied to the shift registers. Mismatches between Simulink and the generated code occur for some number of samples during the initial phase, when registers are not fully loaded. To avoid spurious test bench errors, determine the number of samples required to fill the shift registers. Then, set the **Ignore output data checking (number of samples)** option accordingly. (You can use the `IgnoreDataChecking` property for this purpose, if you are using the command-line interface.)

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).
ResetType	Suppress reset logic generation. The default is `default`, which generates reset logic. See also “ResetType” (HDL Coder).

See Also

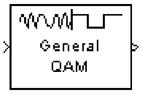
Blocks

Convolutional Interleaver | General Multiplexed Deinterleaver | Helical Interleaver

Introduced before R2006a

General QAM Demodulator Baseband

Demodulate QAM-modulated data



Library

AM, in Digital Baseband sublibrary of Modulation

Description

The General QAM Demodulator Baseband block demodulates a signal that was modulated using quadrature amplitude modulation. The input is a baseband representation of the modulated signal.

The input must be a discrete-time complex signal. The **Signal constellation** parameter defines the constellation by listing its points in a length-M vector of complex numbers. The block maps the *m*th point in the **Signal constellation** vector to the integer *m*-1.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 5-338 table on this page.

Parameters

Signal constellation

A real or complex vector that lists the constellation points.

Output type

Determines whether the block produces integers or binary representations of integers.

If you set this parameter to `Integer`, the block produces integers.

If you set this parameter to `Bit`, the block produces a group of K bits, called a *binary word*, for each symbol, when **Decision type** is set to `Hard decision`. If **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the block outputs bitwise LLR and approximate LLR, respectively.

Decision type

This field appears when `Bit` is selected in the pull-down list **Output type**.

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. See “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the *Communications Toolbox User's Guide* for algorithm details.

Noise variance source

This field appears when you set `Approximate log-likelihood ratio` or `Log-likelihood ratio` for **Decision type**.

When you set this parameter to `Dialog`, you can then specify the noise variance in the **Noise variance** field. When you set this option to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

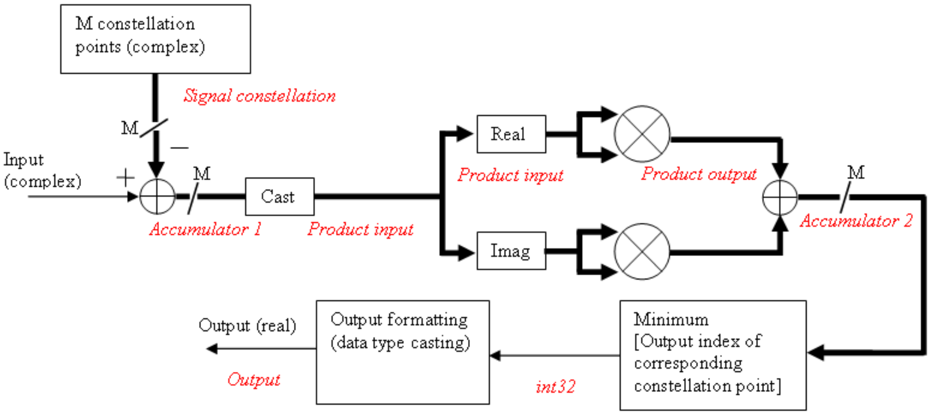
If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.

Fixed-Point Signal Flow Diagrams

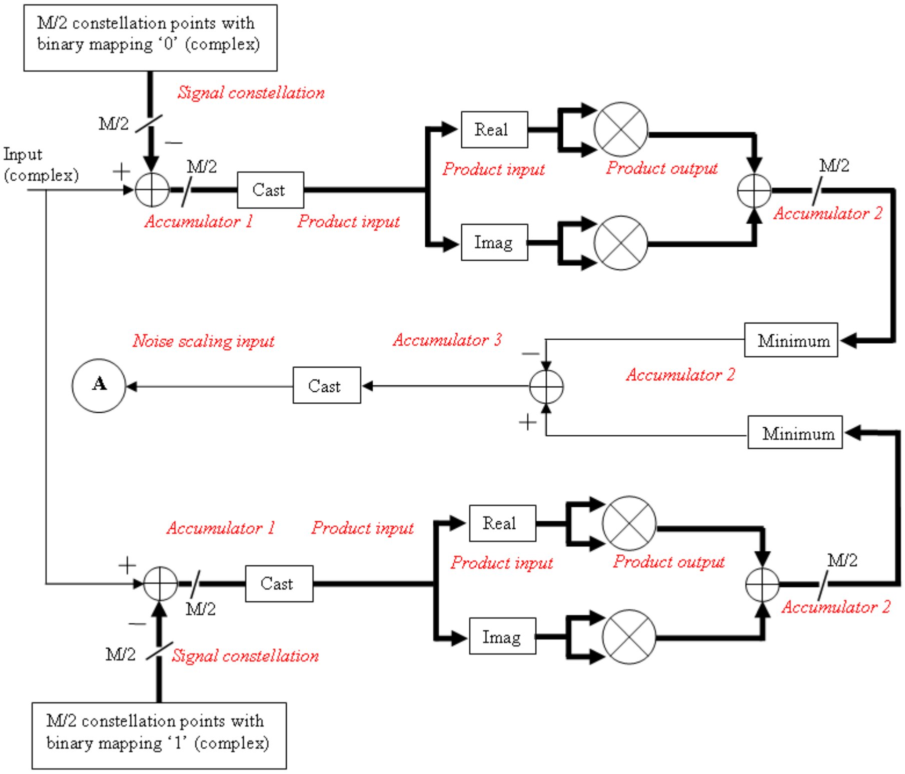


Fixed-Point Signal Flow Diagram for Hard Decision Mode

Note In the figure above, *M* represents the size of the **Signal constellation** .

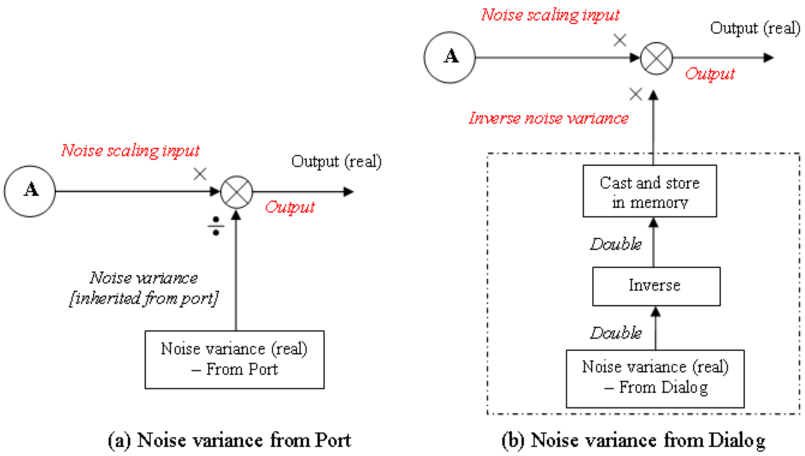
The general QAM Demodulator Baseband block supports fixed-point operations for computing Hard Decision (**Output** type set to `Bit` and **Decision type** is set to `Hard decision`) and Approximate LLR (**Output type** is set to `Bit` and **Decision type** is set to `Approximate Log-Likelihood ratio`) output values. The input values must have fixed-point data type for fixed-point operations.

Note Fixed-Point operations are NOT yet supported for Exact LLR output values.



Fixed-Point Signal Flow Diagram for Approximate LLR Mode

Note In the figure above, *M* represents the size of the **Signal constellation**.


Noise scaling input
A
×
Output (real)
Output
÷
Noise variance [inherited from port]
Noise variance (real) – From Port
(a) Noise variance from Port
Noise scaling input
A
×
Output (real)
Output
×
Inverse noise variance
Cast and store in memory
Double
Inverse
Double
Noise variance (real) – From Dialog
(b) Noise variance from Dialog

Fixed-Point Signal Flow Diagram for Approximate LLR Mode: Noise Variance Operation Modes

Note If **Noise variance** is set to `Dialog`, the block performs the operations shown inside the dotted line once during initialization. The block also performs these operations if the **Noise variance** value changes during simulation.

Data Types Attributes

Output

The block supports the following Output options:

When you set the parameter to `Inherit via internal rule` (default setting), the block inherits the output data type from the input port. The output data type is the same as the input data type if the input is of type `single` or `double`.

For integer outputs, you can set this block's output to `Inherit via internal rule` (default setting), `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`.

For bit outputs, when you set **Decision type** to `Hard decision`, you can set the output to `Inherit via internal rule`, `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

When you set **Decision type** to `Hard decision` or `Approximate log-likelihood ratio` and the input is a floating point data type, then the output inherits its data type from the input. For example, if the input is of data type `double`, the output is also of data type `double`. When you set **Decision type** to `Hard decision` or `Approximate log-likelihood ratio`, and the input is a fixed-point signal, the **Output** parameter, located in the Fixed-Point algorithm parameters region of the Data-Type tab, specifies the output data type.

When you set the parameter to `Smallest unsigned integer`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box. If you select `ASIC/FPGA` in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., `ufix(1)` for bit outputs, and $ufix(\lceil \log_2 M \rceil)$ for integer outputs. For all other choices, the **Output** data type is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., `uint8`).

Rounding Mode Parameter

Use this parameter to specify the rounding method to be used when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result.

For more information, see “Rounding Modes” or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result:

- Saturate represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used.
- Wrap uses modulo arithmetic to cast an overflow back into the representable range of the data type. See Modulo Arithmetic (Fixed-Point Designer) for more information.

For more information, see the **Saturate on integer overflow** parameter subsection of “Specify Fixed-Point Attributes for Blocks”.

Signal constellation

Use this parameter to define the data type of the **Signal constellation** parameter.

- When you select `Same word length as input` the word length of the **Signal constellation** parameter matches that of the input to the block. The fraction length is computed to provide the best precision for given signal constellation values.
- When you select `Specify word length`, the **Word Length** field appears, and you may enter a value for the word length. The fraction length is computed to provide the best precision for given signal constellation values.

Accumulator 1

Use this parameter to specify the data type for **Accumulator 1**:

- When you select `Inherit via internal rule`, the block automatically calculates the output word and fraction lengths. For more information, see the “Inherit via Internal Rule” subsection of the *DSP System Toolbox™ User's Guide*.
- When you select `Binary point scaling`, you can enter the word length and the fraction length of **Accumulator 1**, in bits.

Product Input

Use this parameter to specify the data type for **Product input**.

- When you select `Same as accumulator 1`, the **Product Input** characteristics match those of **Accumulator 1**.
- When you select `Binary point scaling` you can enter the word length and the fraction length of **Product input**, in bits.

Product Output

Use this parameter to select the data type for Product output.

- When you select `Inherit via internal rule`, the block automatically calculates the output signal type. For more information, see the `Inherit via Internal Rule` subsection of the *DSP System Toolbox User's Guide*.
- When you select `Binary point scaling` enter the word length and the fraction length for **Product output**, in bits.

Accumulator 2

Use this parameter to specify the data type for **Accumulator 2**:

- When you select `Inherit via internal rule`, the block automatically calculates the accumulator data type. The internal rule calculates the ideal, full-precision word length and fraction length as follows:

  $WL_{\text{ideal accumulator 2}} = WL_{\text{input to accumulator 2}}$

  $FL_{\text{ideal accumulator 2}} = FL_{\text{ input to accumulator 2}}$

  After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see The Effect of the

Hardware Implementation Pane on the Internal Rule subsection of the *DSP System Toolbox User's Guide*.

The internal rule always sets the sign of data-type to `Unsigned` .

- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of **Accumulator 2**, in bits.

The settings for the following fixed-point parameters only apply when you set **Decision type** to `Approximate log-likelihood ratio`.

Accumulator 3

When you select `Inherit via internal rule`, the block automatically calculates the accumulator data type. The internal rule first calculates ideal, full-precision word length and fraction length as follows:

$WL_{\text{ideal accumulator 3}} = WL_{\text{input to accumulator 3}} + 1$

$FL_{\text{ideal accumulator 3}} = FL_{\text{input to accumulator 3}}$.

After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see The Effect of the Hardware Implementation Pane on the Internal Rule subsection of the *DSP System Toolbox User's Guide*.

The internal rule always sets the sign of data-type to `Signed`.

Noise scaling input

- When you select `Same as accumulator 3`, the **Noise scaling input** characteristics match those of **Accumulator 3**.
- When you select `Binary point scaling` you are able to enter the word length and the fraction length of **Noise scaling input**, in bits.

Inverse noise variance

This field appears when **Noise variance** source is set to Dialog.

- When you select `Same word length as input` the word length of the **Inverse noise variance** parameter matches that of the input to the block. The fraction length is computed to provide the best precision for a given inverse noise variance value.
- When you select `Specify word length`, the **Word Length** field appears, and you may enter a value for the word length. The fraction length is computed to provide the best precision for a given inverse noise variance value.

Output

When you select `Inherit via internal rule` , the **Output data type** is automatically set for you.

If you set the **Noise variance source** parameter to `Dialog`, the output is a result of product operation as shown in the Noise Variance Operation Modes Signal Flow Diagram “Fixed-Point Signal Flow Diagram for Approximate LLR Mode: Noise Variance Operation Modes” on page 5-334. In this case, it follows the internal rule for Product data types specified in the Inherit via Internal Rule subsection of the *DSP System Toolbox User's Guide*.

If the **Noise variance source** parameter is set to `Port`, the output is a result of division operation as shown in the signal flow diagram. In this case, the internal rule calculates the ideal, full-precision word length and fraction length as follows:

$WL_{\text{output}} = \max(WL_{\text{Noise scaling input}}, WL_{\text{Noise variance}})$

$FL_{\text{output}} = FL_{\text{Noise scaling input (dividend)}} - FL_{\text{Noise variance (divisor)}}$.

After the full-precision result is calculated, your particular hardware may still affect the final word and fraction lengths set by the internal rule. For more information, see “The Effect of the Hardware Implementation Pane on the Internal Rule” subsection of the *DSP System Toolbox User's Guide*.

The internal rule for **Output** always sets the sign of data-type to `Signed`.

For additional information about the parameters pertaining to fixed-point applications, see “Specify Fixed-Point Attributes for Blocks”.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed–point when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is either `Hard-decision` or `Approximate LLR`
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` and **Decision type** is `Hard-decision`. • 8-, 16-, and 32-bit signed integers when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is `Hard-decision` • 8-, 16-, and 32-bit unsigned integers when **Output type** is `Integer` or **Output type** is `Bit` and **Decision type** is `Hard-decision` • ufix(1) in ASIC/FPGA when **Output type** is `Bit` and **Decision type** is `Hard-decision` • $ufix(\lceil \log_2 M \rceil)$ in ASIC/FPGA when **Output type** is `Integer` • Signed fixed-point when **Output type** is `Bit` and **Decision type** is `Approximate LLR`

Pair Block

General QAM Modulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

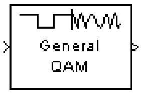
Blocks

General QAM Modulator Baseband | Rectangular QAM Demodulator Baseband

Introduced before R2006a

General QAM Modulator Baseband

Modulate using quadrature amplitude modulation



Library

AM, in Digital Baseband sublibrary of Modulation

Description

The General QAM Modulator Baseband block modulates using quadrature amplitude modulation. The output is a baseband representation of the modulated signal.

The **Signal constellation** parameter defines the constellation by listing its points in a length-M vector of complex numbers. The input signal values must be integers between 0 and M-1. The block maps an input integer m to the (m+1)st value in the **Signal constellation** vector.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 5-341 table on this page.

Constellation Visualization

The General QAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications Toolbox User's Guide*.

Parameters

Signal constellation

A real or complex vector that lists the constellation points.

Output data type

The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this to `Fixed-point` or `User-defined` will enable fields in which you can further specify details. Setting this to `Inherit via back propagation`, sets the output data type and scaling to match the following block..

Output word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following two methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Output fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter, or when you select `User-defined` and the specified output data type is a fixed-point data type.

Output fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • 8-, 16-, 32-bit signed integers • 8-, 16-, 32-bit unsigned integers • $ufix(\lceil \log_2 M \rceil)$
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

General QAM Demodulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

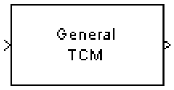
Blocks

General QAM Demodulator Baseband | Rectangular QAM Modulator Baseband

Introduced before R2006a

General TCM Decoder

Decode trellis-coded modulation data, mapped using arbitrary constellation



Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The General TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using an arbitrary signal constellation.

The **Trellis structure** and **Signal constellation** parameters in this block should match those in the General TCM Encoder block, to ensure proper decoding. In particular, the **Signal constellation** parameter must be in set-partitioned order.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. The input signal must be `double` or `single`. The reset port signal must be `double` or `Boolean`. For information about the data types each block port supports, see “Supported Data Types” on page 5-344.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the General TCM Decoder block's output is a binary column vector whose length is *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

  If you select **Enable the reset input port**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.
- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.
- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you

know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Signal constellation

A complex vector that lists the points in the signal constellation in set-partitioned order.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode

The operation mode of the Viterbi decoder. The choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port

When you select this check box, the block has a second input port labeled `Rst`. Providing a nonzero value to this port causes the block to set its internal memory to the initial state before processing the input data. This field appears only if you set **Operation mode** to `Continuous`.

Output data type

Select the data type for the block output signal as `boolean` or `single`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Reset	• Double-precision floating point • Boolean
Output	• Double-precision floating point • Boolean

Pair Block

General TCM Encoder

References

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

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General TCM Encoder | M-PSK TCM Decoder | Rectangular QAM TCM Decoder

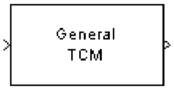
Functions

`poly2trellis`

Introduced before R2006a

General TCM Encoder

Convolutionally encode binary data and map using arbitrary constellation



Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The General TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to an arbitrary signal constellation. The **Signal constellation** parameter lists the signal constellation points in set-partitioned order. This parameter is a complex vector with a length, *M*, equal to the number of possible output symbols from the convolutional encoder. (That is, $\log_2 M$ is equal to n for a rate *k*/*n* convolutional code.)

Input Signals and Output Signals

If the convolutional encoder represents a rate *k*/*n* code, then the General TCM Encoder block's input must be a binary column vector with a length of *L***k* for some positive integer *L*.

This block accepts a binary-valued input signal. The output signal is a complex column vector of length *L*. For information about the data types each block port supports, see “Supported Data Types” on page 5-348.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code”. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink software to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation`

mode to **Reset on nonzero input via port**. The block then opens a second input port, labeled Rst. The signal at the Rst port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

Signal Constellations

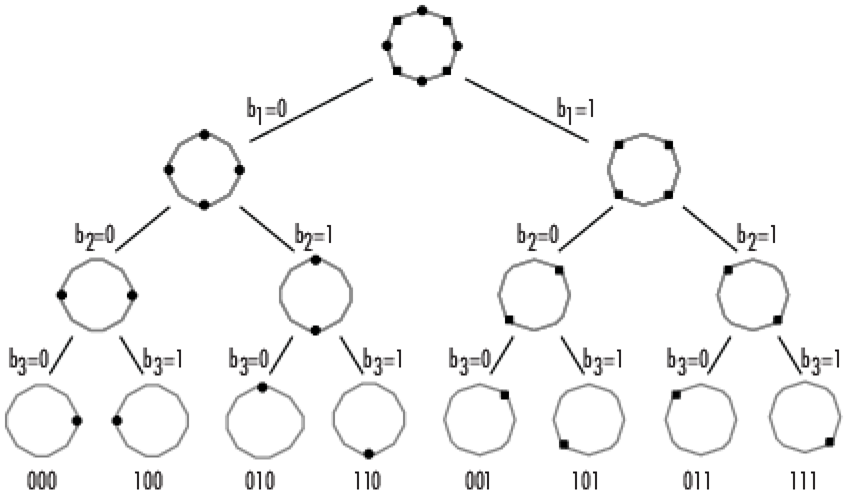
The trellis-coded modulation technique partitions the constellation into subsets called cosets so as to maximize the minimum distance between pairs of points in each coset.

Note When you set the **Signal constellation** parameter, you must ensure that the constellation vector is already in set-partitioned order. Otherwise, the block might produce unexpected or suboptimal results.

As an example, the diagram below shows one way to devise a set-partitioned order for the points for an 8-PSK signal constellation. The figure at the top of the tree is the entire 8-PSK signal constellation, while the eight figures at the bottom of the tree contain one constellation point each. Each level of the tree corresponds to a different bit in a binary sequence (b_3,b_2,b_1), while each branch in a given level of the tree corresponds to a particular value for that bit. Listing the constellation points using the sequence at the bottom of the tree leads to the vector

```
exp(2*pi*j*[0 4 2 6 1 5 3 7]/8)
```

which is a valid value for the **Signal constellation** parameter in this block.



For other examples of signal constellations in set-partitioned order, see [1] or the reference pages for the M-PSK TCM Encoder and Rectangular QAM TCM Encoder blocks.

Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes [3].

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by $y = n \cdot (x + s)/k$, where x is the number of input bits, and s = constraint length − 1 (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

Signal constellation

A complex vector that lists the points in the signal constellation in set-partitioned order.

Output data type

The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

General TCM Decoder

References

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

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001.

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

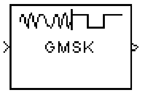
Blocks
General TCM Decoder | M-PSK TCM Encoder | Rectangular QAM TCM Encoder

Functions
`poly2trellis`

Introduced before R2006a

GMSK Demodulator Baseband

Demodulate GMSK-modulated data



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The GMSK Demodulator Baseband block uses a Viterbi algorithm to demodulate a signal that was modulated using the Gaussian minimum shift keying method. The input to this block is a baseband representation of the modulated signal.

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`. If you set the **Output type** parameter to `Integer`, then the block produces values of 1 and -1. If you set the **Output type** parameter to `Bit`, then the block produces values of 0 and 1.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is two times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

For a column vector input signal, the width of the input equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, D, in this block is the number of trellis branches used to construct each traceback path. D influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the five-times-the-constraint-length rule, which corresponds to 5 · log2(*numStates*). The number of states is determined by the following equation:

$$numStates = \begin{Bmatrix} p \cdot 2^{(L-1)}, \ for\,even\,m \\ 2p \cdot 2^{(L-1)}, \ for\,odd\,m \end{Bmatrix}$$

where:

- $h = m/p$ is the modulation index in proper rational form
  - m = numerator of modulation index
  - p = denominator of modulation index
- L is the Pulse length

Parameters

Output type

Determines whether the output consists of bipolar or binary values.

BT product

The product of bandwidth and time.

Pulse length (symbol intervals)

The length of the frequency pulse shape.

Symbol prehistory

The data symbols the modulator uses before the start of the simulation.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see “Upsample Signals and Rate Changes” in *Communications Toolbox User's Guide*.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).

- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth

The number of trellis branches that the GMSK Demodulator Baseband block uses to construct each traceback path.

Output data type

The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

GMSK Modulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

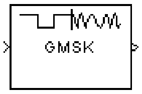
Blocks

CPM Demodulator Baseband | GMSK Modulator Baseband | Viterbi Decoder

Introduced before R2006a

GMSK Modulator Baseband

Modulate using Gaussian minimum shift keying method



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The GMSK Modulator Baseband block modulates using the Gaussian minimum shift keying method. The output is a baseband representation of the modulated signal.

The **BT product** parameter represents bandwidth multiplied by time. This parameter is a nonnegative scalar. It is used to reduce the bandwidth at the expense of increased intersymbol interference. The **Pulse length** parameter measures the length of the Gaussian pulse shape, in symbol intervals. For an explanation of the pulse shape, see the work by Anderson, Aulin, and Sundberg among the references on page 5-355 listed below. The frequency pulse shape is defined by the following equations.

$$g(t) = \frac{1}{2T}\left\{Q\left[2\pi B_b\frac{t-\frac{T}{2}}{\sqrt{\ln(2)}}\right] - Q\left[2\pi B_b\frac{t+\frac{T}{2}}{\sqrt{\ln(2)}}\right]\right\}$$

$$Q(t) = \int_t^\infty \frac{1}{\sqrt{2\pi}}e^{-\tau^2/2}d\tau$$

For this block, an input symbol of 1 causes a phase shift of π/2 radians.

The group delay is the number of samples between the start of a filter's response and its peak. The group delay that the block introduces is **Pulse length**/2 * **Samples per symbol** (using a reference of output sample periods). For GMSK, **Pulse length** denotes the truncated frequency pulse length in symbols. The net delay effect at the receiver (demodulator) is due to the **Traceback depth** parameter, which in most cases would be larger than the group delay.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, then the block accepts values of 1 and -1.

When you set the **Input type** parameter to `Bit`, then the block accepts values of 0 and 1.

This block accepts a scalar-valued or column vector input signal. For a column vector input signal, the width of the output equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of 2.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Parameters

Input type

Indicates whether the input consists of bipolar or binary values.

BT product

The product of bandwidth and time.

The block uses this parameter to reduce bandwidth at the expense of increased intersymbol interference. Enter a nonnegative scalar value for this parameter.

Pulse length (symbol intervals)

The length of the frequency pulse shape.

Symbol prehistory

A scalar or vector value that specifies the data symbols the block uses before the start of the simulation, in reverse chronological order. If it is a vector, then its length must be one less than the **Pulse length** parameter.

Phase offset (rad)

The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or bit in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes”.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Output data type

The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (When **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

GMSK Demodulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CPM Modulator Baseband | GMSK Demodulator Baseband

Topics

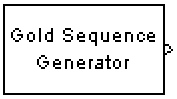
“Compare GMSK and MSK Signals in Simulink”

Introduced before R2006a

Gold Sequence Generator

Generate Gold sequence from set of sequences

Library: Communications Toolbox / Comm Sources / Sequence Generators



Description

The Gold Sequence Generator block generates a binary sequence with small periodic cross-correlation properties from a bounded set of sequences. For more information on Gold sequences, see “Gold Sequences” on page 5-360.

This block can output sequences that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Ports

Input

`oSiz` — Current output size

scalar | two-element row vector

Current output size, specified as a scalar or a two-element row vector. The second element of the vector must be `1`.

Example: `[10 1]` indicates the current output column vector will be of size 10-by-1.

Dependencies

To enable this port select the **Output variable-size signals** parameter and set **Maximum output size source** to `Dialog parameter`.

Data Types: `double`

`Ref` — Reference input signal

scalar | column vector

Reference input signal, specified as a scalar, column vector.

Dependencies

To enable this port select the **Output variable-size signals** parameter and set **Maximum output size source** to `Inherit from reference input`.

Data Types: `double`

`Rst` — Reset signal

scalar | column vector

Reset signal, specified in one of these forms.

- When the output size is variable specify as a scalar.
- Otherwise, specify as a scalar or a 2-D column vector with a length equal to **Samples per frame**.

The output signal resets for nonzero **Rst** input values. For more information, see “Reset Behavior” on page 5-362

Dependencies

To enable this port, select the **Reset on nonzero input** parameter.

Data Types: `double`

Output

`Out` — Output signal

binary column vector

Output signal, returned as a binary column vector. At least one element of the **Initial states (1)** or **Initial states (2)** vector must be nonzero in order for the block to generate a nonzero sequence.

Data Types: `double`

Parameters

`Preferred polynomial (1)` — First sequence polynomial

`'z^6 + z + 1'` (default) | polynomial character vector | binary row vector | integer row vector

First sequence polynomial, specified in one of these forms.

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial x^3+ z^2+ 1.
- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, `[3 2 0]` represents the polynomial $z^3 + z^2 + 1$.

For more information, see “Character Representation of Polynomials” and “Preferred Pairs of Sequences” on page 5-363.

`Initial states (1)` — Initial states for first sequence polynomial

`[0 0 0 0 0 1]` (default) | binary vector

Initial states of the shift register for first sequence polynomial of the preferred pair, specified as a binary vector with length equal to the degree of **Preferred polynomial (1)**.

`Preferred polynomial (2)` — Second sequence polynomial

`'z^6 + z^5 + z^2 + z + 1'` (default) | polynomial character vector | binary row vector | integer row vector

Second sequence polynomial, specified in one of these forms.

- A polynomial character vector such as `'z^3 + z^2 + 1'`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The length of this vector is (N+1), where N is the degree of the generator polynomial. For example, `[1 1 0 1]` represents the polynomial x^3+ z^2+ 1.

- An integer row vector containing the exponents of z for the nonzero terms in the polynomial in descending order. For example, `[3 2 0]` represents the polynomial $z^3 + z^2 + 1$.

For more information, see “Character Representation of Polynomials”.

`Initial states (2)` — Initial states for second sequence polynomial

`[0 0 0 0 0 1]` (default) | binary vector

Initial states of the shift register for second sequence polynomial of the preferred pair, specified as a binary vector with length equal to the degree of **Preferred polynomial (2)**.

`Sequence index` — Sequence index

`0` (default) | integer scalar in the range [–2, 2^n–2]

Sequence index of the output sequence from the set of sequences, specified as an integer scalar in the range [–2, 2^n–2]. n is the degree of the preferred polynomials.

`Shift` — Offset of Gold sequence

`0` (default) | integer scalar

Offset of Gold sequence from the initial time, specified as an integer scalar.

`Output variable-size signals` — Output variable-size signals

off (default) | on

Select this parameter to permit variable length output sequences during simulation. When set to `off`, fixed-length sequences are output. When set to `on`, variable-length sequences can be output. For information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

`Maximum output size source` — Maximum output size source

`Dialog parameter` (default) | `Inherit from reference port`

Maximum output size source, which indicates how the maximum sequence output size is specified.

- `Dialog parameter` configures the block to use the **Maximum output size** parameter setting as the maximum permitted output sequence length. When you make this selection, the **oSiz** input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value of **oSiz** must be less than or equal to the **Maximum output size** parameter.
- `Inherit from reference port` adds the `Ref` input port and configures the block to inherit the sample time, maximum size, and the current output size from the variable-sized signal at the `Ref` input port to set the maximum permitted output sequence length.

Dependencies

To enable this parameter, select **Output variable-size signals**.

`Maximum output size` — Maximum output size

`[10 1]` (default) | two-element row vector

Maximum output size, specified as a two-element row vector that denotes the maximum output size for the block. The second element of the vector must be `1`.

Example: `[10 1]` gives a 10-by-1 maximum sized output signal.

Dependencies

To enable this parameter select **Output variable-size signals** and set **Maximum output size source** to `Dialog parameter`.

Data Types: `double`

`Sample time` — Output sample time

`1` (default)

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-360.

Dependencies

To enable this parameter do not select **Output variable-size signals**.

`Samples per frame` — Samples per frame

`1` (default) | positive integer

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-360.

Dependencies

To enable this parameter do not select **Output variable-size signals**.

`Reset on nonzero input` — Reset output signal

`off` (default) | `on`

Select this parameter to enable the **Rst** port. When a nonzero value is input at the **Rst** port, the internal shift registers are reset to the original values of the **Initial states (1)** and **Initial states (2)** parameters.

`Output data type` — Output data type

`double` (default) | `boolean` | `Smallest unsigned integer`

Output data type, specified as `boolean`, `double`, or `Smallest unsigned integer`.

When set to `Smallest unsigned integer`, the output data type is selected based on the settings used in the “Hardware Implementation Pane” (Simulink) of the Configuration Parameters dialog box of the model. If ASIC/FPGA is selected in the Hardware Implementation pane, the output data type is the ideal minimum one-bit size (ufix(1)). For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (uint8).

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`
Multidimensional Signals	`no`

Variable-Size Signals	yes

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Gold Sequences

The characteristic cross-correlation properties of Gold sequences make them useful when multiple devices are broadcasting in the same frequency range. The Gold sequences are defined using a specified pair of sequences *u* and *v*, of period $N = 2^n - 1$, called a preferred pair, as defined in “Preferred Pairs of Sequences” on page 5-363. The set $G(u, v)$ of Gold sequences is defined by

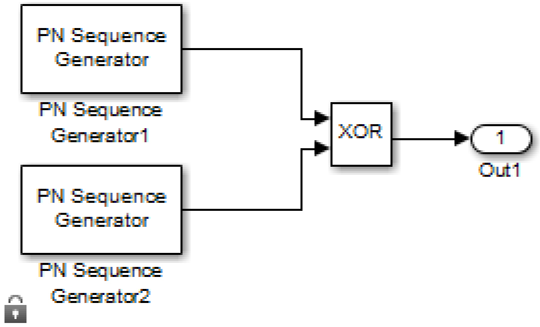
$$G(u,v) = \left\{u, v, u \oplus v, u \oplus Tv, u \oplus T^2 v, \ldots, u \oplus T^{N-1} v\right\}$$

where *T* represents the operator that shifts vectors cyclically to the left by one place, and ⊕ represents addition modulo 2. Note that $G(u,v)$ contains $N + 2$ sequences of period *N*. The Gold Sequence Generator block outputs one Gold sequence according to the configured parameters and inputs.

Gold sequences have the property that the cross-correlation between any two, or between shifted versions of them, takes on one of three values: $-t(n)$, -1, or $t(n) - 2$, where

$$t(n) = \begin{cases} 1 + 2^{(n+1)/2} & n \text{ even} \\ 1 + 2^{(n+2)/2} & n \text{ odd} \end{cases}$$

The Gold Sequence Generator block uses two PN Sequence Generator blocks to generate the preferred pair of sequences, and then XORs these sequences to produce the output sequence, as shown in the following diagram.



You can specify the preferred pair by the **Preferred polynomial (1)** and **Preferred polynomial (2)** parameters in the dialog for the Gold Sequence Generator block. These polynomials, both of which must have degree *n*, describe the shift registers that the PN Sequence Generator blocks use to generate their output. For more details on how these sequences are generated, see the reference page for the PN Sequence Generator block. You can specify the preferred polynomials using these formats:

- A polynomial character vector that includes the number `1`, for example, `'z^4 + z + 1'`.
- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, the polynomial $z^5 + z^2 + 1$ can be represented by the character vector `'z^5 + z^2 + 1'`, the vector `[5 2 0]`, and the vector `[1 0 0 1 0 1]`.

The following table provides a short list of preferred pairs.

n	N	Preferred Polynomial (1)	Preferred Polynomial (2)
5	31	`[5 2 0]`	`[5 4 3 2 0]`
6	63	`[6 1 0]`	`[6 5 2 1 0]`
7	127	`[7 3 0]`	`[7 3 2 1 0]`
9	511	`[9 4 0]`	`[9 6 4 3 0]`
10	1023	`[10 3 0]`	`[10 8 3 2 0]`
11	2047	`[11 2 0]`	`[11 8 5 2 0]`

The **Initial states (1)** and **Initial states (2)** parameters are vectors specifying the initial values of the registers corresponding to **Preferred polynomial (1)** and **Preferred polynomial (2)**, respectively. These parameters must satisfy these criteria:

- All elements of the **Initial states (1)** and **Initial states (2)** vectors must be binary numbers.
- The length of the **Initial states (1)** vector must equal the degree of the **Preferred polynomial (1)**, and the length of the **Initial states (2)** vector must equal the degree of the **Preferred polynomial (2)**.

Note At least one element of the initial states vectors (**Initial states (1)** or **Initial states (2)**) must be nonzero in order for the block to generate a nonzero sequence. Specifically, the initial state of at least one of the registers must be nonzero.

The **Sequence index** parameter specifies which Gold sequence in the set $G(u, v)$ is output. The range of **Sequence index** is $[–2, –1, 0, 1, 2, ..., 2^n–2]$, where *n* is the degree of the generator polynomials specified by the **Preferred polynomial (1)** and **Preferred polynomial (2)** parameters. This table shows the correspondence between **Sequence index** and the output sequence.

Sequence Index	Output Sequence
–2	u
–1	v
0	$u \oplus v$
1	$u \oplus Tv$

Sequence Index	Output Sequence
2	$u \oplus T^{2}v$
…	…
2^{n} –2	$u \oplus T^{2^{n}-2}v$

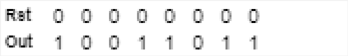
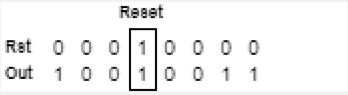
T represents the operator that shifts vectors cyclically to the left by one place, and ⊕ represents addition modulo 2.

You can shift the starting point of the Gold sequence with the **Shift** parameter, which is an integer representing the length of the shift.

You can use an external signal to reset the values of the internal shift register to the initial state by selecting **Reset on nonzero input**. This creates an input port for the external signal in the Gold Sequence Generator block. The way the block resets the internal shift register depends on whether its output signal and the reset signal are scalar or vector. For more information, see “Reset Behavior” on page 5-362.

Reset Behavior

To reset the generator sequence, you must first select **Reset on nonzero input** to add the `Rst` input. Suppose that the Gold Sequence Generator block outputs `[1 0 0 1 1 0 1 1]` when there is no reset. The following table shows the effect on the Gold Sequence Generator block output for the property values indicated.

	Reset Signal Properties	Gold Sequence Generator block	Reset Signal Output Signal
No reset	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 0 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1 **Out** = `[1 0 0 1 1 0 1 1]`	 Rst 0 0 0 0 0 0 0 0 Out 1 0 0 1 1 0 1 1
Scalar reset signal	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 1 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1	 Reset Rst 0 0 0 1 0 0 0 0 Out 1 0 0 1 0 0 1 1
Vector reset signal	**Sample time** = 1 **Samples per frame** = 8 **Rst** = `[0 0 0 1 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 8	

For the no reset case, the sequence is output without being reset. For the scalar and vector reset signal cases, the reset signal `[0 0 0 1 0 0 0 0]` is input to the `Rst` port. The sequence output is reset at the fourth bit, because the fourth bit of the reset signal is a 1 and the **Sample time** is 1.

For variable-sized outputs, the block only supports scalar reset signal inputs.

The “Gold Sequence Generator Reset Behavior” example demonstrates the reset behavior in a Simulink model.

Preferred Pairs of Sequences

The requirements for a pair of sequences u, v of period $N = 2^n–1$ to be a preferred pair are as follows:

- n is not divisible by 4
- $v = u[q]$, where
  - q is odd
  - $q = 2^k+1$ or $q = 2^{2k}–2^k+1$
  - v is obtained by sampling every qth symbol of u
- $\gcd(n, k) = \begin{cases} 1 & n \equiv 1 \mathrm{mod} 2 \\ 2 & n \equiv 2 \mathrm{mod} 4 \end{cases}$

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Gold Sequence Generator block version available before R2015b.

Existing models automatically update to load the Gold Sequence Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

References

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

[2] Gold, R. “Maximal Recursive Sequences with 3-Valued Recursive Cross-Correlation Functions (Corresp.).” *IEEE Transactions on Information Theory* 14, no. 1 (January 1968): 154–56. https://doi.org/10.1109/TIT.1968.1054106.

[3] Gold, R. “Optimal Binary Sequences for Spread Spectrum Multiplexing (Corresp.).” *IEEE Transactions on Information Theory* 13, no. 4 (October 1967): 619–21. https://doi.org/10.1109/TIT.1967.1054048.

[4] Sarwate, D.V., and M.B. Pursley, "Crosscorrelation Properties of Pseudorandom and Related Sequences," *Proc. IEEE* , Vol. 68, No. 5, May, 1980, pp. 583-619.

[5] Dixon, Robert C. *Spread Spectrum Systems: With Commercial Applications*. 3rd ed. New York: Wiley, 1994.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Kasami Sequence Generator | PN Sequence Generator

Objects
`comm.GoldSequence`

Introduced before R2006a

Hadamard Code Generator

Generate Hadamard code from orthogonal set of codes

Library

Sequence Generators sublibrary of Comm Sources

Description

The Hadamard Code Generator block generates a Hadamard code from a Hadamard matrix, whose rows form an orthogonal set of codes. Orthogonal codes can be used for spreading in communication systems in which the receiver is perfectly synchronized with the transmitter. In these systems, the despreading operation is ideal, as the codes are decorrelated completely.

The Hadamard codes are the individual rows of a Hadamard matrix. Hadamard matrices are square matrices whose entries are +1 or -1, and whose rows and columns are mutually orthogonal. If *N* is a nonnegative power of 2, the*N*-by-*N* Hadamard matrix, denoted H_N, is defined recursively as follows.

$$H_1 = [1]$$

$$H_{2N} = \begin{bmatrix} H_N & H_N \\ H_N & -H_N \end{bmatrix}$$

The N-by-N Hadamard matrix has the property that

$H_N H_N^T = NI_N$

where I_N is the *N*-by-*N* identity matrix.

The Hadamard Code Generator block outputs a row of H_N. The output is bipolar. You specify the length of the code, *N*,by the **Code length** parameter. The **Code length** must be a power of 2. You specify the index of the row of the Hadamard matrix, which is an integer in the range [0, 1, ... , *N*-1], by the **Code index** parameter.

Parameters

Code length

A positive integer that is a power of two specifying the length of the Hadamard code.

Code index

An integer between 0 and *N*-1, where *N* is the **Code length**, specifying a row of the Hadamard matrix.

Sample time

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-368.

Samples per frame

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-368.

Output data type

The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

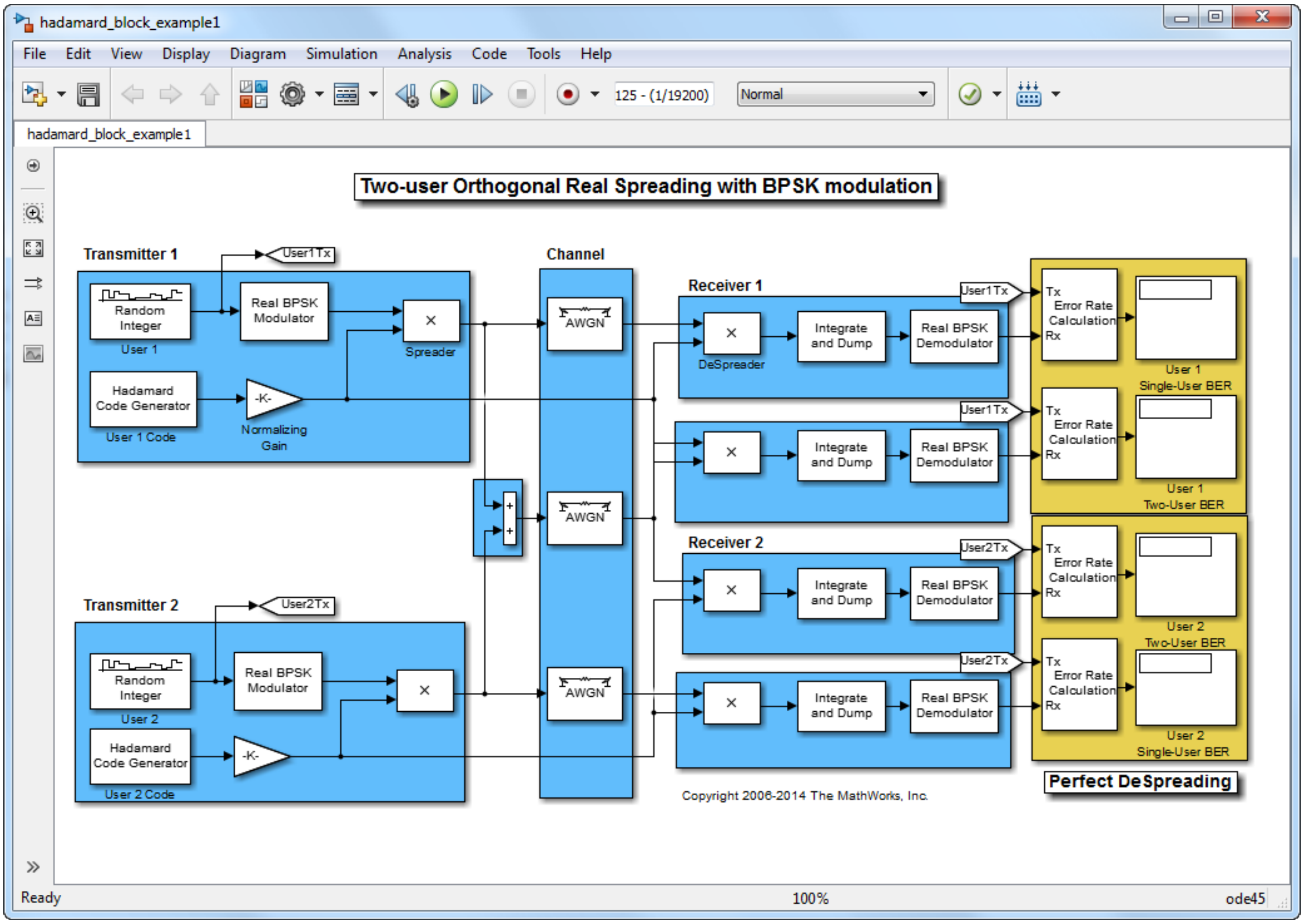
Examples

Orthogonal Spreading - Single-User vs. Two-User Comparison

This example model compares a single-user system vs. a two-user data transmission system with the two data streams being independently spread by different orthogonal codes.

The model uses random binary data which is BPSK modulated (real), spread by Hadamard codes of length 64 and then transmitted over an AWGN channel. The receiver consists of a despreader followed by a BPSK demodulator. Open the model here: hadamard_block_example1.

```
modelname = 'hadamard_block_example1';
open_system(modelname);
sim(modelname);
```



For the same data and channel settings, the model calculates the performance for one- and two-user transmissions.

Note that for the individual users, the error rates are exactly the same in both cases. This shows that perfect despreading is possible due to the ideal cross-correlation properties of the Hadamard codes.

To experiment with this model further, specify a different **Code length** or **Code index** for the individual users to examine the variations in relative performance.

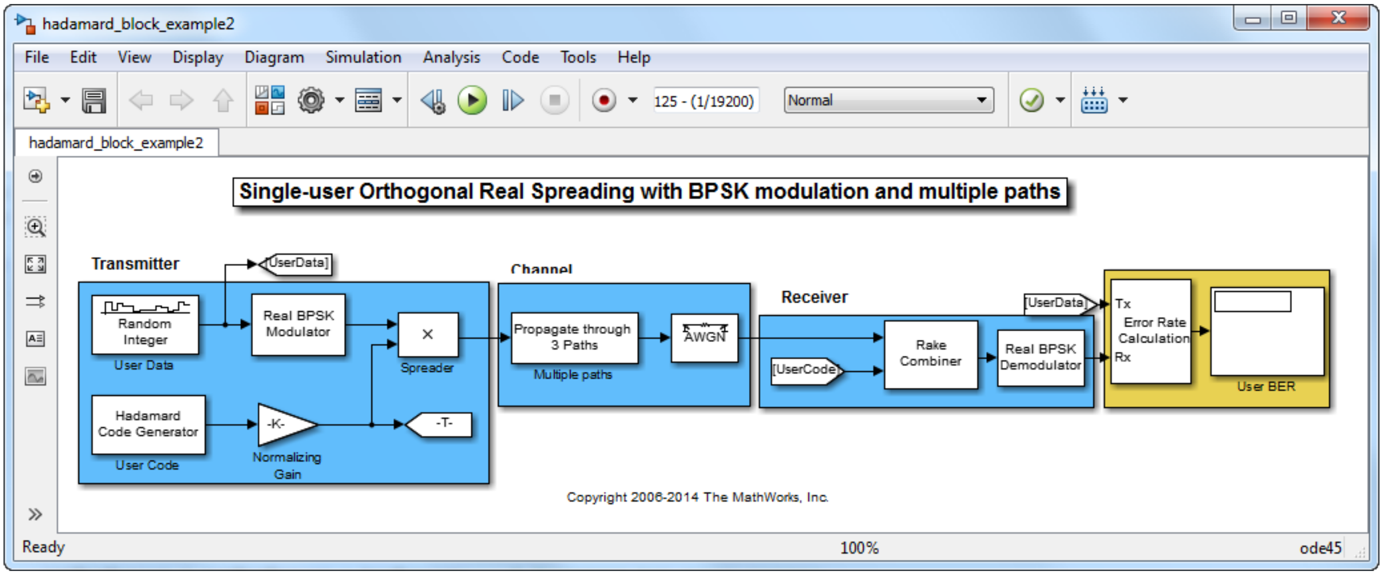
```
close_system(modelname, 0);
```

Orthogonal Spreading - Multipath Scenario

This example model considers a single-user system in which the signal is transmitted over multiple paths. This is similar to a mobile channel environment where the signals are received over multiple paths, each of which have different amplitudes and delays. To take advantage of the multipath transmission, the receiver employs diversity reception which combines the independent paths coherently.

Note, to keep the system simple, no shadowing effects are considered and the receiver has *a priori* knowledge of the number of paths and their respective delays. Open the model here: hadamard_block_example2.

```
modelname = 'hadamard_block_example2';
open_system(modelname);
sim(modelname);
```



For the data transmission with the same spreading code that was used in the first example, we now see deterioration in performance when compared with that example (compare the 180 errors with 81 in the previous case). This can be attributed to the non-ideal auto-correlation values of the orthogonal spreading codes chosen, which prevents perfect resolution of the individual paths. Consequently, we don't see the merits of diversity combining.

To experiment with this model further, try selecting other path delays to see how the performance varies for the same code. Also try different codes with the same delays.

```
close_system(modelname, 0);
```

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Hadamard Code Generator block version available before R2015b.

Existing models automatically update to load the Hadamard Code Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the

R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Does not support integer only code generation.

See Also

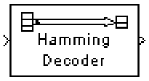
Blocks

OVSF Code Generator | Walsh Code Generator

Introduced before R2006a

Hamming Decoder

Decode Hamming code to recover binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Hamming Decoder block recovers a binary message vector from a binary Hamming codeword vector. For proper decoding, the parameter values in this block should match those in the corresponding Hamming Encoder block.

If the Hamming code has message length *K* and codeword length *N*, then *N* must have the form 2^M-1 for some integer *M* greater than or equal to 3. Also, *K* must equal *N*-*M*.

This block accepts a column vector input signal of length *N*. The output signal is a column vector of length *K*.

The coding scheme uses elements of the finite field GF(2^M). You can either specify the primitive polynomial that the algorithm should use, or you can rely on the default setting:

- To use the default primitive polynomial, simply enter *N* and *K* as the first and second dialog parameters, respectively. The algorithm uses `gfprimdf(M)` as the primitive polynomial for GF(2^M).
- To specify the primitive polynomial, enter *N* as the first parameter and a binary vector as the second parameter. The vector represents the primitive polynomial by listing its coefficients in order of ascending exponents. You can create primitive polynomials using the Communications Toolbox `gfprimfd` function.
- In addition, you can specify the primitive polynomial as a character vector, for example, `'D^3 + D + 1'`.

For information about the data types each block port supports, see the “Supported Data Type” on page 5-371 table on this page.

Parameters

Codeword length N

The codeword length *N*, which is also the input vector length.

Message length K, or M-degree primitive polynomial

The message length, which is also the input vector length or a binary vector that represents a primitive polynomial for GF(2^M) or a polynomial character vector.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • Fixed-point

Pair Block

Hamming Encoder

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Hamming Encoder

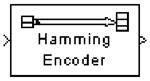
Functions

`hammgen`

Introduced before R2006a

Hamming Encoder

Create Hamming code from binary vector data



Library

Block sublibrary of Error Detection and Correction

Description

The Hamming Encoder block creates a Hamming code with message length *K* and codeword length *N*. The number *N* must have the form 2^M-1, where *M* is an integer greater than or equal to 3. Then *K* equals *N*-*M*.

This block accepts a column vector input signal of length *K*. The output signal is a column vector of length *N*.

The coding scheme uses elements of the finite field GF(2^M). You can either specify the primitive polynomial that the algorithm should use, or you can rely on the default setting:

- To use the default primitive polynomial, simply enter *N* and *K* as the first and second dialog parameters, respectively. The algorithm uses `gfprimdf(M)` as the primitive polynomial for GF(2^M).
- To specify the primitive polynomial, enter *N* as the first parameter and a binary vector as the second parameter. The vector represents the primitive polynomial by listing its coefficients in order of ascending exponents. You can create primitive polynomials using the Communications Toolbox `gfprimfd` function.
- In addition, you can specify the primitive polynomial as a character vector, for example, `'D^3 + D + 1'`.

For information about the data types each block port supports, see the “Supported Data Type” on page 5-373 table on this page.

Parameters

Codeword length N

The codeword length, which is also the output vector length.

Message length K, or M-degree primitive polynomial

The message length, which is also the input vector length or a binary vector that represents a primitive polynomial for GF(2^M) or a polynomial character vector.

Supported Data Type

Port	Supported Data Types
In	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point
Out	- Double-precision floating point - Single-precision floating point - Boolean - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers - Fixed-point

Pair Block

Hamming Decoder

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Hamming Decoder

Functions

`hammgen`

Introduced before R2006a

Helical Deinterleaver

Restore ordering of symbols permuted by helical interleaver

Library

Convolutional sublibrary of Interleaving

Description

The Helical Deinterleaver block permutes the symbols in the input signal by placing them in an array row by row and then selecting groups in a helical fashion to send to the output port.

The block uses the array internally for its computations. If *C* is the **Number of columns in helical array** parameter, then the array has *C* columns and unlimited rows. If *N* is the **Group size** parameter, then the block accepts an input of length $C{\cdot}N$ at each time step and inserts them into the next *N* rows of the array. The block also places the **Initial condition** parameter into certain positions in the top few rows of the array (not only to accommodate the helical pattern but also to preserve the vector indices of symbols that pass through the Helical Interleaver and Helical Deinterleaver blocks in turn).

The output consists of consecutive groups of *N* symbols. Counting from the beginning of the simulation, the block selects the kth output group in the array from column *k* mod *C*. The selection is helical because of the reduction modulo C and because the first symbol in the k^{th} group is in row 1+(*k*-1)*s, where *s* is the **Helical array step size** parameter.

This block accepts a column vector input signal containing $C{\cdot}N$ elements.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

Delay of Interleaver-Deinterleaver Pair

After processing a message with the Helical Interleaver block and the Helical Deinterleaver block, the deinterleaved data lags the original message by

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

samples. Before this delay elapses, the deinterleaver output is either the **Initial condition** parameter in the Helical Deinterleaver block or the **Initial condition** parameter in the Helical Interleaver block.

If your model incurs an additional delay between the interleaver output and the deinterleaver input, then the restored sequence lags the original sequence by the sum of the additional delay and the amount in the formula above. For proper synchronization, the delay between the interleaver and deinterleaver must be m$\cdot$ $C \cdot N$ for some nonnegative integer *m*. You can use the DSP System Toolbox Delay block to adjust delays manually, if necessary.

Parameters

Number of columns in helical array

The number of columns, *C*, in the helical array.

Group size

The size, *N*, of each group of symbols. The input width is *C* times *N*.

Helical array step size

The number of rows of separation between consecutive output groups as the block selects them from their respective columns of the helical array.

Initial conditions

A scalar that fills the array before the first input is placed.

Pair Block

Helical Interleaver

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Multiplexed Deinterleaver | Helical Interleaver

Introduced before R2006a

Helical Interleaver

Permute input symbols using helical array

Library

Convolutional sublibrary of Interleaving

Description

The Helical Interleaver block permutes the symbols in the input signal by placing them in an array in a helical fashion and then sending rows of the array to the output port.

The block uses the array internally for its computations. If *C* is the **Number of columns in helical array** parameter, then the array has *C* columns and unlimited rows. If *N* is the **Group size** parameter, then the block accepts an input of length $C \cdot N$ at each time step and partitions the input into consecutive groups of *N* symbols. Counting from the beginning of the simulation, the block places the k^{th} group in the array along column *k* mod *C*. The placement is helical because of the reduction modulo *C* and because the first symbol in the k^{th} group is in row $1+(k\text{-}1)\cdot s$, where *s* is the **Helical array step size** parameter. Positions in the array that do not contain input symbols have default contents specified by the **Initial condition** parameter.

The block sends $C \cdot N$ symbols from the array to the output port by reading the next *N* rows sequentially. At a given time step, the output symbols might be the **Initial condition** parameter value, symbols from that time step's input vector, or symbols left in the array from a previous time step.

This block accepts a column vector input signal containing $C \cdot N$ elements.

The block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The data type of this output will be the same as that of the input signal.

Parameters

Number of columns in helical array

The number of columns, C, in the helical array.

Group size

The size, N, of each group of input symbols. The input width is C times N.

Helical array step size

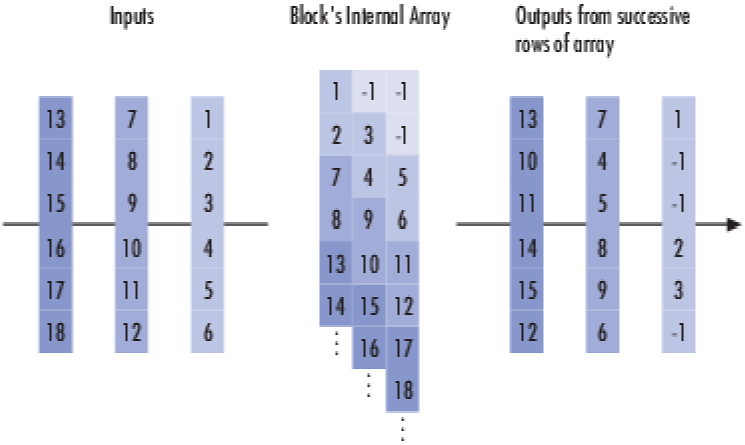
The number of rows of separation between consecutive input groups in their respective columns of the helical array.

Initial conditions

A scalar that fills the array before the first input is placed.

Examples

Suppose that C = 3, N = 2, the **Helical array step size** parameter is 1, and the **Initial condition** parameter is -1. After receiving inputs of `[1:6]'`, `[7:12]'`, and `[13:18]'`, the block's internal array looks like the schematic below. The coloring of the inputs and the array indicate how the input symbols are placed within the array. The outputs at the first three time steps are `[1; -1; -1; 2; 3; -1]`, `[7; 4; 5; 8; 9; 6]`, and `[13; 10; 11; 14; 15; 12]`. (The outputs are not color-coded in the schematic.)



Pair Block

Helical Deinterleaver

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Multiplexed Interleaver | Helical Deinterleaver

Introduced before R2006a

Ideal Rectangular Pulse Filter

Shape input signal using ideal rectangular pulses

Library

Comm Filters

Description

The Ideal Rectangular Pulse Filter block upsamples and shapes the input signal using rectangular pulses. The block replicates each input sample N times, where N is the **Pulse length** parameter. After replicating input samples, the block can also normalize the output signal and/or apply a linear amplitude gain.

If the **Pulse delay** parameter is nonzero, then the block outputs that number of zeros at the beginning of the simulation, before starting to replicate any of the input values.

This block accepts a scalar, column vector, or matrix input signal. For information about the data types each block port supports, see the “Supported Data Type” on page 5-381 table on this page.

The vector size, the pulse length, and the pulse delay are mutually independent. They do not need to satisfy any conditions with respect to each other.

Single-Rate Processing

When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To generate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is L times larger than that of the input ($M_o = M_i*L$), where L is the **Pulse length (number of samples)** parameter value.

Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size. However, the sample rate of the output is L times faster than that of the input (i.e. the output sample time is 1/N times the input sample time). When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an M-by-N matrix input as $M*N$ independent channels, and processes each channel over time. The output sample period (T_{so}) is L times shorter than the input sample period ($T_{so} = T_{si}/L$), while the input and output sizes remain identical.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-N matrix input as N independent channels. The block processes each column of the input over time by keeping the frame size constant (M_i=M_o), while making the output frame period (T_{fo}) L times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

Normalization Methods

You determine the block's normalization behavior using the **Normalize output signal** and **Linear amplitude gain** parameters.

- If you clear **Normalize output signal**, then the block multiplies the set of replicated values by the **Linear amplitude gain** parameter. This parameter must be a scalar.
- If you select **Normalize output signal**, then the **Normalization method** parameter appears. The block scales the set of replicated values so that one of these conditions is true:
  - The sum of the samples in each pulse equals the original input value that the block replicated.
  - The energy in each pulse equals the energy of the original input value that the block replicated. That is, the sum of the squared samples in each pulse equals the square of the input value.

After the block applies the scaling specified in the **Normalization method** parameter, it multiplies the scaled signal by the constant scalar value specified in the **Linear amplitude gain** parameter.

The output is scaled by $\sqrt{N}$. If the output of this block feeds the input to the AWGN Channel block, specify the AWGN signal power parameter to be 1/N.

Parameters

Pulse length (number of samples)

The number of samples in each output pulse; that is, the number of times the block replicates each input value when creating the output signal.

Pulse delay (number of samples)

The number of zeros that appear in the output at the beginning of the simulation, before the block replicates any input values.

Input processing

Specify how the block processes the input signal. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should upsample and shape the input signal. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and processes the signal by increasing the output frame size by a factor of *L*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block processes the signal such that the output sample rate is *L* times faster than the input sample rate.

Normalize output signal

If you select this, then the block scales the set of replicated values before applying the linear amplitude gain.

Normalization method

The quantity that the block considers when scaling the set of replicated values. Choices are `Sum of samples` and `Energy per pulse`. This field appears only if you select **Normalize method**.

Linear amplitude gain

A positive scalar used to scale the output signal.

Rounding mode

Use this parameter to specify the rounding method to be used when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`.

For more information, see Rounding Modes or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” in *DSP System Toolbox Reference Guide* for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” in *DSP System Toolbox Reference Guide* for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this check box to prevent any fixed-point scaling you specify in the block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Examples

If **Pulse length** is 4 and **Pulse delay** is the scalar 3, then the table below shows how the block treats the beginning of a ramp (1, 2, 3,...) in several situations. (The values shown in the table do not reflect vector sizes but merely indicate numerical values.)

Normalization Method, If Any	Linear Amplitude Gain	First Several Output Values
None (**Normalize output signal** cleared)	1	0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3,...
None (**Normalize output signal** cleared)	10	0, 0, 0, 10, 10, 10, 10, 20, 20, 20, 20, 30, 30, 30, 30,...
Sum of samples	1	0, 0, 0, 0.25, 0.25, 0.25, 0.25, 0.5, 0.5, 0.5, 0.5, 0.75, 0.75, 0.75, 0.75,..., where 0.25*4=1
Sum of samples	10	0, 0, 0, 2.5, 2.5, 2.5, 2.5, 5, 5, 5, 5, 7.5, 7.5, 7.5, 7.5, ...
Energy per pulse	1	0, 0, 0, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.5, 1.5, 1.5, 1.5,..., where (0.5)^2*4=1^2
Energy per pulse	10	0, 0, 0, 5, 5, 5, 5, 10, 10, 10, 10, 15, 15, 15, 15,...

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Integrate and Dump | Upsample

Introduced before R2006a

Insert Zero

(To be removed) Distribute input elements in output vector

Note will be removed in a future release. Use MATLAB® code in a MATLAB Function block instead. For more information, see “Compatibility Considerations”.

Library

Sequence Operations

Description

The Insert Zero block constructs an output vector by inserting zeros among the elements of the input vector. The input signal can be real or complex. Both the input signal and the **Insert zero vector** parameter are column vector signals. The number of 1s in the **Insert zero vector** parameter must be evenly divisible by the input data length. If the input vector length is greater than the number of 1s in the **Insert zero vector** parameter, then the block repeats the insertion pattern until it has placed all input elements in the output vector.

The block determines where to place the zeros by using the **Insert zero vector** parameter.

- For each 1 the block places the *next* element of the input vector in the output vector
- For each 0 the block places a 0 in the output vector

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

To implement punctured coding using the Puncture and Insert Zero blocks, use the same vector for the **Insert zero vector** parameter in this block and for the **Puncture vector** parameter in the Puncture block.

Parameters

Insert zero vector

A binary vector with a pattern of 0s and 1s that indicate where the block places either 0s or input vector elements in the output vector.

Compatibility Considerations

Insert Zero will be removed

Not recommended starting in R2020a

- Insert Zero will be removed in a future release. Use MATLAB code in a MATLAB Function block instead.
- This code can be used in a MATLAB Function block to insert zeros into a data stream.

  ```
  function y = fcn(u,insertZeroVector)
      numSeg = length(u)/sum(insertZeroVector);
      c = zeros(length(insertZeroVector), numSeg,'like',u);
      c(logical(insertZeroVector),:) = reshape(u,[],numSeg);
      y = c(:);
  end
  ```

  As with Insert Zero, the input length must be an integer multiple of the number of ones in the **Insert zero vector** parameter.
- For an example using this code, see “Insert Zeros into a Random Number Stream”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Puncture

Topics

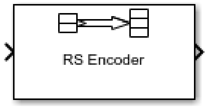
“Insert Zeros into a Random Number Stream”

Introduced before R2006a

Integer-Input RS Encoder

Create Reed-Solomon code from integer vector data

Library: Communications Toolbox / Error Detection and Correction / Block



Description

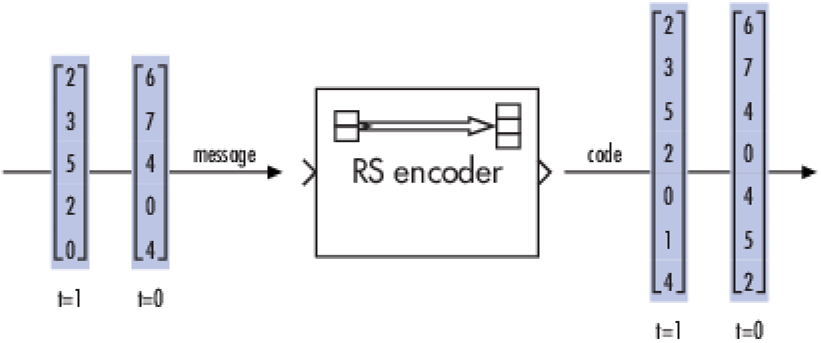
The Integer-Input RS Encoder block creates a Reed-Solomon code.

The symbols for the code are integers between 0 and 2^M-1, which represent elements of the finite field $GF(2^M)$. The default value of M is the smallest integer that is greater than or equal to log2(N+1), that is, `ceil(log2(N+1))`. You can change the default value of M by specifying the primitive polynomial for $GF(2^M)$, as described in “Specify the Primitive Polynomial” on page 5-389 below. Restrictions on M and N are described in “Restrictions on M and the Codeword Length N” on page 5-389.

The input and output are integer-valued signals that represent messages and codewords, respectively. For more information, see “Input and Output Signal Length in RS Blocks” on page 5-388.

An (N, K) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (not bit errors) in each codeword.

Suppose M = 3, N = 2^3-1 = 7, and K = 5. Then a message is a vector of length 5 whose entries are integers between 0 and 7. A corresponding codeword is a vector of length 7 whose entries are integers between 0 and 7. The following figure illustrates possible input and output signals to this block when **Codeword length N** is set to `7`, **Message length K** is set to `5`, and the default primitive and generator polynomials are used.



Ports

Input

In — Message
integer column vector

Message, specified as one of the following:

- When there is no message shortening, a (N_C×K)-by-1 integer column vector.
- When there is message shortening, a (N_C×S)-by-1 integer column vector.

N_C is the number of message words, K is the **Message length K**, and S is the **Shortened message length S**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-388.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

Output

`Out` — Reed-Solomon codeword

integer column vector

Reed-Solomon codeword, returned as an (N_C×(N – K + S – P)-by-1 integer column vector. N_C is the number of codewords, N is the **Codeword length N**, K is the **Message length K**, S is the **Shortened message length S**, P is the number of punctures per codeword.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-388.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32`

For more information, see “Supported Data Types” on page 5-390.

Parameters

`Codeword length N` — Codeword length

7 (default) | integer

Codeword length, specified as an integer.

For more information, see “Restrictions on M and the Codeword Length N” on page 5-389 and “Input and Output Signal Length in RS Blocks” on page 5-388.

`Message length K` — Message word length

3 (default) | integer

Message word length, specified as an integer in the range [1, N–2], where N is the codeword length.

`Shortened message length S` — Shortened message word length

3 (default) | integer

Shortened message word length, specified as an integer, such that $S \leq K$. When **Shortened message length S < Message length K**, the Reed-Solomon code is shortened.

You still specify N and K values for the full-length (N, K) code but the decoding is shortened to an (N–K+S, S) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values in the range [0 to 2^M–1], in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.
- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see “Specify the Generator Polynomial” on page 5-389.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Specify the Primitive Polynomial” on page 5-389.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 5-390.

Note If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

Dependencies

To enable this parameter, select **Puncture code**.

Block Characteristics

Data Types	`double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, K, or shortened message word length, S. The codeword length is $N - K + S - P$, where N is the full codeword length and P is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N - P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that y is an integer multiple of x.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Integer-Input RS Encoder	**Input Length (symbols):**
	$N_C \times K$
	Output Length (symbols):
	$N_C \times (N–P)$
		Input Length (symbols):
		$N_C \times S$
		Output Length (symbols):
		$N_C \times (N–K+S–P)$

Integer-Output RS Decoder	**Input Length (symbols):**
	$N_C \times (N–P)$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (symbols):
	$N_C \times K$
		Input Length (symbols):
		$N_C \times (N–K+S–P)$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (symbols):
		$N_C \times S$

- *N* is the codeword length.
- *K* is the message word length.
- *S* is the shortened message word length.
- N_C is the number of codewords (and message words).
- *P* is the number of punctures, and is equal to the number of zeros in the puncture vector.
- *M* is the degree of the primitive polynomial. Each group of *M* bits represents an integer between `0` and $2^M–1$ that belongs to the finite Galois field `GF(`2^M`)`.

For more information on representing data for Reed-Solomon codes, see “Integer Format (Reed-Solomon Only)”.

Restrictions on M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, *N*, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, *M*, are from 3 to 16. The valid values for *N* in this case are from 7 to $2^M–1$. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field $GF(2^M)$, which corresponds to the values that form message words and codewords.

Specify the Primitive Polynomial

You can specify the primitive polynomial that defines the finite field $GF(2^M)$, corresponding to the integers that form messages and codewords. To do so, first select **Specify primitive polynomial**. Then, in the **Primitive polynomial** text box, enter a binary row vector that represents a primitive polynomial over $GF(2^M)$, in descending order of powers. For example, to specify the polynomial x^3+x+1, enter the vector `[1 0 1 1]`.

If you do not select **Specify primitive polynomial**, the block uses the default primitive polynomial of degree M = ceil(log2(N+1)). You can display the default polynomial by entering `primpoly(ceil(log2(N+1)))` at the MATLAB prompt.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 2^M-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of $GF(2^M)$ represented in integer format. For more information about integer and binary format, see “Integer Format (Reed-Solomon Only)”. The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})...(x+\alpha^{b+N-K-1})$$

α is the primitive element of the Galois field over which the input message is defined, and *b* is an integer.

If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to *b*=1, for Reed-Solomon encoding. You can display the default generator polynomial by running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.
- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

Integer-Output RS Decoder

Algorithms

This object implements the algorithm, inputs, and outputs described in “Algorithms for BCH and RS Errors-only Decoding”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Binary-Input RS Encoder | Integer-Output RS Decoder

Objects

`comm.RSEncoder`

Functions

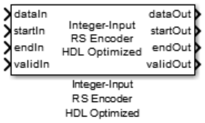
`primpoly` | `rsenc` | `rsgenpoly`

Introduced before R2006a

Integer-Input RS Encoder HDL Optimized

Encode data using a Reed-Solomon encoder

Library: Communications Toolbox HDL Support / Error Detection and Correction / Block
Communications Toolbox / Error Detection and Correction / Block



Description

The Integer-Input RS Encoder HDL Optimized decodes data using the RS encoder. The RS encoding follows the same standards as any other cyclic redundancy code. Use this block to model communications system forward error correction (FEC) codes.

For more about the RS encoder, see the Integer-Input RS Encoder block. For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

Ports

Input

`dataIn` — Input data
scalar

Input data, specified as a scalar representing one symbol. For binary point scaling, the input data type must be an integer or `fixdt`. The word length of each symbol must be equal to `ceil`($\log_2$(**Codeword length (N)**) + 1. The `double` data type is allowed for simulation, but not for HDL code generation.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

`startIn` — Start of input frame indicator
scalar

Start of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`endIn` — End of input frame indicator
scalar

End of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`validIn` — Valid input data indicator
scalar

Valid input data indicator, specified as a Boolean scalar.

This is a control signal that indicates if the data on the **dataIn** port is valid.

Data Types: `Boolean`

Output

`dataOut` — Output data
scalar

Output data, returned as a scalar. This output data width is the same size as the input data.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

`startOut` — Start of output frame indicator
scalar

Start of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`endOut` — End of output frame indicator
scalar

End of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`validOut` — Valid output data indicator
scalar

Valid output data indicator, returned as a Boolean scalar.

This is a control signal that indicates if the data on the **dataOut** port is valid.

Data Types: `Boolean`

Parameters

`Codeword length` — Length of codeword
`7` (default) | range from 7 to 65, 535

Specify the codeword length.

The codeword length `N` must be an integer equal to 2^M – 1, where M is an integer in the range from 3 to 16. For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

`Message length` — Length of message
`3` (default) | positive integer

Specify the length of the message.

For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

Each input frame, that is, the number of valid data samples between **startIn** and **endIn** port values, must contain more than *N*– *K* symbols and less than or equal to *K* symbols. A shortened code is inferred anytime the number of input data samples in a frame is less than *K*.

`Source of primitive polynomial` — Primitive polynomial source

`Auto` (default) | `Property`

Specify the source of the primitive polynomial.

- Select `Auto` to specify the primitive polynomial based on the **Codeword length (N)** parameter value. The degree of the primitive polynomial is calculated as *M* = `ceil`($\log_2$(**Codeword length (N)**). .
- Select `Property` to specify the primitive polynomial using the **Primitive polynomial** parameter.

`Primitive polynomial` — Primitive polynomial provider

`[ 1 0 1 1 ]` (default) | binary row vector

Specify a binary row vector representing the primitive polynomial in descending order of powers.

For more information on how to specify a primitive polynomial, see “Primitive Polynomials and Element Representations”.

Dependencies

To enable this parameter, set the **Source of primitive polynomial** parameter to `Property`.

`Source of puncture pattern` — Puncture pattern source

`None` (default) | `Property`

Select `Property` to enable the **Puncture pattern vector** parameter.

`Puncture pattern vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Specify a column vector of length *N*– *K*. In a puncture vector, a value of `1` represents that the data symbol passes unaltered. A value of `0` represents that the data symbol is punctured, or removed, from the data stream.

Dependencies

To enable this parameter, set the **Source of puncture pattern** parameter to `Property`.

`Source of B, the starting power for roots of the primitive polynomial` — Starting power for roots of primitive polynomial

`Auto` (default) | `Property`

Specify the source of the starting power for roots of the primitive polynomial.

- Select `Property` to enable the **B value** parameter.
- Select `Auto`, to use the **B value** parameter default value of `1`.

`B value` — Starting exponent of roots

`1` (default) | positive integer

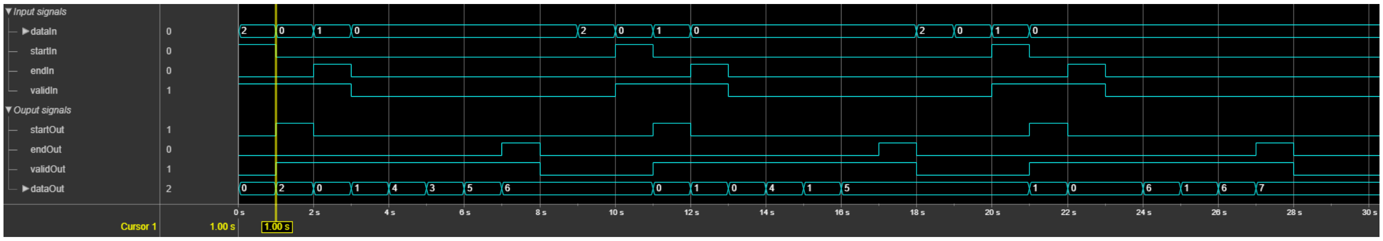
The starting exponent of the roots.

Dependencies

To enable this parameter, set the **Source of B, the starting power for roots of the primitive polynomial** parameter to `Property`.

Algorithms

This figure shows a sample output of the Integer-Input RS Encoder HDL Optimized block with a default configuration.



Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

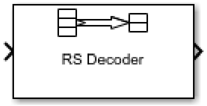
Integer-Input RS Encoder | Integer-Output RS Decoder HDL Optimized | `comm.HDLRSEncoder`

Introduced in R2012b

Integer-Output RS Decoder

Decode Reed-Solomon code to recover integer vector data

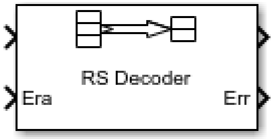
Library: Communications Toolbox / Error Detection and Correction / Block



Description

The Integer-Output RS Decoder block recovers a message vector from a Reed-Solomon codeword vector. For proper decoding, the parameter values in this block must match those in the corresponding Integer-Input RS Encoder block.

The Reed-Solomon code has message length *K*, and codeword length *N* – *number of punctures*. You specify *N* and *K* directly in the block dialog. The symbols for the code are integers in the range [0, 2^{M}-1], which represent elements of the finite field *GF*(2^{M}). Restrictions on *M* and *N* are described in “Restrictions on the M and the Codeword Length N” on page 5-402 below.



This icon shows optional ports.

The input and output are integer-valued signals that represent codewords and messages, respectively. For more information, see “Input and Output Signal Length in RS Blocks” on page 5-401. The block inherits the output data type from the input data type. For information about the data types each block port supports, see “Supported Data Types” on page 5-403.

For more information on representing data for Reed-Solomon codes, see the section “Integer Format (Reed-Solomon Only)”.

If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

The default value of M is `ceil(log2(N+1))`, that is, the smallest integer greater than or equal to log2(N+1). You can change the value of M from the default by specifying the primitive polynomial for GF(2^{M}), as described in “Specify the Primitive Polynomial” on page 5-402 below.

You can also specify the generator polynomial for the Reed-Solomon code, as described in “Specify the Generator Polynomial” on page 5-402.

An (*N*, *K*) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (*not* bit errors) in each codeword.

If decoding fails, the message portion of the decoder input is returned unchanged as the decoder output.

The sample times of the input and output signals are equal.

Ports

Input

`In` — Reed-Solomon codeword
integer column vector

Reed-Solomon codeword, specified as an (N_C×(N – K + S – P)-by-1 integer column vector. N_C is the number of codewords, N is the **Codeword length N**, K is the **Message length K**, S is the **Shortened message length S**, P is the number of punctures per codeword.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-401.

Data Types: `single` | `double` | `integer`

`Era` — Erasure vector
binary column vector

Erasure vector, specified as a binary column vector input signal with the same size as the input Reed-Solomon codeword.

Erasure values of 1 correspond to erased bits in the same position in the codeword. Values of 0 correspond to bits that are not erased. For more information, see “Puncturing and Erasures” on page 5-403.

Dependencies

To enable this port, select **Enable erasures input port**.

Data Types: `double` | `Boolean`

Output

`Out` — Decoded message
integer column vector

Decoded message, returned as one of the following:

- When there is no message shortening, a (N_C×K)-by-1 integer column vector.
- When there is message shortening, a (N_C×S)-by-1 integer column vector.

N_C is the number of message words, K is the **Message length K (symbols)**, and S is the **Shortened message length S (symbols)**.

Note The number of decoded message words equals the number of codewords.

For more information, see “Input and Output Signal Length in RS Blocks” on page 5-401.

`Err` — Decoding errors
integer vector

Symbol decoding errors, returned as an integer vector with N_C elements, where N_C is the number of codewords. This port indicates the number of symbol errors detected during decoding of each codeword. A negative integer indicates that the block detected more errors than it could correct by using the specified coding scheme.

Note An (*N*,*K*) Reed-Solomon code can correct up to `floor((N-K)/2)` symbol errors (not bit errors) in each codeword. When a received codeword contains more than (*N*-*K*)/2 symbol errors, a decoding failure occurs.

Dependencies

To enable this port, select **Output number of corrected symbol errors**.

Data Types: `double`

For more information, see “Supported Data Types” on page 5-403.

Parameters

`Codeword length N` — Codeword length

7 (default) | integer

Codeword length, specified as an integer.

For more information, see“Restrictions on the M and the Codeword Length N” on page 5-402 and “Input and Output Signal Length in RS Blocks” on page 5-401.

`Message length K` — Message word length

3 (default) | integer

Message word length, specified as an integer in the range [1, *N*–2], where *N* is the codeword length.

`Shortened message length S` — Shortened message word length

3 (default) | integer

Shortened message word length, specified as an integer, such that $S \leq K$. When **Shortened message length S < Message length K**, the Reed-Solomon code is shortened.

You still specify *N* and *K* values for the full-length (*N*, *K*) code but the decoding is shortened to an (*N*–*K*+*S*, *S*) code.

Dependencies

To enable this parameter, select **Specify shortened message length**.

`Generator polynomial` — Generator polynomial

`rsgenpoly(7, 3, [], [], 'double')` (default) | polynomial character vector | binary row vector | binary Galois row vector

Generator polynomial with values in the range [0, 2^M–1], in order of descending power, specified as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- An integer row vector that represents the coefficients of the generator polynomial in order of descending power.

- An integer Galois row vector that represents the coefficients of the generator polynomial in order of descending power.

Each coefficient is an element of the Galois field defined by the primitive polynomial. For more information, see “Specify the Generator Polynomial” on page 5-389.

Example: `[1 3 1 2 3]`, which is equivalent to `rsgenpoly(7,3)`

Dependencies

To enable this parameter, select **Specify generator polynomial**.

`Primitive polynomial` — Primitive polynomial

`'X^3 + X + 1'` (default) | polynomial character vector | binary row vector

Primitive polynomial in order of descending power. This polynomial is of order *M* and defines the finite Galois field GF(2^M) corresponding to the integers that form message words and codewords. Specify the primitive polynomial as one of the following:

- A polynomial character vector. For more information, see “Character Representation of Polynomials”.
- A binary row vector that represents the coefficients of the generator polynomial.

For more information, see “Specify the Primitive Polynomial” on page 5-389.

Example: `'X^3 + X + 1'`, which is the primitive polynomial used for a (7,3) code, `de2bi(primpoly(3,'nodisplay'),'left-msb')`

Dependencies

To enable this parameter, select **Specify primitive polynomial**.

`Puncture vector` — Puncture vector

`[ones(2,1); zeros(2,1)]` (default) | binary column vector

Puncture vector, specified as an (*N*–*K*)-by-1 binary column vector. Element indices with `1`s represent data symbol indices that pass through the block unaltered. Element indices with `0`s represent data symbol indices that get punctured, or removed, from the data stream. For more information, see “Puncturing and Erasures” on page 5-403.

Note If the encoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

Dependencies

To enable this parameter, select **Puncture code**.

`Enable erasures input port` — Enable erasures input port

off (default) | on

Selecting this check box enables the erasures port, **Era**. For more information, see “Puncturing and Erasures” on page 5-403.

`Output number of corrected symbol errors` — Enable port to output number of corrected symbol errors

off (default) | on

Selecting this check box enables an additional output port, **Err**, which indicates the number of symbol errors the block corrected in the input codeword.

Block Characteristics

Data Types	`double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Input and Output Signal Length in RS Blocks

The Reed-Solomon code has a message word length, *K*, or shortened message word length, *S*. The codeword length is $N – K + S – P$, where *N* is the full codeword length and *P* is the number of punctures per codeword. When there is no message shortening, the codeword length expression reduces to $N – P$, because $K = S$. If the decoder is processing multiple codewords per frame, then the same puncture pattern holds for all codewords.

This table provides expressions for the input and output signal lengths for the Reed-Solomon encoder and decoder.

The notation $y = N_C \times x$ denotes that *y* is an integer multiple of *x*.

	Input, Erasure, and Output Vector Lengths	
RS Block Coder	**No Message Shortening Used**	**Message Shortening Used**

Integer-Input RS Encoder	**Input Length (symbols):**
	$N_C \times K$
	Output Length (symbols):
	$N_C \times (N–P)$
		Input Length (symbols):
		$N_C \times S$
		Output Length (symbols):
		$N_C \times (N–K+S–P)$

Integer-Output RS Decoder	**Input Length (symbols):**
	$N_C \times (N–P)$
	Erasures Length (symbols):
	$N_C \times (N–P)$
	Output Length (symbols):
	$N_C \times K$
		Input Length (symbols):
		$N_C \times (N–K+S–P)$
		Erasures Length (symbols):
		$N_C \times (N–K+S–P)$
		Output Length (symbols):
		$N_C \times S$

- *N* is the codeword length.

- *K* is the message word length.
- *S* is the shortened message word length.
- N_C is the number of codewords (and message words).
- *P* is the number of punctures, and is equal to the number of zeros in the puncture vector.
- *M* is the degree of the primitive polynomial. Each group of *M* bits represents an integer between `0` and `2`^{M}`–1` that belongs to the finite Galois field `GF(2`^{M}`)`.

For more information on representing data for Reed-Solomon codes, see "Integer Format (Reed-Solomon Only)".

Restrictions on the M and the Codeword Length N

- If you do not select **Specify primitive polynomial**, valid values for the codeword length, *N*, are from 7 to 65535. In this case, the block uses the default primitive polynomial of degree `M = ceil(log2(N+1))`. You can display the default primitive polynomial by running `primpoly(ceil(log2(N+1)))`.
- If you select **Specify primitive polynomial**, valid values for the primitive polynomial degree, *M*, are from 3 to 16. The valid values for *N* in this case are from 7 to 2^{M}–1. Selecting **Specify primitive polynomial** enables you to specify the primitive polynomial that defines the finite field *GF*(2^{M}), which corresponds to the values that form message words and codewords.

Specify the Primitive Polynomial

You can specify the primitive polynomial that defines the finite field *GF*(2^{M}), corresponding to the integers that form messages and codewords. To do so, first select **Specify primitive polynomial**. Then, in the **Primitive polynomial** text box, enter a binary row vector that represents a primitive polynomial over *GF*(2^{M}), in descending order of powers. For example, to specify the polynomial x^3+x +1, enter the vector `[1 0 1 1]`.

If you do not select **Specify primitive polynomial**, the block uses the default primitive polynomial of degree M = ceil(log2(N+1)). You can display the default polynomial by entering `primpoly(ceil(log2(N+1)))` at the MATLAB prompt.

Specify the Generator Polynomial

Select **Specify generator polynomial** to enable the **Generator polynomial** parameter for specifying the generator polynomial of the Reed-Solomon code. Enter an integer row vector with element values from 0 to 2^{M}-1. The vector represents a polynomial, in descending order of powers, whose coefficients are elements of *GF*(2^{M}) represented in integer format. For more information about integer and binary format, see "Integer Format (Reed-Solomon Only)". The generator polynomial must be equal to a polynomial with this factored form:

$$g(x) = (x+\alpha^{b})(x+\alpha^{b+1})(x+\alpha^{b+2})...(x+\alpha^{b+N-K-1})$$

α is the primitive element of the Galois field over which the input message is defined, and *b* is an integer.

If you do not select **Specify generator polynomial**, the block uses the default generator polynomial, corresponding to *b*=1, for Reed-Solomon encoding. You can display the default generator polynomial by running `rsgenpoly`.

- If you are using the default primitive polynomial (**Specify primitive polynomial** is not selected), the default generator polynomial is `rsgenpoly(N,K)`, where `N = 2`^{M}`-1`.

- If you are not using the default primitive polynomial (**Specify primitive polynomial** is selected) and you specify the primitive polynomial as `poly`, the generator polynomial is `rsgenpoly(N,K,poly)`.

Note The degree of the generator polynomial is $N - K$, where N is the codeword length and K is the message word length.

Puncturing and Erasures

`1`s and `0`s have precisely opposite meanings for the puncture and erasure vectors.

In a puncture vector,

- `1` means that the data symbol is passed through the block unaltered.
- `0` means that the data symbol is to be punctured, or removed, from the data stream.

In an erasure vector,

- `1` means that the data symbol is to be replaced with an erasure symbol.
- `0` means that the data symbol is passed through the block unaltered.

These conventions apply to both the encoder and the decoder. For more information, see “Shortening, Puncturing, and Erasures”.

Supported Data Types

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Out	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Era	• Double-precision floating point • Boolean
Err	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • If the input is uint8, uint16, or uint32, then the number of errors output datatype is int8, int16, or int32, respectively.

Pair Block

Integer-Input RS Encoder

Algorithms

This block uses the Berlekamp-Massey decoding algorithm. For information about this algorithm, see “Algorithms for BCH and RS Errors-only Decoding”.

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.

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
Binary-Output RS Decoder

Objects
`comm.RSDecoder`

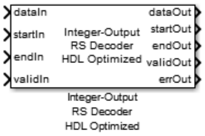
Functions
`primpoly` | `rsdec` | `rsgenpoly`

Introduced before R2006a

Integer-Output RS Decoder HDL Optimized

Decode data using Reed-Solomon (RS) decoder

Library: Communications Toolbox HDL Support / Error Detection and Correction / Block
Communications Toolbox / Error Detection and Correction / Block



Description

The Integer-Output RS Decoder HDL Optimized decodes data using RS decoder. The RS decoding follows the same standards as any other cyclic redundancy code. Use this block to model communications system forward error correction (FEC) codes.

For more information about the RS decoder, see the Integer-Output RS Decoder block. For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

Ports

Input

`dataIn` — Input data
scalar

Input data, specified as a scalar representing one symbol. For binary point scaling, the input data type must be an integer or `fixdt`. The `double` data type is allowed for simulation, but not for HDL code generation.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

`startIn` — Start of input frame indicator
scalar

Start of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`endIn` — End of input frame indicator
scalar

End of input frame indicator, specified as a Boolean scalar.

Data Types: `Boolean`

`validIn` — Valid input data indicator
scalar

Valid input data indicator, specified as a Boolean scalar.

This is a control signal that indicates if the data on the **dataIn** port is valid.

Data Types: `Boolean`

Output

`dataOut` — Decoded message data
scalar

Decoded message data, returned as a scalar. This output data width is the same size as the input data.

Data Types: `double` | `int8` | `int16` | `int32` | `int64` | `fixed point`

`startOut` — Start of output frame indicator
scalar

Start of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`endOut` — End of output frame indicator
scalar

End of output frame indicator, returned as a Boolean scalar.

Data Types: `Boolean`

`validOut` — Valid output data indicator
scalar

Valid output data indicator, returned as a Boolean scalar.

This is a control signal that indicates if the data on the **dataOut** port is valid.

Data Types: `Boolean`

`errOut` — Indications of corruption of received data
scalar

Indications of corruption of the received data, returned as a Boolean scalar.

Data Types: `Boolean`

`numErrors` — Number of detected errors
nonnegative scalar

Number of detected errors, returned as a nonnegative scalar.

Dependencies

To enable this port, select the **Output number of corrected symbol errors** parameter.

Data Types: `uint8`

Parameters

`Codeword length` — Length of codeword
7 (default) | range from 7 to 65, 535

Specify the codeword length.

The codeword length N must be an integer equal to 2^M – 1, where M is an integer in the range from 3 to 16. For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

Message length — Length of message

3 (default) | positive integer

Specify the length of the message.

For more information on representing data for RS codes, see “Integer Format (Reed-Solomon Only)”.

Source of primitive polynomial — Primitive polynomial source

Auto (default) | Property

Specify the source of the primitive polynomial.

- Select Auto to specify the primitive polynomial based on the **Codeword length (N)** parameter value. The degree of the primitive polynomial is calculated as M = ceil(log$_2$(**Codeword length (N)**). .
- Select Property to specify the primitive polynomial using the **Primitive polynomial** parameter.

Primitive polynomial — Primitive polynomial

[1 0 1 1] (default) | binary row vector

Specify a binary row vector representing the primitive polynomial in descending order of powers.

For more information on how to specify a primitive polynomial, see “Primitive Polynomials and Element Representations”.

Dependencies

To enable this parameter, set the **Source of primitive polynomial** parameter to Property.

Source of B, the starting power for roots of the primitive polynomial — Source of starting power for roots of primitive polynomial

Auto (default) | Property

Specify the source of the starting power for roots of the primitive polynomial.

- Select Property to enable the **B value** parameter.
- Select Auto, to use the **B value** parameter default value of 1.

B value — Starting exponent of roots

1 (default) | positive integer

The starting exponent of the roots.

Dependencies

To enable this parameter, set the **Source of B, the starting power for roots of the primitive polynomial** parameter to Property.

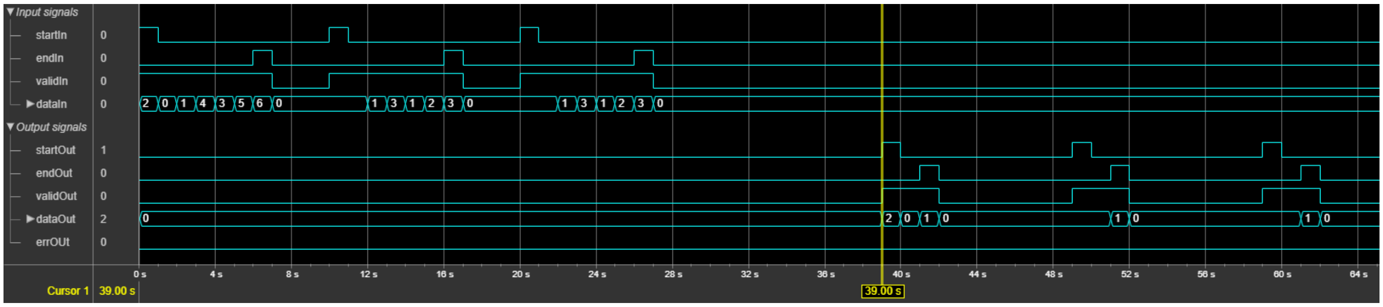
Output number of corrected symbol errors — Number of corrected symbol errors

off (default) | on

Select this parameter to enable the **numErrors** output port. This port outputs the detected symbol error count.

Algorithms

This figure shows a sample output of the Integer-Output RS Decoder HDL Optimized block with a default configuration.



Troubleshooting

- Each input frame must contain more than (*N*– *K*) x 2 symbols and less than or equal to *N* symbols. A shortened code is inferred when the number of valid data samples between **startIn** and **endIn** is less than *N*. A shortened code still requires *N* cycles to perform the Chien search. If the input is less than *N* symbols, leave a guard interval of at least `N-size` inactive cycles before starting the next frame.
- The decoder can operate on up to four messages at a time. If the block receives the start of a fifth message before completely decoding the first message, the block drops data samples from the first message. To avoid this issue, increase the number of inactive cycles between input messages.
- The generator polynomial is not specified explicitly. However, it is defined by the codeword length, message length, and the **B value** for the starting exponent of the roots.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Not recommended for production code.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.

See Also

Integer-Input RS Encoder HDL Optimized | Integer-Output RS Decoder | `comm.HDLRSDecoder`

Introduced in R2012b

Integer to Bit Converter

Map vector of integers to vector of bits
Library: Communications Toolbox / Utility Blocks

Description

The Integer to Bit Converter block maps each integer (or fixed-point value) in the input vector to a group of bits in the output vector.

This block is single-rate and single-channel. The block maps each integer value (or stored integer when you use a fixed point input) to a group of *M* bits, using the selection for the **Output bit order** to determine the most significant bit. The resulting output vector length is *M* times the input vector length.

Ports

Input

`In` — Input signal
integer | integer column vector

Input signal, specified as an integer or a length *N* column vector of integers.

If *M* is specified by the **Number of bits per integer(M)** parameter:

- When the **Number of bits per integer** parameter is set to `Unsigned`, input values must be integers in the range [0, (2^{M} – 1)].
- When the **Number of bits per integer** parameter is set to `Signed`, input values must be integers in the range [(–2^{M-1}), ($2^{M - 1}$ – 1)].

During simulation, the block performs a run-time check and issues an error if any input value is outside of the appropriate range. When the block generates code, it does not perform this run-time check.

Data Types: `double`

Output

`Out` — Output signal
bit scalar | bit column vector

Output signal, returned as a scalar or column vector of bits of length $M \cdot N$.

Parameters

`Number of bits per integer(M)` — Number of bits per integer

3 (default) | integer in the range [1, 32]

Number of input bits mapped to each integer in the input, specified as an integer in the range [1, 32].

`Treat input values as` — Treat input values as

`Unsigned` (default) | `Signed`

Indicate if the integer value input ranges should be treated as signed or unsigned.

`Output bit order` — Output bit order

`MSB first` (default) | `LSB first`

Define whether the first bit of the output signal is the most significant bit (MSB) or the least significant bit (LSB).

`Output data type` — Output data type

`Inherit via internal rule` (default) | `Smallest unsigned integer` | `Same as input` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `boolean`

Specify the data type of the output bits. You can choose one of the following **Output data type** options:

- `Inherit via internal rule` –– The block determines the output data type based on the input data type.
  - If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
  - If the input data type is not floating-point, the output data type is determined as if the parameter is set to `Smallest integer`.
- `Smallest integer` ––The block selects the output data type based on settings used in the “Hardware Implementation Pane” (Simulink) of the Configuration Parameters dialog box.
  - If you select `ASIC/FPGA` for the device vendor, the output data type is the ideal one-bit size (ufix1).
  - For all other device vendor selections, the output data type is an unsigned integer with the smallest available word length, as defined in the Hardware Implementation settings (for example, uint8)
- `Same as input`
- `double`
- `single`
- `uint8`
- `uint16`
- `uint32`

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[a] | `integer` | `single`

Multidimensional Signals	`no`
Variable-Size Signals	`yes`

a. ufix(1) only at the output when ASIC/FPGA is selected in the Hardware Implementation Pane.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Bit to Integer Converter on page 5-78

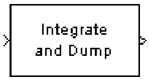
Functions

`de2bi` | `dec2bin`

Introduced before R2006a

Integrate and Dump

Integrate discrete-time signal, resetting to zero periodically



Library

Comm Filters

Description

The Integrate and Dump block creates a cumulative sum of the discrete-time input signal, while resetting the sum to zero according to a fixed schedule. When the simulation begins, the block discards the number of samples specified in the **Offset** parameter. After this initial period, the block sums the input signal along columns and resets the sum to zero every *N* input samples, where *N* is the **Integration period** parameter value. The reset occurs after the block produces its output at that time step.

Receiver models often use the integrate-and-dump operation when the system's transmitter uses a simple square-pulse model. Fiber optics and in spread-spectrum communication systems, such as CDMA (code division multiple access) applications, also use the operation.

This block accepts a scalar, column vector, or matrix input signal. When the input signal is not a scalar value, it must contain $k \cdot N$ rows for some positive integer *k*. For these input signals, the block processes each column independently.

Selecting **Output intermediate values** affects the contents, dimensions, and sample time as follows:

- If you clear the check box, then the block outputs the cumulative sum at each reset time.
  - If the input is a scalar value, then the output sample time is *N* times the input sample time and the block experiences a delay whose duration is one output sample period. In this case, the output dimensions match the input dimensions.
  - If the input is a ($k \cdot N$)-by-*n* matrix, then the output is *k*-by-*n*. In this case, the block experiences no delay and the output period matches the input period.
- If you select the check box, then the block outputs the cumulative sum at each time step. The output has the same sample time and the same matrix dimensions as the input.

Transients and Delays

A nonzero value in the **Offset** parameter causes the block to output one or more zeros during the initial period while it discards input samples. If the input is a matrix with *n* columns and the **Offset** parameter is a length-*n* vector, then the m^{th} element of the **Offset** vector is the offset for the m^{th} column of data. If **Offset** is a scalar, then the block applies the same offset to each column of data. The output of initial zeros due to a nonzero **Offset** value is a transient effect, not a persistent delay.

When you clear **Output intermediate values**, the block's output is delayed, relative to its input, throughout the simulation:

- If the input is a scalar value, then the output is delayed by one sample after any transient effect is over. That is, after removing transients from the input and output, you can see the result of the *m*th integration period in the output sample indexed by m+1.
- If the input is a column vector or matrix and the **Offset** parameter is nonzero, then after the transient effect is over, the result of each integration period appears in the output frame corresponding to the *last* input sample of that integration period. This is one frame later than the output frame corresponding to the first input sample of that integration period, in cases where an integration period spans two input frames. For an example of this situation, see "Example of Transient and Delay" on page 5-416.

Parameters

Integration period

The number of input samples between resets.

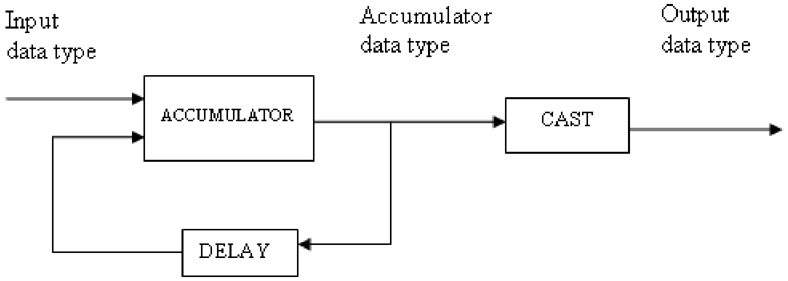
Offset

A nonnegative integer vector or scalar specifying the number of input samples to discard from each column of input data at the beginning of the simulation.

Output intermediate values

Determines whether the block outputs the intermediate cumulative sums between successive resets.

Fixed-Point Signal Flow Diagram



Fixed-Point Attributes

The settings for the following parameters only apply when block inputs are fixed-point signals.

Rounding mode

Use this parameter to specify the rounding method to be used when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result.

For more information, see "Rounding Modes" or "Rounding Mode: Simplest" (Fixed-Point Designer).

Saturate on integer overflow

Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result:

- Saturate represents positive overflows as the largest positive number in the range being used, and negative overflows as the largest negative number in the range being used.
- Wrap uses modulo arithmetic to cast an overflow back into the representable range of the data type. See “Modulo Arithmetic” (Fixed-Point Designer) for more information.

Accumulator—Mode

Use the **Accumulator—Mode** parameter to specify how you would like to designate the accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the block automatically calculates the accumulator output word and fraction lengths.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator.

Output

Use the **Output** parameter to choose how you specify the word length and fraction length of the output of the block:

- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, enter the word length, in bits, and the slope of the output.

For additional information about the parameters pertaining to fixed-point applications, see “Specify Fixed-Point Attributes for Blocks”.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Fixed-point

Examples

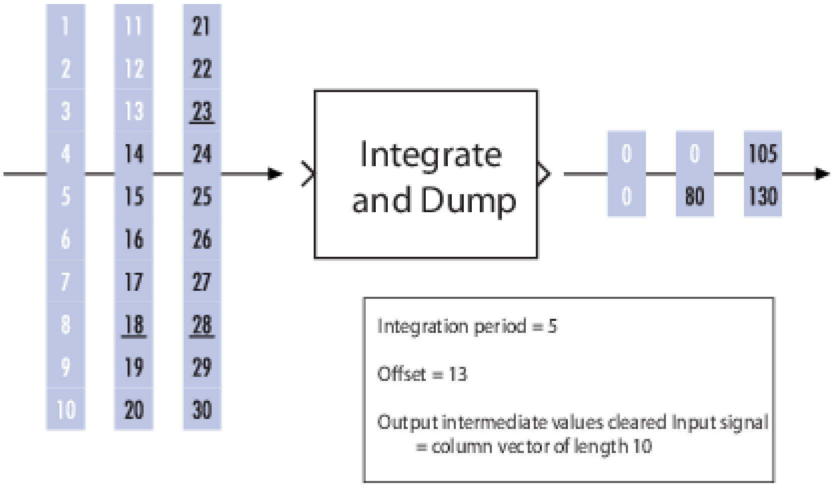
If **Integration period** is 4 and **Offset** is the scalar 3, then the table below shows how the block treats the beginning of a ramp (1, 2, 3, 4,...) in several situations. (The values shown in the table do not reflect vector sizes but merely indicate numerical values.)

Output intermediate values Check Box	Input Signal Properties	First Several Output Values
Cleared	Scalar	0, 0, 4+5+6+7, and 8+9+10+11, where one 0 is an initial transient value and the other 0 is a delay value that results from the cleared check box and scalar value input.
Cleared	Column vector of length 4	0, 4+5+6+7, and 8+9+10+11, where 0 is an initial delay value that results from the nonzero offset. The output is a scalar value.
Selected	Scalar	0, 0, 0, 4, 4+5, 4+5+6, 4+5+6+7, 8, 8+9, 8+9+10, 8+9+10+11, and 12, where the three 0s are initial transient values.
Selected	Column vector of length 4	0, 0, 0, 4, 4+5, 4+5+6, 4+5+6+7, 8, 8+9, 8+9+10, 8+9+10+11, and 12, where the three 0s are initial transient values. The output is a column vector of length 4.

In all cases, the block discards the first three input samples (1, 2, and 3).

Example of Transient and Delay

The figure below illustrates a situation in which the block exhibits both a transient effect for three output samples, as well as a one-sample delay in alternate subsequent output samples for the rest of the simulation. The figure also indicates how the input and output values are organized as column vectors. In each vector in the figure, the last sample of each integration period is underlined, discarded input samples are white, and transient zeros in the output are white.



The transient effect lasts for ceil(13/5) output samples because the block discards 13 input samples and the integration period is 5. The first output sample after the transient effect is over, 80, corresponds to the sum 14+15+16+17+18 and appears at the time of the input sample 18. The next output sample, 105, corresponds to the sum 19+20+21+22+23 and appears at the time of the input sample 23. Notice that the input sample 23 is one frame later than the input sample 19; that is, this five-sample integration period spans two input frames. As a result, the output of 105 is delayed compared to the first input (19) that contributes to that sum.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

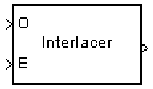
Blocks

Discrete-Time Integrator | Ideal Rectangular Pulse Filter | Windowed Integrator

Introduced before R2006a

Interlacer

Alternately select elements from two input vectors to generate output vector
Library: Communications Toolbox / Sequence Operations



Description

The Interlacer block accepts two vector with the size, complexity, and sample time. It produces one output vector by alternating elements from the first input (labeled O for odd) and the second input (labeled E for even) .

Ports

Input

O — Odd-numbered elements
vector

Odd-numbered elements, specified as a vector. The complexity of inputs O and E must match.

Data Types: double | single

E — Even-numbered elements
vector

Even-numbered elements, specified as a vector. The complexity of inputs O and E must match.

Data Types: double | single

Output

Out — Output signal
column vector

Output signal, returned as an even length column vector. Odd numbered elements in the output vector contain the elements from input O and even numbered elements in the output vector contain the elements from input E. The output vector has the same data type and sample time as the input.

Block Characteristics

Data Types	Boolean | double | enumerated | fixed point | integer | single
Multidimensional Signals	no
Variable-Size Signals	no

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Deinterlacer | General Block Interleaver | Mux

Introduced before R2006a

I/Q Imbalance

Apply I/Q imbalances to complex signal
Library: Communications Toolbox / RF Impairments

Description

The I/Q Imbalance block applies in-phase and quadrature imbalances to a complex signal. This block applies an amplitude imbalance, a phase imbalance, and a DC offset to the in-phase and quadrature signal components. For more information, see “I/Q Imbalance Implementation” on page 5-421 and “Algorithms” on page 5-421.

Ports

Input

`In1` — Complex signal
scalar | vector

Complex signal, specified as a scalar or vector.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Out1` — Output signal
scalar | vector

Output signal, returned as a scalar or vector. This output is the same dimension and data type as the input signal.

Parameters

`I/Q amplitude imbalance (dB)` — I/Q amplitude imbalance
`0` (default) | scalar

I/Q amplitude imbalance in decibels of signal power, specified as a scalar. For more information, see “Algorithms” on page 5-421.

`I/Q phase imbalance (deg)` — I/Q phase imbalance
`0` (default) | scalar

I/Q amplitude imbalance in degrees, specified as a scalar.

`I dc offset` — In-phase component DC offset
`0` (default) | scalar

In-phase component DC offset, specified as a scalar.

`Q dc offset` — Quadrature component DC offset

`0` (default) | scalar

Quadrature component DC offset, specified as a scalar.

Block Characteristics

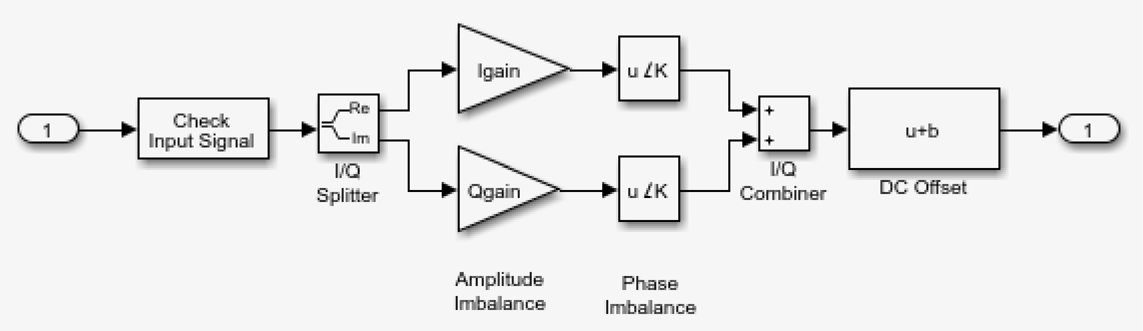
Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

I/Q Imbalance Implementation

The I/Q Imbalance block applies amplitude imbalance, phase imbalance, and DC offsets to the in-phase and quadrature components of the complex input signal.

The block performs these operations consecutively, as shown by the subsystem in this model diagram. You can view the subsystem by right-clicking the block and selecting **Mask > Look under mask**.



To apply the impairments, the block follows this workflow.

1. Separate the signal into its in-phase and quadrature components.
2. Apply amplitude imbalance, specified by the **I/Q amplitude imbalance (dB)** parameter.
3. Apply phase imbalance, specified by the **I/Q phase imbalance (deg)** parameter.
4. Recombine the in-phase and quadrature components into a complex signal.
5. Apply an in-phase DC offset, specified by the **I dc offset** parameter, and a quadrature DC offset, specified by the **Q dc offset** parameter, to the signal.

For more information, see “Algorithms” on page 5-421.

Algorithms

The I/Q amplitude imbalance, I/Q phase imbalance, and DC offset impairments are described sequentially in the section.

1. For an I/Q amplitude imbalance, I_{a}, the impairment is applied to the input signal, $x_{\mathrm{r}}+\mathrm{j}x_{\mathrm{i}}$ and $y_{\mathrm{AmplitudeImbalance}}$ is an intermediate output.

   $$y_{\mathrm{AmplitudeImbalance}} \triangleq y_{\mathrm{r}_{\mathrm{AmplitudeImbalance}}} + jy_{\mathrm{i}_{\mathrm{AmplitudeImbalance}}}$$

   $$y_{\mathrm{AmplitudeImbalance}} = \left(10^{(0.5I_{\mathrm{a}}/20)}x_{\mathrm{r}}\right) + j\left(10^{(-0.5I_{\mathrm{a}}/20)}x_{\mathrm{i}}\right)$$

2. For an I/Q phase imbalance, I_{p}, the impairment is applied to $y_{\mathrm{AmplitudeImbalance}}$ and $y_{\mathrm{PhaseImbalance}}$ is an intermediate output.

   $$y_{\mathrm{PhaseImbalance}} \triangleq y_{\mathrm{r}_{\mathrm{PhaseImbalance}}} + jy_{\mathrm{i}_{\mathrm{PhaseImbalance}}}$$

   $$y_{\mathrm{PhaseImbalance}} = \left(e\left(j\left(-0.5\pi\frac{Ip}{180}\right)\right)y_{\mathrm{r}_{\mathrm{AmplitudeImbalance}}}\right) + \left(e\left(j\left(\frac{\pi}{2} + 0.5\pi\frac{I\mathrm{p}}{180}\right)\right)y_{\mathrm{r}_{\mathrm{AmplitudeImbalance}}}\right)$$

3. For DC offsets, I_{DC} and Q_{DC}, the impairment is applied to $y_{\mathrm{PhaseImbalance}}$ and y is the final output.

   $$y = (y_{\mathrm{r}_{\mathrm{PhaseImbalance}}} + I_{\mathrm{DC}}) + j(y_{\mathrm{i}_{\mathrm{PhaseImbalance}}} + Q_{\mathrm{DC}})$$

Variables for these calculations are defined in this list.

- I_{a} is the I/Q amplitude imbalance.
- I_{p} is the I/Q phase imbalance.
- I_{DC} is the in-phase DC offset.
- Q_{DC} is the quadrature DC offset.
- x is the complex input signal and is given by $x_{\mathrm{r}} + jx_{\mathrm{i}}$.
  - x_{r} and x_{i} are the real and imaginary parts, respectively, of x.
- y is the complex output signal and is given by $y_{\mathrm{r}} + jy_{\mathrm{i}}$.
  - y_r and y_i are the real and imaginary parts, respectively, of y.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Free Space Path Loss | I/Q Compensator Coefficient to Imbalance | I/Q Imbalance Compensator | Memoryless Nonlinearity | Phase Noise | Receiver Thermal Noise

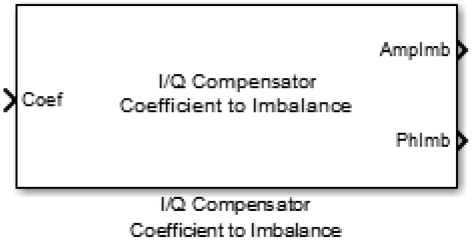
Functions

`iqimbal`

Introduced before R2006a

I/Q Compensator Coefficient to Imbalance

Convert compensator coefficient into amplitude and phase imbalance



Library

RF Impairments Correction

Description

The I/Q Compensator Coefficient to Imbalance block converts a compensator coefficient into its equivalent amplitude and phase imbalance.

This block has a single input port, which accepts a complex coefficient or a vector of coefficients. There are amplitude and phase imbalance output ports both of which are real. The amplitude imbalance is expressed in dB while the phase imbalance is expressed in degrees.

Algorithms

See the `iqcoef2imbal` function reference page for more information on the inputs, outputs, and algorithms.

Supported Data Types

Port	Supported Data Types
Compensator Coefficient	• Double-precision, complex floating point • Single-precision, complex floating point
Amplitude Imbalance (dB)	• Double-precision floating point • Single-precision floating point
Phase Imbalance (deg)	• Double-precision floating point • Single-precision floating point

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

I/Q Imbalance Compensator

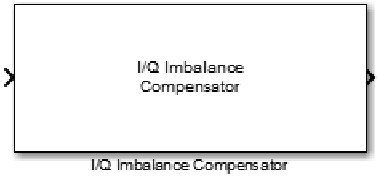
Functions

`iqcoef2imbal`

Introduced in R2014b

I/Q Imbalance Compensator

Compensate for imbalance between in-phase and quadrature components



Library

RF Impairments Correction

Description

The I/Q Imbalance Compensator mitigates the effects of an amplitude and phase imbalance between the in-phase and quadrature components of a modulated signal. The supported modulation schemes include OFDM, M-PSK, and M-QAM, where M > 2.

This block accepts up to three input ports, of which one is the input signal. When you set the **Source of compensator coefficient** parameter to `Estimated from input signal`, two additional input ports are enabled. The first is enabled when you set the **Source of adaptation step size** parameter to `Input port` and the second is enabled when you check the **Coefficient adaptation input port** box. The two options are independent. Additionally, you can check the **Estimated coefficient output port** box to create an optional output port from which the estimated compensator coefficients are made available.

When you set the **Source of compensator coefficient** parameter to `Input port`, only one possible configuration is possible (input signal port, coefficient input port, and output signal port).

Parameters

Source of compensator coefficient

Specify the source of the compensator coefficients as `Estimated from input signal` or `Input port`. If set to `Estimated from input signal`, the compensator calculates the coefficients from the input signal. If set to `Input port`, all other properties are disabled and you must provide the coefficients through the input port. The default value is `Estimated from input signal`.

Initial compensator coefficient

Specify the initial coefficient used by the internal algorithm to compensate for the I/Q imbalance. The default value is `0+0j`.

Source of adaptation step size

Specify the source of the adaptation step size as `Property` or `Input port`. If set to `Property`, specify the step size in the **Adaptation step size** field. If set to `Input port`, you must specify the step size through an input port. The default value is `Property`.

Adaptation step size

Specify the step size of the adaptation algorithm as a real scalar. This parameter is available only when **Source of adaptation step size** is set to `Property`. The default value is `0.00001`.

Coefficient adaptation input port

Select this check box to create an input port that permits a signal to control the adaptation process. If the check box is selected and if the input signal is `true`, the estimated compensation coefficients are updated. If the adaptation port is not enabled or if the input signal is `false`, the compensation coefficients do not change. By default, the check box is not selected.

Estimated coefficient output port

Select this check box to provide the estimated compensation coefficients to an output port. By default, the check box is not selected.

Algorithms

This block implements the algorithm, inputs, and outputs described on the `comm.IQImbalanceCompensator` reference page. The object properties correspond to the block parameters.

Examples

Compensate for I/Q Imbalance

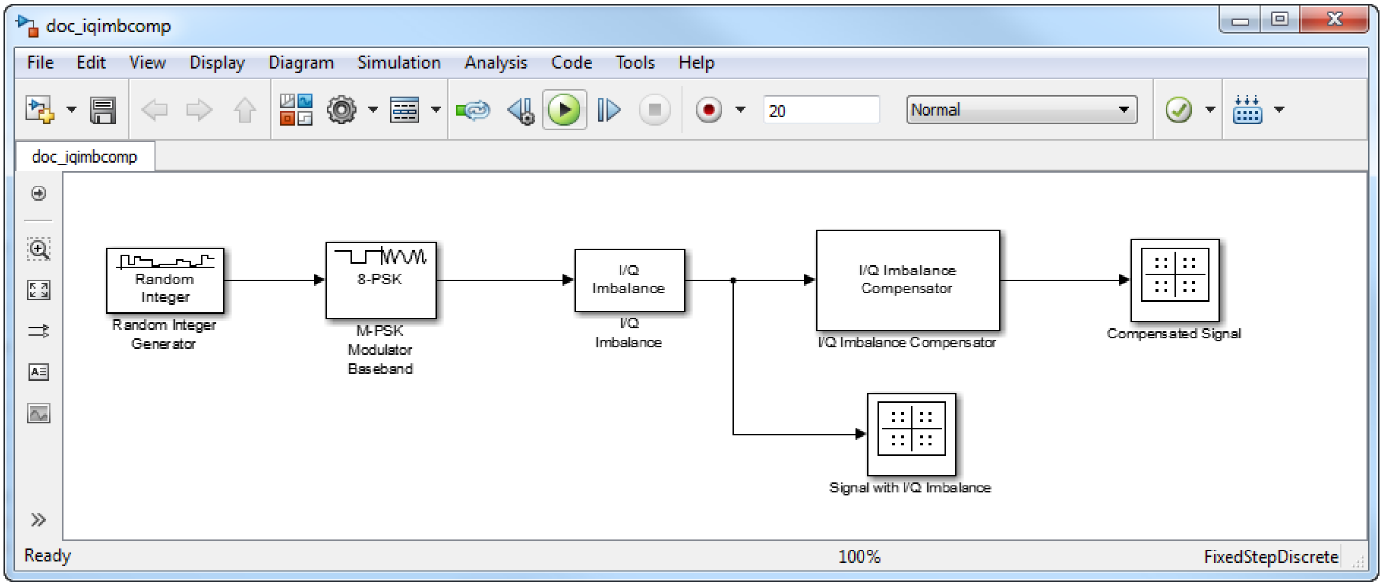
This example shows how to use the I/Q Imbalance Compensator block to remove the effects of an amplitude and phase imbalance on a modulated signal.

Open the model, doc_iqimbcomp, from the MATLAB command prompt.

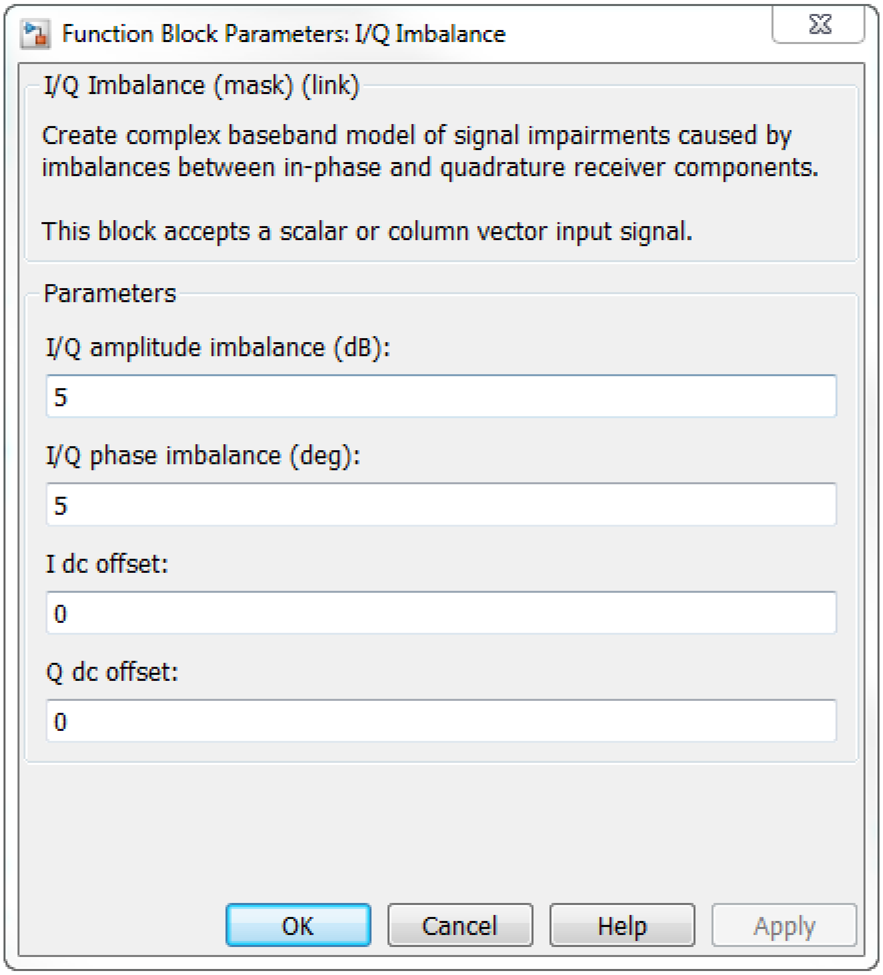
```
doc_iqimbcomp
```

The model includes these blocks:

- Random Integer Generator
- M-PSK Modulator Baseband
- I/Q Imbalance
- I/Q Imbalance Compensator
- Constellation Diagram



Double-click the I/Q Imbalance block. You can see that the **I/Q amplitude imbalance (dB)** parameter is set to 5 and the **I/Q phase imbalance (deg)** parameter is also set to 5.



Function Block Parameters: I/Q Imbalance

I/Q Imbalance (mask) (link)

Create complex baseband model of signal impairments caused by imbalances between in-phase and quadrature receiver components.

This block accepts a scalar or column vector input signal.

Parameters

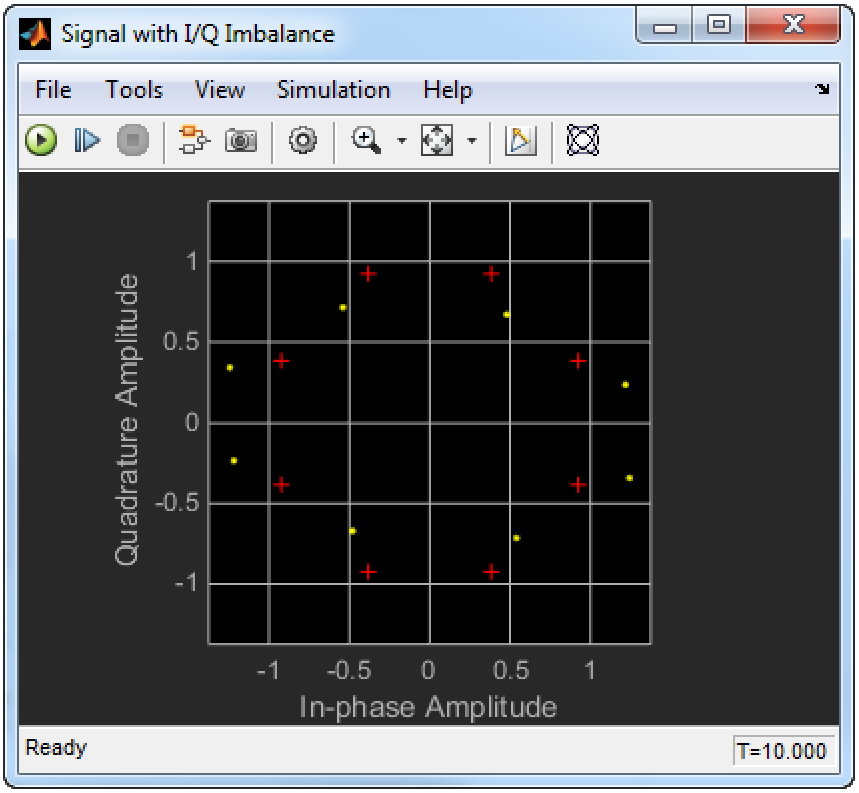
I/Q amplitude imbalance (dB): 5

I/Q phase imbalance (deg): 5

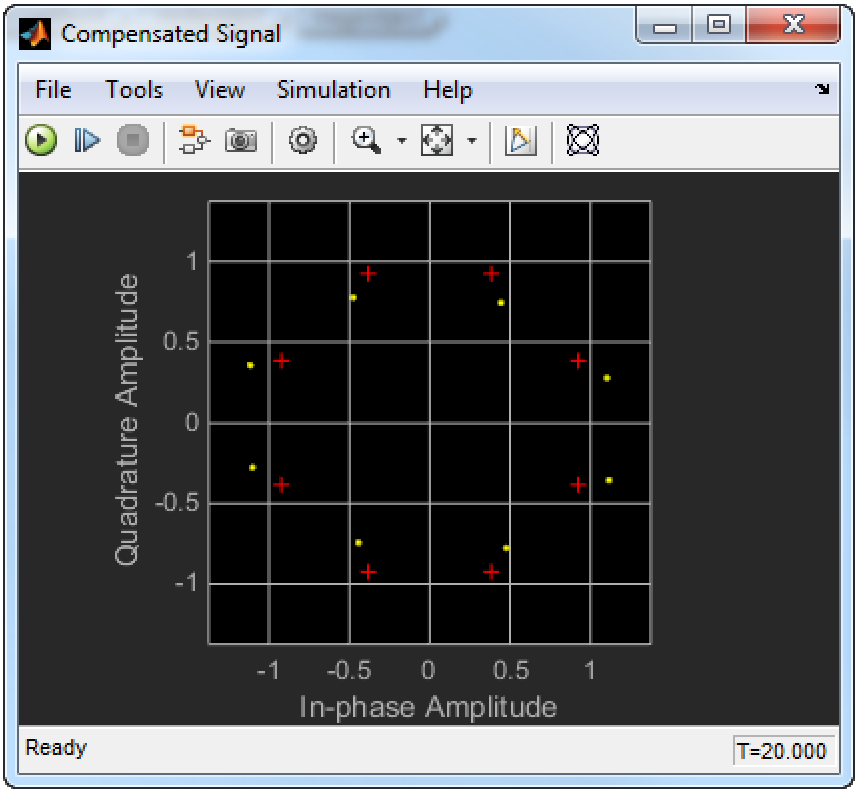
I dc offset: 0

Q dc offset: 0

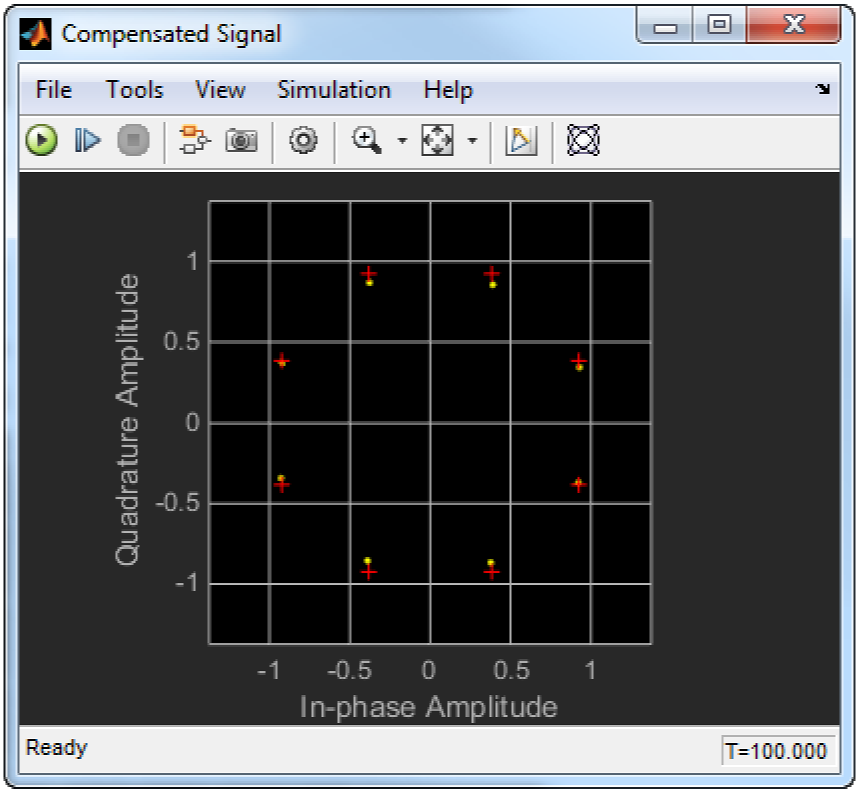
Run the model. In the *Signal with I/Q Imbalance* constellation diagram, observe the effects of the amplitude imbalance and phase imbalance on the 8-PSK signal.



Look at the *Compensated Signal* constellation diagram. Observe that the signal is not well aligned with the reference constellation (shown in red).



Increase the simulation time from 20 seconds to 100 seconds and run the model again. You can see that the constellation is now well aligned with the reference constellation. This is because the compensation algorithm is adaptive; consequently, it requires time to accurately estimate the I/Q imbalance.



Try changing other simulation parameters such as the step size in the I/Q Imbalance Compensator block, the amplitude and phase imbalance in the I/Q Imbalance block, the modulation type etc. Observe the effects on the *Compensated Signal* constellation diagram.

Supported Data Types

Port	Supported Data Types
Signal Input	• Double-precision floating point • Single-precision floating point
Signal Output	• Double-precision floating point • Single-precision floating point
Step Size	• Double-precision floating point • Single-precision floating point
Adaptation	• Logical
Input Coefficients	• Double-precision floating point • Single-precision floating point
Output Coefficients	• Double-precision floating point • Single-precision floating point

See Also

I/Q Imbalance

`comm.IQImbalanceCompensator`

`iqcoef2imbal`

`iqimbal2coef`

Selected Bibliography

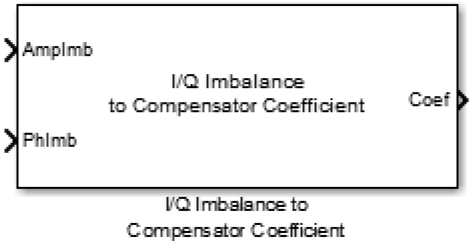
[1] Anttila, L., M. Valkama and M. Renfors. “Blind Compensation of Frequency-Selective I/Q Imbalances in Quadrature Radio Receivers: Circularity-Based Approach”. Proc. IEEE ICASSP. 2007, pp. III-245 -III-248.

[2] Kiayani, A., L. Anttila, Y. Zou, and M. Valkama, “Advanced Receiver Design for Mitigating Multiple RF Impairments in OFDM Systems: Algorithms and RF Measurements”. Journal of Electrical and Computer Engineering. Vol. 2012.

Introduced in R2014b

I/Q Imbalance to Compensator Coefficient

Converts amplitude and phase imbalance into I/Q compensator coefficient



Library

RF Impairments Correction

Description

The I/Q Imbalance to Compensator Coefficient block returns a complex coefficient to compensate for amplitude and phase imbalance.

This block has an amplitude imbalance input port and a phase imbalance input port, where the amplitude imbalance is a real number expressed in dB and the phase imbalance is a real number expressed in degrees. The imbalance inputs are vectors. The complex coefficients are returned from a single output port.

Algorithms

See `iqimbal2coef` for more information on the inputs, outputs, and algorithms.

Supported Data Types

Port	Supported Data Types
Compensator Coefficient	• Double-precision, complex floating point • Single-precision, complex floating point
Amplitude Imbalance (dB)	• Double-precision floating point • Single-precision floating point
Phase Imbalance (deg)	• Double-precision floating point • Single-precision floating point

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

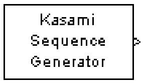
Blocks
I/Q Imbalance Compensator

Functions
`iqimbal2coef`

Introduced in R2014b

Kasami Sequence Generator

Generate Kasami sequence from set of Kasami sequences



Library

Sequence Generators sublibrary of Comm Sources

Description

The Kasami Sequence Generator block generates a sequence from the set of Kasami sequences. The Kasami sequences are a set of sequences that have good cross-correlation properties.

This block can output sequences that vary in length during simulation. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Kasami Sequences

There are two sets of Kasami sequences: the *small set* and the *large set*. The large set contains all the sequences in the small set. Only the small set is optimal in the sense of matching Welch's lower bound for correlation functions.

Kasami sequences have period $N = 2^n - 1$, where n is a nonnegative, even integer. Let u be a binary sequence of length N, and let w be the sequence obtained by decimating u by $2^{n/2} +1$. The small set of Kasami sequences is defined by the following formulas, in which T denotes the left shift operator, m is the shift parameter for w, and ⊕ denotes addition modulo 2.

$$K_S(u, n, m) = \begin{cases} u & m = -1 \\ u \oplus T^m w & m = 0, \ldots, 2^{n/2} - 2 \end{cases}$$

Small Set of Kasami Sequences for n Even

Note that the small set contains $2^{n/2}$ sequences.

For mod(n, 4) = 2, the large set of Kasami sequences is defined as follows. Let v be the sequence formed by decimating the sequence u by $2^{n/2 + 1}+ 1$. The large set is defined by the following table, in which k and m are the shift parameters for the sequences v and w, respectively.

$$K_L(u,n,k,m) = \begin{cases} u & k = -2;\ m = -1 \\ v & k = -1;\ m = -1 \\ u \oplus T^k v & k = 0, \dots, 2^n - 2;\ m = -1 \\ u \oplus T^m w & k = -2;\ m = 0, \dots, 2^{n/2} - 2 \\ v \oplus T^m w & k = -1;\ m = 0, \dots, 2^{n/2} - 2 \\ u \oplus T^k v \oplus T^m w & k = 0, \dots, 2^n - 2;\ m = 0, \dots, 2^{n/2} - 2 \end{cases}$$

Large Set of Kasami Sequences for mod(n, 4) = 2

The sequences described in the first three rows of the preceding figure correspond to the Gold sequences for mod(n, 4) = 2. See the reference page for the Gold Sequence Generator block for a description of Gold sequences. However, the Kasami sequences form a larger set than the Gold sequences.

The correlation functions for the sequences takes on the values

{-$t(n)$, -$s(n)$, -1, $s(n)$ -2 , $t(n)$ - 2}

where

$$t(n) = 1 + 2^{(n+2)/2},\ n \text{ even}$$

$$s(n) = \frac{1}{2}(t(n) + 1)$$

Block Parameters

The **Generator polynomial** parameter specifies the generator polynomial, which determines the connections in the shift register that generates the sequence *u*. You can specify the **Generator polynomial** parameter using these formats:

- A polynomial character vector that includes the number `1`, for example, `'z^4 + z + 1'`.
- A vector that lists the coefficients of the polynomial in descending order of powers. The first and last entries must be 1. Note that the length of this vector is one more than the degree of the generator polynomial.
- A vector containing the exponents of *z* for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For example, `'z^8 + z^2 + 1'`, `[1 0 0 0 0 0 1 0 1]`, and `[8 2 0]` represent the same polynomial, p(z) = z^8+z^2+1.

The **Initial states** parameter specifies the initial states of the shift register that generates the sequence u. **Initial States** is a binary scalar or row vector of length equal to the degree of the **Generator polynomial**. If you choose a binary scalar, the block expands the scalar to a row vector of length equal to the degree of the **Generator polynomial**, all of whose entries equal the scalar.

The **Sequence index** parameter specifies the shifts of the sequences *v* and *w* used to generate the output sequence. You can specify the parameter in either of two ways:

- To generate sequences from the small set, for *n* is even, you can specify the **Sequence index** as an integer *m*. The range of *m* is [-1, ..., $2^{n/2}$ - 2]. The following table describes the output sequences corresponding to **Sequence index** `m`:

Sequence Index	Range of Indices	Output Sequence
`-1`	$m = -1$	u
`m`	$m = 0, \dots, 2^{n/2} - 2$	$u \oplus T^m w$

- To generate sequences from the large set, for mod (n, 4) = 2, where n is the degree of the **Generator polynomial**, you can specify **Sequence index** as an integer vector [k m]. In this case, the output sequence is from the large set. The range for k is [-2, ..., 2^n - 2], and the range for m is [-1, ..., $2^{n/2}$ - 2]. The following table describes the output sequences corresponding to **Sequence index** [k m]:

Sequence Index [k m]	Range of Indices	Output Sequence
`[-2 -1]`	$k = -2$, $m = -1$	u
`[-1 -1]`	$k = -1$, $m = -1$	v
`[k -1]`	$k = 0, 1, \dots, 2^n - 2$ $m = -1$	$u \oplus T^k v$
`[-2 m]`	$k = -2$ $m = 0, 1, \dots, 2^{n/2} - 2$	$u \oplus T^m w$
`[-1 m]`	$k = -1$ $m = 0, \dots, 2^{n/2} - 2$	$v \oplus T^m w$
`[k m]`	$k = 0, \dots, 2^n - 2$ $m = 0, \dots, 2^{n/2} - 2$	$u \oplus T^k v \oplus T^m w$

You can shift the starting point of the Kasami sequence with the **Shift** parameter, which is an integer representing the length of the shift.

You can use an external signal to reset the values of the internal shift register to the initial state by selecting **Reset on nonzero input**. This creates an input port for the external signal in the Kasami Sequence Generator block. The way the block resets the internal shift register depends on whether its output signal and the reset signal are sample-based or frame-based. See “Reset Behavior” on page 5-663 for an example.

Polynomials for Generating Kasami Sequences

The following table lists some of the polynomials that you can use to generate the Kasami set of sequences.

n	N	Polynomial	Set
4	15	[4 1 0]	Small
6	63	[6 1 0]	Large
8	255	[8 4 3 2 0]	Small
10	1023	[10 3 0]	Large
12	4095	[12 6 4 1 0]	Small

Parameters

Generator polynomial

Character vector or binary vector specifying the generator polynomial for the sequence *u*.

Initial states

Binary scalar or row vector of length equal to the degree of the **Generator polynomial**, which specifies the initial states of the shift register that generates the sequence *u*.

Sequence index

Integer or vector specifying the shifts of the sequences *v* and *w* used to generate the output sequence.

Shift

Integer scalar that determines the offset of the Kasami sequence from the initial time.

Output variable-size signals

Select this if you want the output sequences to vary in length during simulation. The default selection outputs fixed-length signals.

Maximum output size source

Specify how the block defines maximum output size for a signal.

- When you select `Dialog parameter`, the value you enter in the **Maximum output size** parameter specifies the maximum size of the output. When you make this selection, the `oSiz` input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value must be less than or equal to the **Maximum output size** parameter.
- When you select `Inherit from reference port`, the block output inherits sample time, maximum size, and current size from the variable-sized signal at the Ref input port.

This parameter only appears when you select **Output variable-size signals**. The default selection is `Dialog parameter`.

Maximum output size

Specify a two-element row vector denoting the maximum output size for the block. The second element of the vector must be 1. For example, [10 1] gives a 10-by-1 maximum sized output signal. This parameter only appears when you select **Output variable-size signals**.

Sample time

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-440.

Samples per frame

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-440.

Reset on nonzero input

When selected, you can specify an input signal that resets the internal shift registers to the original values of the **Initial states**.

Output data type

The output type of the block can be specified as a `boolean` or `double`. By default, the block sets this to `double`.

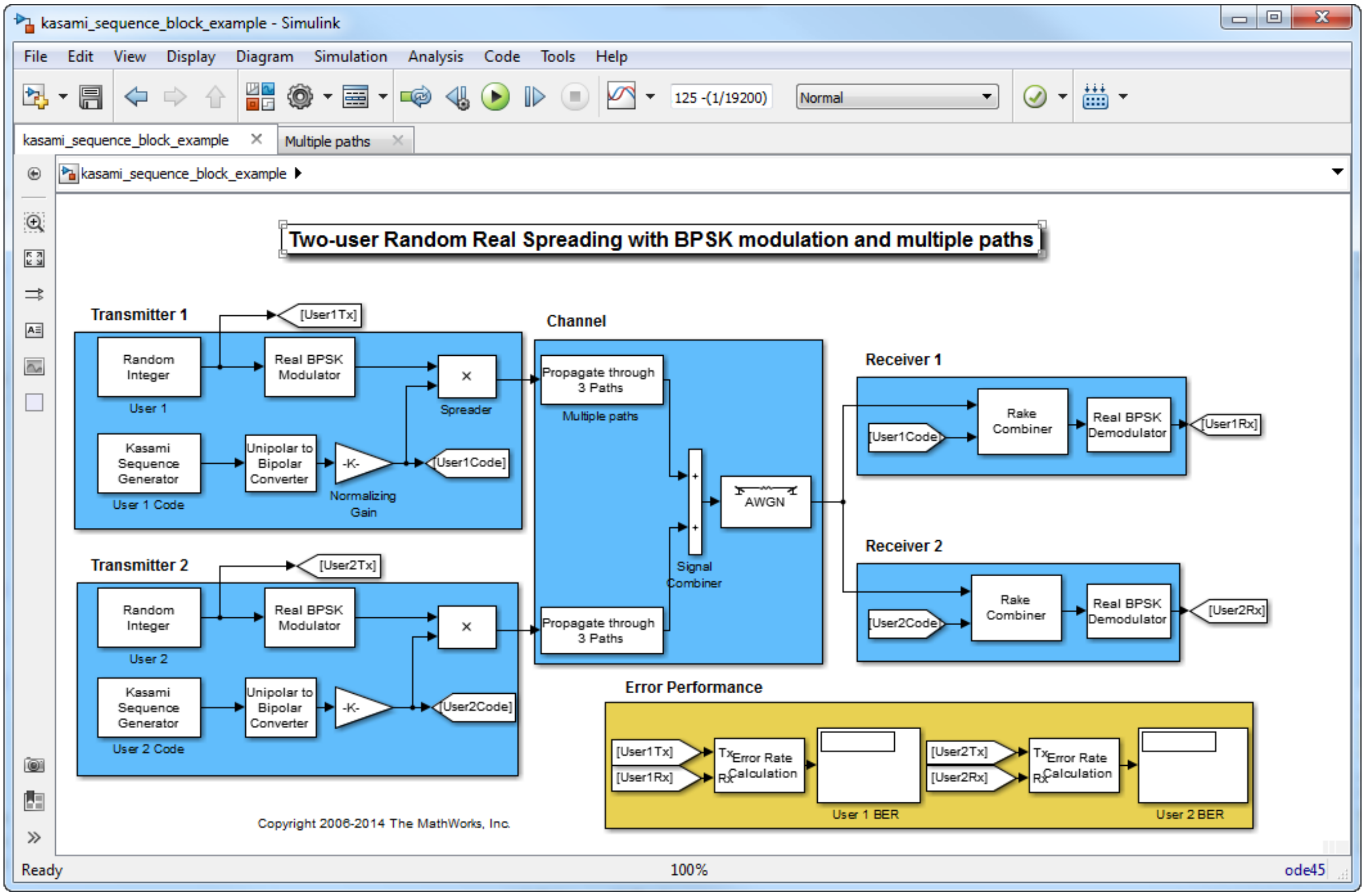
Example

Kasami Spreading with Two Users and Multipath

This model considers Kasami spreading for a combined two-user transmission in a multipath environment.

Open the model here: kasami_sequence_block_example

```
modelname = 'kasami_sequence_block_example';
open_system(modelname);
sim(modelname);
```



You can see very good user separation over multiple paths with the gains of combining. This can be attributed to the "good" correlation properties of Kasami sequences, which provide a balance between the ideal cross-correlation properties of orthogonal codes and the ideal auto-correlation properties of PN sequences. See the relevant examples on the Hadamard Code Generator and PN Sequence Generator reference pages.

To experiment with this model further, try selecting other path delays to see how the performance varies for the same code. Also try different codes with the same delays.

```
close_system(modelname, 0);
```

Reference

[1] Peterson and Weldon, *Error Correcting Codes*, 2nd Ed., MIT Press, Cambridge, MA, 1972.

[2] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

[3] Sarwate, D. V. and Pursley, M.B., "Crosscorrelation Properties of Pseudorandom and Related Sequences," *Proc. IEEE*, Vol. 68, No. 5, May 1980, pp. 583-619.

Blocks
Gold Sequence Generator | Hadamard Code Generator | PN Sequence Generator

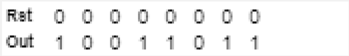
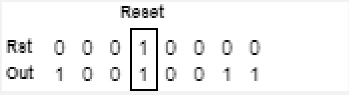
More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Reset Behavior

To reset the generator sequence, you must first select **Reset on nonzero input** to add the `Rst` input. Suppose that the Kasami Sequence Generator block outputs `[1 0 0 1 1 0 1 1]` when there is no reset. The following table shows the effect on the Kasami Sequence Generator block output for the property values indicated.

	Reset Signal Properties	Kasami Sequence Generator block	Reset Signal Output Signal
No reset	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 0 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1 **Out** = `[1 0 0 1 1 0 1 1]`	 Rst 0 0 0 0 0 0 0 0 Out 1 0 0 1 1 0 1 1
Scalar reset signal	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 1 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1	 Reset Rst 0 0 0 1 0 0 0 0 Out 1 0 0 1 0 0 1 1

	Reset Signal Properties	Kasami Sequence Generator block	Reset Signal Output Signal
Vector reset signal	**Sample time** = 1 **Samples per frame** = 8 **Rst** = [0 0 0 1 0 0 0 0]	**Sample time** = 1 **Samples per frame** = 8	

For the no reset case, the sequence is output without being reset. For the scalar and vector reset signal cases, the reset signal [0 0 0 1 0 0 0 0] is input to the Rst port. The sequence output is reset at the fourth bit, because the fourth bit of the reset signal is a 1 and the **Sample time** is 1.

For variable-sized outputs, the block only supports scalar reset signal inputs.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Kasami Sequence Generator block version available before R2015b.

Existing models automatically update to load the Kasami Sequence Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

Extended Capabilities

C/C++ Code Generation

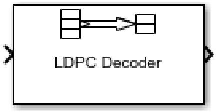
Generate C and C++ code using Simulink® Coder™.

Introduced before R2006a

LDPC Decoder

Decode binary low-density parity-check (LDPC) code

Library: Communications Toolbox / Error Detection and Correction / Block



Description

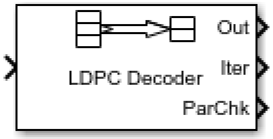
The LDPC Decoder block uses the belief propagation algorithm to decode a binary LDPC code, which is input to the block as the soft-decision output (log-likelihood ratio of received bits) from demodulation. The block decodes generic binary LDPC codes where no patterns in the parity-check matrix are assumed. For more information, see “Belief Propagation Decoding” on page 5-445.

The input and output are discrete-time signals. The ratio of the output sample time to the input sample time is:

- *N/K* when only the information-part of the codeword is decoded
- 1 when the entire codeword is decoded

N is the length of the received signal and must be in the range (0, 2^{31}). *K* is the length of the uncoded message and must be less than *N*.

This icon shows all ports, including optional ports, for the LDPC Decoder block.



Ports

Input

In — Log-likelihood ratios

column vector

Log-likelihood ratios, specified as an *N*-by-1 column vector containing the soft-decision output from demodulation. *N* is the number of bits in the LDPC codeword before modulation. Each element is the log-likelihood ratio for a received bit and the value is more likely to be `0` if the log-likelihood ratio is positive. The first *K* elements correspond to the information-part of the input message.

Data Types: `double`

Output

Out — Decoded data

column vector

Decoded data, returned as a column vector. The **Decision type** parameter specifies whether the block outputs hard decisions or soft decisions (log-likelihood ratios).

- If the **Output format** parameter is set to `Information part`, the output includes only the information-part of the received codeword.
- If the **Output format** parameter is set to `Whole codeword`, the output includes the whole log-likelihood ratio vector.

Data Types: `double` | `Boolean`

`Iter` — Number of executed decoding iterations

positive integer

Number of executed decoding iterations, returned as a positive integer.

Dependencies

To enable this port, select the **Output number of iterations executed** parameter.

Data Types: `double`

`ParChk` — Final parity checks

column vector

Final parity checks after decoding the input LDPC code, returned as an (*N*-*K*)-by-1 column vector. *N* is the number of bits in the LDPC codeword before modulation. *K* is the length of the uncoded message.

Dependencies

To enable this port, select the **Output final parity checks** parameter.

Parameters

`Parity-check matrix (sparse binary (N-K)-by-N matrix)` — Parity-check matrix

`dvbs2ldpc(1/2)` (default) | sparse binary matrix | nonsparse index matrix

Parity-check matrix, specified as a sparse $(N - K)$-by-N binary-valued matrix. N is the length of the received signal and must be in the range $(0, 2^{31})$. K is the length of the uncoded message and must be less than N. The last $(N - K)$ columns in the parity-check matrix must be an invertible matrix in the Galois field of order 2, `gf`(2).

You can also specify the parity-check matrix as a two-column nonsparse index matrix, `I`, that defines the row and column indices of the `1`s in the parity-check matrix such that `sparse(I(:,1),I(:,2),1)`.

This parameter accepts numeric data types. When you set this parameter to a sparse binary matrix, this parameter also accepts the `Boolean` data type.

The default value uses the `dvbs2ldpc` function to configure a sparse parity-check matrix for half-rate LDPC coding, as specified in the DVB-S.2 standard.

Example: `dvbs2ldpc(R,'indices')` configures the index matrix for the DVB-S.2 standard, where `R` is the code rate, and `'indices'` specifies the output format of `dvbs2ldpc` as a two-column double-precision matrix that defines the row and column indices of the `1`s in the parity-check matrix.

Data Types: `double` | `Boolean`

`Output format` — Output value format

`Information part` (default) | `Whole codeword`

Output value format, specified as one of these values:

- `Information part` — The block outputs a *K*-by-1 column vector containing only the information-part of the received log-likelihood ratio vector. *K* is the length of the uncoded message.
- `Whole codeword` — The block outputs an *N*-by-1 column vector containing the whole log-likelihood ratio vector. *N* is the length of the received signal.

  N and *K* must align with the dimension of the (*N*–*K*)-by-*K* parity-check matrix.

`Decision type` — Decision method

`Hard decision` (default) | `Soft decision`

Decision method used for decoding, specified as one of these values:

- `Hard decision` — The block outputs decoded data of data type `double` or `boolean`. Specify this data type using the **Output data type** parameter.
- `Soft decision` — The block outputs log-likelihood ratios of data type `double`.

`Output data type` — Output value data type

`double` (default) | `boolean`

Output value data type, specified as `double` or `boolean`.

Dependencies

To enable this parameter, set the **Decision type** parameter to `Hard decision`.

`Number of iterations` — Maximum number of decoding iterations

`50` (default) | positive integer

Maximum number of decoding iterations, specified as a positive integer.

`Stop iterating when all parity-checks are satisfied` — Condition for iteration termination

`off` (default) | `on`

Select this parameter to terminate decoding after all parity checks are satisfied. If not all parity checks are satisfied, decoding terminates after the number of iterations specified by the **Number of iterations** parameter.

`Output number of iterations executed` — Output number of iterations executed

`off` (default) | `on`

Select this parameter to enable the **Iter** output port.

`Output final parity-checks` — Output number of iterations executed

`off` (default) | `on`

Select this parameter to enable the **ParChk** output port.

Block Characteristics

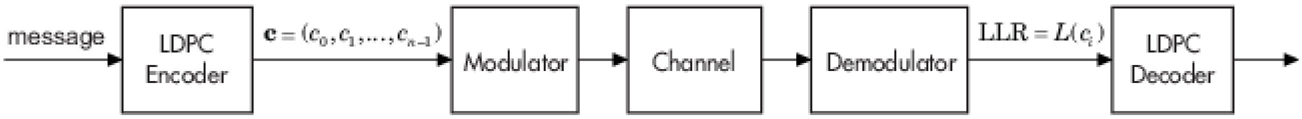
Data Types	`Boolean` | `double`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Algorithms

This block performs LDPC decoding using the belief propagation algorithm, also known as a message-passing algorithm.

Belief Propagation Decoding

The implementation of the belief propagation algorithm is based on the decoding algorithm presented by Gallager.



For transmitted LDPC-encoded codeword $c = c_0, c_1, \ldots, c_{n-1}$, the input to the LDPC decoder is the log-likelihood ratio (LLR) value $L(c_i) = \log\left(\frac{\Pr(c_i = 0 \mid \text{channel output for } c_i)}{\Pr(c_i = 1 \mid \text{channel output for } c_i)}\right)$.

In each iteration, the key components of the algorithm are updated based on these equations:

$$L(r_{ji}) = 2\,\text{atanh}\left(\prod_{i' \in V_j \backslash i} \tanh\left(\frac{1}{2}L(q_{i'j})\right)\right),$$

$L(q_{ij}) = L(c_i) + \sum_{j' \in C_i \backslash j} L(r_{j'i})$, initialized as $L(q_{ij}) = L(c_i)$ before the first iteration, and

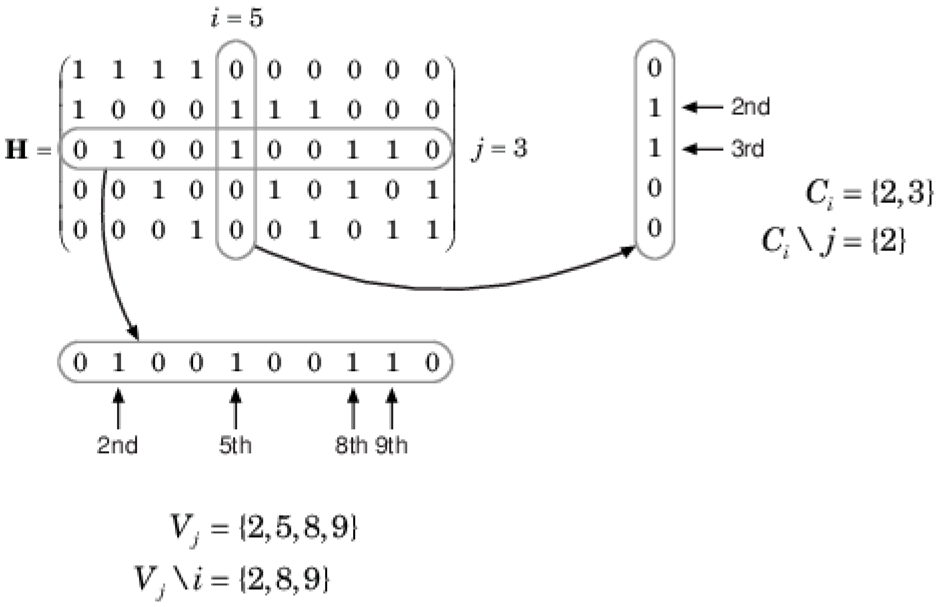
$$L(Q_i) = L(c_i) + \sum_{j' \in C_i} L(r_{j'i}).$$

At the end of each iteration, $L(Q_i)$ contains the updated estimate of the LLR value for transmitted bit c_i. The value $L(Q_i)$ is the soft-decision output for c_i. If $L(Q_i) < 0$, the hard-decision output for c_i is 1. Otherwise, the hard-decision output for c_i is 0.

If configured to stop when all parity checks are satisfied, the algorithm verifies the parity-check equation ($H\,c' = 0$) at the end of each iteration. When all parity checks are satisfied, or if the maximum number of iterations is reached, decoding stops.

Index sets $C_i \backslash j$ and $V_j \backslash i$ are based on the parity-check matrix (PCM). Index sets C_i and V_j correspond to all nonzero elements in column i and row j of the PCM, respectively.

This figure highlights the computation of these index sets in a given PCM for $i = 5$ and $j = 3$.



To avoid infinite numbers in the algorithm equations, atanh(1) and atanh(-1) are set to 19.07 and –19.07, respectively. Due to finite precision, MATLAB returns 1 for tanh(19.07) and –1 for tanh(-19.07).

References

[1] Gallager, Robert G. *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

LDPC Encoder

Objects

`comm.LDPCDecoder`

Functions

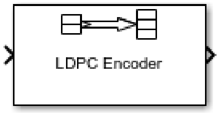
`dvbs2ldpc`

Introduced in R2007a

LDPC Encoder

Encode binary low-density parity-check (LDPC) code

Library: Communications Toolbox / Error Detection and Correction / Block



Description

The LDPC Encoder block applies LDPC coding to a binary input message. LDPC codes are linear error control codes with sparse parity-check matrices and long block lengths that can attain performance near the Shannon limit.

The input and output are discrete-time signals. The ratio of the output sample time to the input sample time is *K/N*, where:

- *N* is the length of the received signal and must be in the range (0, 2^{31}).
- *K* is the length of the uncoded message and must be less than *N*.

Ports

Input

In — Input message

binary column vector

Input message, specified as a *K*-by-1 column vector containing binary-valued elements. *K* is the length of the uncoded message.

Data Types: `double` | `Boolean`

Output

Out — LDPC codeword

column vector

LDPC codeword, returned as an *N*-by-1 column vector. *N* is the number of bits in the LDPC codeword. The output signal inherits its data type from the input signal. The LDPC codeword output is a solution to the parity-check equation. The input message comprises the first *K* bits of the LDPC codeword output, and the parity check comprises the remaining (*N* – *K*) bits.

Data Types: `double` | `Boolean`

Parameters

`Parity-check matrix (sparse binary (N-K)-by-N matrix)` — Parity-check matrix

`dvbs2ldpc(1/2)` (default) | sparse binary matrix | nonsparse index matrix

Parity-check matrix, specified as a sparse (*N* – *K*)-by-*N* binary-valued matrix. *N* is the length of the output LDPC codeword and must be in the range (0, 2^{31}). *K* is the length of the uncoded message and must be less than *N*. The last (*N* – *K*) columns in the parity-check matrix must be an invertible matrix in the Galois field of order 2, `gf`(2).

You can also specify the parity-check matrix as a two-column nonsparse index matrix, `I`, that defines the row and column indices of the `1`s in the parity-check matrix such that `sparse(I(:,1),I(:,2),1)`.

This parameter accepts numeric data types. When you set this parameter to a sparse binary matrix, this parameter also accepts the `Boolean` data type.

The default value uses the `dvbs2ldpc` function to configure a sparse parity-check matrix for half-rate LDPC coding, as specified in the DVB-S.2 standard.

Note

- When the last (*N* – *K*) columns of the parity-check matrix form a triangular matrix, forward or backward substitution is performed to solve the parity-check equation.
- When the last (*N* – *K*) columns of the parity-check matrix do not form a triangular matrix, a matrix inversion is performed to solve the parity-check equation. If a large matrix needs to be inverted, initializations or updates take more time.

Example: `dvbs2ldpc(R,'indices')` configures the index matrix for the DVB-S.2 standard, where `R` is the code rate, and `'indices'` specifies the output format of `dvbs2ldpc` as a two-column double-precision matrix that defines the row and column indices of the `1`s in the parity-check matrix.

Data Types: `double` | `Boolean`

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
LDPC Decoder

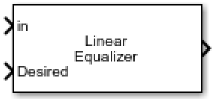
Objects
`comm.LDPCEncoder`

Functions
`dvbs2ldpc`

Introduced in R2007a

Linear Equalizer

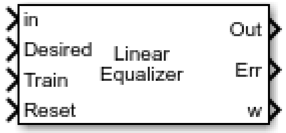
Equalize modulated signals using linear filtering
Library: Communications Toolbox / Equalizers



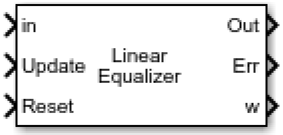
Description

The Linear Equalizer block uses a tapped delay line filter to equalize a linearly modulated signal through a dispersive channel. Using an estimate of the channel modeled as a finite input response (FIR) filter, the block processes input frames and outputs the estimated signal.

This icon shows the block with all ports enabled for configurations that use the LMS or RLS adaptive algorithm.



This icon shows the block with all ports enabled for configurations that use the CMA adaptive algorithm.



Ports

Input

`in` — Input signal

column vector

Input signal, specified as a column vector. The vector length of **in** must be equal to an integer multiple of the **Number of input samples per symbol** parameter. For more information, see “Symbol Tap Spacing” on page 5-456.

Data Types: `double`
Complex Number Support: Yes

`Desired` — Training symbols

column vector

Training symbols, specified as a column vector. The vector length of **Desired** must be less than or equal to the length of input **in**. The **Desired** input port is ignored when the **Train** input port is `0`.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `LMS` or `RLS`.

Data Types: `double`
Complex Number Support: Yes

`Train` — Train equalizer flag
`1` | `0`

Train equalizer flag, specified as `1` or `0`. The block starts training when this value changes from `0` to `1` (at the rising edge). The block trains until all symbols in the **Desired** input port are processed.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `LMS` or `RLS` and select the **Enable training control input** parameter.

Data Types: `Boolean`

`Update` — Update tap weights flag
`1` | `0`

Update tap weights flag, specified as `1` or `0`. The tap weights are updated when this value is `1`.

Dependencies

To enable this port, set the **Adaptive algorithm** parameter to `CMA` and the **Source of adapt weights flag** parameter to `Input port`.

Data Types: `Boolean`

`Reset` — Reset equalizer flag
`1` | `0`

Reset equalizer flag, specified as `1` or `0`. If **Reset** is set to `1`, the block resets the tap weights before processing the incoming signal. The block performs initial training until all symbols in the **Desired** input port are processed.

Dependencies

To enable this port, select the **Enable reset input** parameter.

Data Types: `Boolean`

Output

`Out` — Equalized symbols
column vector

Equalized symbols, returned as a column vector that has the same length as input signal **in**.

This port is unnamed until you select the **Output error signal** or **Output taps weights** parameter.

`Err` — Error signal
column vector

Error signal, returned as a column vector that has the same length as input signal **in**.

`w` — Tap weights

column vector

Tap weights, returned as an N_{Taps}-by-1 vector, where N_{Taps} is the value of the **Number of Taps** parameter. **w** contains the tap weights from the last tap weight update.

Parameters

Structure parameters

`Number of taps` — Number of equalizer taps

`5` (default) | positive integer

Number of equalizer taps, specified as a positive integer. The number of equalizer taps must be greater than or equal to the value of the **Number of input samples per symbol** parameter.

`Signal constellation` — Signal constellation

`pskmod(0:3,4,pi/4)` (default) | vector

Signal constellation, specified as a vector. The default value is a QPSK constellation generated using this code: `pskmod(0:3,4,pi/4)`.

Tunable: Yes

`Number of input samples per symbol` — Number of input samples per symbol

`1` (default) | positive integer

Number of input samples per symbol, specified as a positive integer. Setting this parameter to any number greater than `1` effectively creates a fractionally spaced equalizer. For more information, see “Symbol Tap Spacing” on page 5-456.

Algorithm parameters

`Adaptive algorithm` — Adaptive algorithm

`LMS` (default) | `RLS` | `CMA`

Adaptive algorithm used for equalization, specified as one of these values:

- `LMS` — Update the equalizer tap weights using the “Least Mean Square (LMS) Algorithm” on page 5-457.
- `RLS` — Update the equalizer tap weights using the “Recursive Least Square (RLS) Algorithm” on page 5-457.
- `CMA` — Update the equalizer tap weights using the “Constant Modulus Algorithm (CMA)” on page 5-458.

`Step size` — Step size

`0.01` (default) | positive scalar

Step size used by the adaptive algorithm, specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `CMA`.

`Forgetting factor` — Forgetting factor

`0.99` (default) | scalar in the range (0, 1]

Forgetting factor used by the adaptive algorithm, specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalizer output estimates to be less stable.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `RLS`.

`Initial inverse correlation matrix` — Initial inverse correlation matrix

`0.1` (default) | scalar | matrix

Initial inverse correlation matrix, specified as a scalar or an N_{Taps}-by-N_{Taps} matrix. N_{Taps} is equal to the **Number of Taps** parameter value. If you specify this value as a scalar, *a*, the equalizer sets the initial inverse correlation matrix to *a* times the identity matrix: a(`eye`(N_{Taps})).

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `RLS`.

Control parameters

`Reference tap` — Reference tap

`3` (default) | positive integer

Reference tap, specified as a positive integer less than or equal to the **Number of Taps** parameter value. The equalizer uses the reference tap location to track the main energy of the channel.

`Input signal delay (samples)` — Input signal delay

`0` (default) | nonnegative integer

Input signal delay in samples relative to the reset time of the equalizer, specified as a nonnegative integer. If the input signal is a vector of length greater than 1, then the input delay is relative to the start of the input vector. If the input signal is a scalar, then the input delay is relative to the first call of the block and to the first call of the block after the **Reset** input port toggles to `1`.

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Source of adapt weights flag` — Source of adapt tap weights request

`Property` (default) | `Input port`

Source of the adapt tap weights request, specified as one of these values:

- `Property` — Specify this value to use the **Adaptive algorithm** parameter to control when the block adapts tap weights.

- `Input port` — Specify this value to use the **Update** input port to control when the block adapts tap weights.

Dependencies

To enable this parameter, set **Adaptive algorithm** to `CMA`.

`Adapt tap weights` — Adapt tap weights

`on` (default) | `off`

Select this parameter to adaptively update the equalizer tap weights. If this parameter is cleared, the block keeps the equalizer tap weights unchanged.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `CMA` and **Source of adapt weights flag** to `Property`.

`Initial tap weights source` — Source for initial tap weights

`Auto` (default) | `Property`

Source for initial tap weights, specified as one of these values:

- `Auto` — Initialize the tap weights to the algorithm-specific default values, as described in the **Initial weights** parameter.
- `Property` — Initialize the tap weights using the **Initial weights** parameter value.

`Initial weights` — Initial tap weights

`0` or `[0;0;1;0;0]` (default) | scalar | column vector

Initial tap weights used by the adaptive algorithm, specified as a scalar or an N_{Taps}-by-1 vector. N_{Taps} is equal to the **Number of Taps** parameter value. The default is `0` when the **Adaptive algorithm** parameter is set to `LMS` or `RLS`. The default is `[0;0;1;0;0]` when the **Adaptive algorithm** parameter is set to `CMA`.

If you specify **Initial weights** as a vector, the vector length must be equal to the **Number of Taps** parameter value. If you specify **Initial weights** as a scalar, the equalizer uses scalar expansion to create a vector of length **Number of Taps** with all values set to **Initial weights**.

Tunable: Yes

Dependencies

To enable this parameter, set **Initial tap weights source** to `Property`.

`Tap weight update period (symbols)` — Tap weight update period

`1` (default) | positive integer

Tap weight update period in symbols, specified as a positive integer. The equalizer updates the tap weights after processing this number of symbols.

`Enable training control input` — Enable training control input

`off` (default) | `on`

Select this parameter to enable input port **Train**. If this parameter is cleared, the block does not reenter training mode after the initial tap training.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Update tap weights when not training` — Update tap weights when not training

`on` (default) | `off`

Select this parameter to use decision directed mode to update equalizer tap weights. If this parameter is cleared, the block keeps the equalizer tap weights unchanged after training.

Tunable: Yes

Dependencies

To enable this parameter, set **Adaptive algorithm** to `LMS` or `RLS`.

`Enable reset input` — Enable reset input

`off` (default) | `on`

Select this parameter to enable input port **Train**. If this parameter is cleared, the block does not reenter training mode after the initial tap training.

Tunable: Yes

Diagnostic parameters

`Output error signal` — Enable error signal output

`off` (default) | `on`

Select this parameter to enable output port **Err** containing the equalizer error signal.

Tunable: Yes

`Output taps weights` — Enable tap weights output

`off` (default) | `on`

Select this parameter to enable output port **w** containing tap weights from the last tap weight update.

Tunable: Yes

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –- Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –- Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

Symbol Tap Spacing

You can configure the equalizer to operate as a symbol-spaced equalizer or as a fractional symbol-spaced equalizer.

- To operate the equalizer at a symbol-spaced rate, specify the number of samples per symbol as `1`. Symbol-rate equalizers have taps spaced at the symbol duration. Symbol-rate equalizers are sensitive to timing phase.
- To operate the equalizer at a fractional symbol-spaced rate, specify the number of input samples per symbol as an integer greater than `1` and provide an input signal oversampled at that sampling rate. Fractional symbol-spaced equalizers have taps spaced at an integer fraction of the input symbol duration. Fractional symbol-spaced equalizers are not sensitive to timing phase.

Algorithms

Linear Equalizers

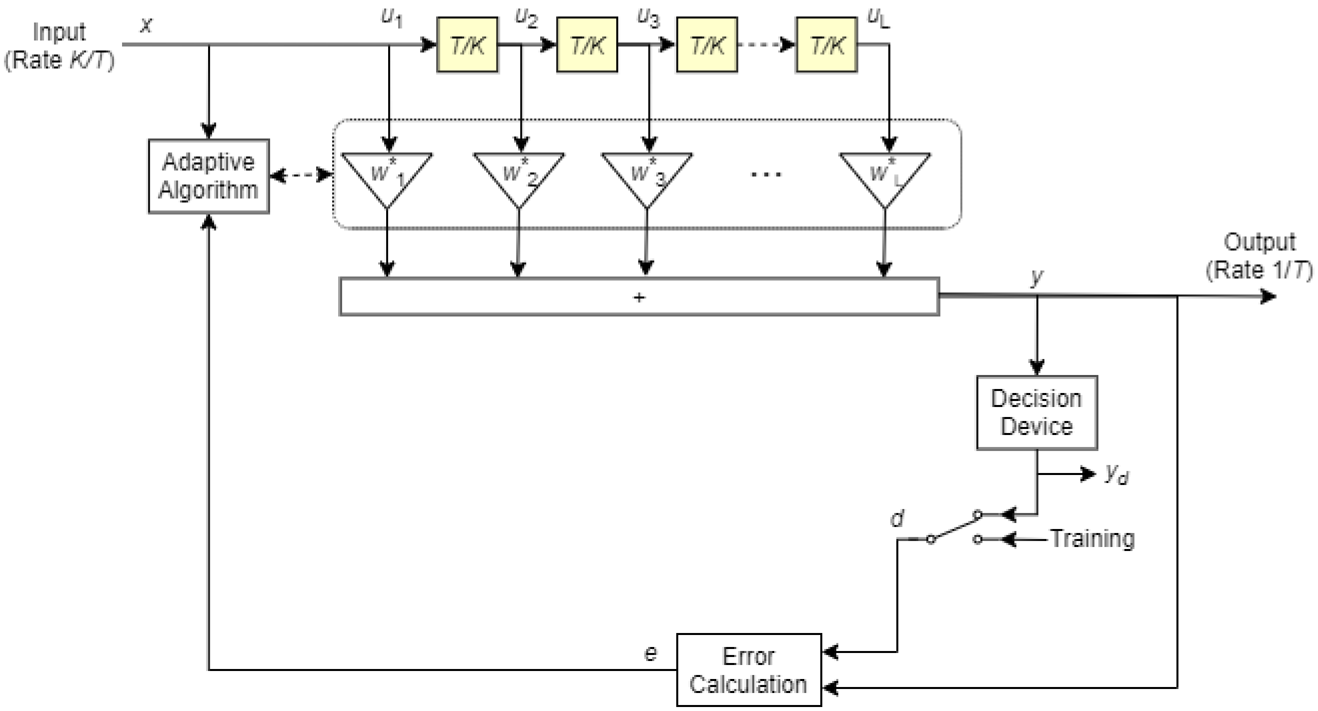
Linear equalizers can remove intersymbol interference (ISI) when the frequency response of a channel has no null. If a null exists in the frequency response of a channel, linear equalizers tend to enhance the noise. In this case, use decision feedback equalizers to avoid enhancing the noise.

A linear equalizer consists of a tapped delay line that stores samples from the input signal. Once per symbol period, the equalizer outputs a weighted sum of the values in the delay line and updates the weights to prepare for the next symbol period.

Linear equalizers can be symbol-spaced or fractional symbol-spaced.

- For a symbol-spaced equalizer, the number of samples per symbol, *K*, is 1. The output sample rate equals the input sample rate.
- For a fractional symbol-spaced equalizer, the number of samples per symbol, *K*, is an integer greater than 1. Typically, *K* is 4 for fractionally spaced equalizers. The output sample rate is 1/*T* and the input sample rate is *K*/*T*, where *T* is the symbol period. Tap-weight updating occurs at the output rate.

This schematic shows a linear equalizer with *L* weights, a symbol period of *T*, and *K* samples per symbol. If *K* is 1, the result is a symbol-spaced linear equalizer instead of a fractional symbol-spaced linear equalizer.



In each symbol period, the equalizer receives *K* input samples at the tapped delay line. The equalizer then outputs a weighted sum of the values in the tapped delay line and updates the weights to prepare for the next symbol period.

For more information, see “Equalization”.

Least Mean Square (LMS) Algorithm

For the LMS algorithm, in the previous schematic, *w* is a vector of all weights w_i, and *u* is a vector of all inputs u_i. Based on the current set of weights, the LMS algorithm creates the new set of weights as

$$w_{\text{new}} = w_{\text{current}} + (StepSize)\ ue^*.$$

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed when using the LMS adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Recursive Least Square (RLS) Algorithm

For the RLS algorithm, in the previous schematic, *w* is the vector of all weights w_i, and *u* is the vector of all inputs u_i. Based on the current set of inputs, *u*, and the inverse correlation matrix, *P*, the RLS algorithm first computes the Kalman gain vector, *K*, as

$$K = \frac{Pu}{(ForgettingFactor) + u^H Pu}.$$

The forgetting factor used by the adaptive algorithm is specified as a scalar in the range (0, 1]. Decreasing the forgetting factor reduces the equalizer convergence time but causes the equalized output signal to be less stable. *H* denotes the Hermitian transpose. Based on the current inverse correlation matrix, the new inverse correlation matrix is

$$P_{\text{new}} = \frac{(1 - Ku^{H})P_{\text{current}}}{ForgettingFactor}.$$

Based on the current set of weights, the RLS algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + K^{*}e$.

The * operator denotes the complex conjugate and the error calculation $e = d - y$.

Constant Modulus Algorithm (CMA)

For the CMA adaptive algorithm, in the previous schematic, w is the vector of all weights w_i, and u is the vector of all inputs u_i. Based on the current set of weights, the CMA adaptive algorithm creates the new set of weights as

$w_{\text{new}} = w_{\text{current}} + (StepSize)\ u^{*}e$.

The step size used by the adaptive algorithm is specified as a positive scalar. Increasing the step size reduces the equalizer convergence time but causes the equalized output signal to be less stable. To determine the maximum step size allowed by the CMA adaptive algorithm, use the `maxstep` object function. The * operator denotes the complex conjugate and the error calculation $e = y(R - |y|^{2})$, where R is a constant related to the signal constellation.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | MLSE Equalizer

Objects

`comm.LinearEqualizer`

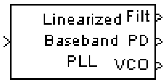
Topics

“Equalization”
“Adaptive Equalizers”

Introduced in R2019a

Linearized Baseband PLL

Implement linearized version of baseband phase-locked loop



Library

Components sublibrary of Synchronization

Description

The Linearized Baseband PLL block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. Unlike the Phase-Locked Loop block, this block uses a baseband model method. Unlike the Baseband PLL block, which uses a nonlinear model, this block simplifies the computations by using *x* to approximate sin(*x*). The baseband PLL model depends on the amplitude of the incoming signal but does not depend on a carrier frequency.

This PLL has these three components:

- An integrator used as a phase detector.
- A filter. You specify the filter's transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```
- A voltage-controlled oscillator (VCO). You specify the sensitivity of the VCO signal to its input using the **VCO input sensitivity** parameter. This parameter, measured in Hertz per volt, is a scale factor that determines how much the VCO shifts from its quiescent frequency.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

Parameters

Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

Lowpass filter denominator

The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

VCO input sensitivity (Hz/V)

This value scales the input to the VCO and, consequently, the shift from the VCO's quiescent frequency.

References

For more information about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

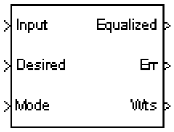
Baseband PLL | Phase-Locked Loop

Introduced before R2006a

LMS Decision Feedback Equalizer

(To be removed) Equalize using decision feedback equalizer that updates weights with LMS algorithm

Note will be removed in a future release. Use Decision Feedback Equalizer instead.



Library

Equalizers

Description

The LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the LMS algorithm to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Parameters

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps

The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of forward taps in the equalizer.

Step size

The step size of the LMS algorithm.

Leakage factor

The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

If you select this check box, the block has an input port that enables you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Compatibility Considerations

LMS Decision Feedback Equalizer will be removed

Warns starting in R2020a

- LMS Decision Feedback Equalizer will be removed in a future release. Use Decision Feedback Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Decision Feedback Equalizer block is equivalent to the **Mode input port** parameter of the LMS Decision Feedback Equalizer block.
- The Decision Feedback Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the LMS Decision Feedback Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

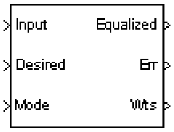
“Equalization”

Introduced before R2006a

LMS Linear Equalizer

(To be removed) Equalize using linear equalizer that updates weights with LMS algorithm

Note will be removed in a future release. Use Linear Equalizer instead.



Library

Equalizers

Description

The LMS Linear Equalizer block uses a linear equalizer and the LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer. When you set the **Number of samples per symbol** parameter to a value greater than one, the block updates the weights once every N^{th} sample for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Parameters

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulated signal, as determined by the modulator in your model

Reference tap

A positive integer less than or equal to the number of taps in the equalizer.

Step size

The step size of the LMS algorithm.

Leakage factor

The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that lists the initial weights for the taps.

Mode input port

If you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you select this check box, the block outputs the current weights.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Compatibility Considerations

LMS Linear Equalizer will be removed

Warns starting in R2020a

- LMS Linear Equalizer will be removed in a future release. Use Linear Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Linear Equalizer block is equivalent to the **Mode input port** parameter of the LMS Linear Equalizer block.
- The Linear Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the LMS Linear Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

"Equalization"

Introduced before R2006a

Matrix Deinterleaver

Permute input symbols by filling matrix by columns and emptying it by rows

Library

Block sublibrary of Interleaving

Description

The Matrix Deinterleaver block performs block deinterleaving by filling a matrix with the input symbols column by column and then sending the matrix contents to the output port row by row. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

This block accepts a column vector input signal. The length of the input vector must be **Number of rows** times **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

Examples

If the **Number of rows** and **Number of columns** parameters are 2 and 3, respectively, then the deinterleaver uses a 2-by-3 matrix for its internal computations. Given an input signal of `[1; 2; 3; 4; 5; 6]`, the block produces an output of `[1; 3; 5; 2; 4; 6]`.

Pair Block

Matrix Interleaver

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
General Block Deinterleaver | Matrix Interleaver

Introduced before R2006a

Matrix Helical Scan Deinterleaver

Restore ordering of input symbols by filling matrix along diagonals

Library

Block sublibrary of Interleaving

Description

The Matrix Helical Scan Deinterleaver block performs block deinterleaving by filling a matrix with the input symbols in a helical fashion and then sending the matrix contents to the output port row by row. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

Helical fashion means that the block places input symbols along diagonals of the matrix. The number of elements in each diagonal matches the **Number of columns** parameter, after the block wraps past the edges of the matrix when necessary. The block traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

The **Array step size** parameter is the slope of each diagonal, that is, the amount by which the row index increases as the column index increases by one. This parameter must be an integer between zero and the **Number of rows** parameter. If the **Array step size** parameter is zero, then the block does not deinterleave and the output is the same as the input.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

Array step size

The slope of the diagonals that the block writes.

Pair Block

Matrix Helical Scan Interleaver

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Block Deinterleaver | Matrix Helical Scan Interleaver

Introduced before R2006a

Matrix Helical Scan Interleaver

Permute input symbols by selecting matrix elements along diagonals

Library

Block sublibrary of Interleaving

Description

The Matrix Helical Scan Interleaver block performs block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port in a helical fashion. The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

Helical fashion means that the block selects output symbols by selecting elements along diagonals of the matrix. The number of elements in each diagonal matches the **Number of columns** parameter, after the block wraps past the edges of the matrix when necessary. The block traverses diagonals so that the row index and column index both increase. Each diagonal after the first one begins one row below the first element of the previous diagonal.

The **Array step size** parameter is the slope of each diagonal, that is, the amount by which the row index increases as the column index increases by one. This parameter must be an integer between zero and the **Number of rows** parameter. If the **Array step size** parameter is zero, then the block does not interleave and the output is the same as the input.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

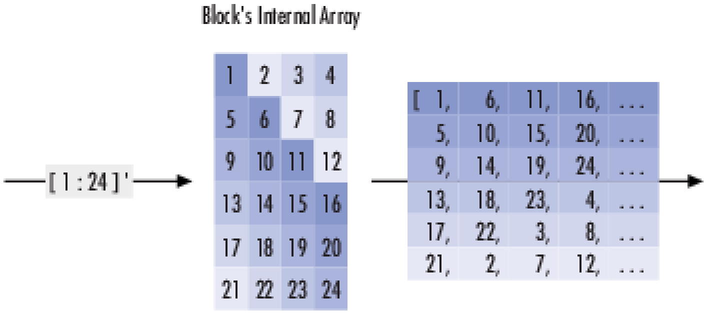
Array step size

The slope of the diagonals that the block reads.

Examples

If the **Number of rows** and **Number of columns** parameters are 6 and 4, respectively, then the interleaver uses a 6-by-4 matrix for its internal computations. If the **Array step size** parameter is 1, then the diagonals are as shown in the figure below. Positions with the same color form part of the same diagonal, and diagonals with darker colors precede those with lighter colors in the output signal.

Given an input signal of `[1:24]'`, the block produces an output of



```
[1; 6; 11; 16; 5; 10; 15; 20; 9; 14; 19; 24; 13; 18; 23;...
4; 17; 22; 3; 8; 21; 2; 7; 12]
```

Pair Block

Matrix Helical Scan Deinterleaver

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Block Interleaver | Matrix Helical Scan Deinterleaver

Introduced before R2006a

Matrix Interleaver

Permute input symbols by filling matrix by rows and emptying it by columns

Library

Block sublibrary of Interleaving

Description

The Matrix Interleaver block performs block interleaving by filling a matrix with the input symbols row by row and then sending the matrix contents to the output port column by column.

The **Number of rows** and **Number of columns** parameters are the dimensions of the matrix that the block uses internally for its computations.

This block accepts a column vector input signal. The number of elements of the input vector must be the product of **Number of rows** and **Number of columns**.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Number of rows

The number of rows in the matrix that the block uses for its computations.

Number of columns

The number of columns in the matrix that the block uses for its computations.

Examples

If the **Number of rows** and **Number of columns** parameters are 2 and 3, respectively, then the interleaver uses a 2-by-3 matrix for its internal computations. Given an input signal of `[1; 2; 3; 4; 5; 6]`, the block produces an output of `[1; 4; 2; 5; 3; 6]`.

Pair Block

Matrix Deinterleaver

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
General Block Interleaver | Matrix Deinterleaver

Introduced before R2006a

M-DPSK Demodulator Baseband

Demodulate DPSK-modulated data

Library

PM, in Digital Baseband sublibrary of Modulation

Description

The M-DPSK Demodulator Baseband block demodulates a signal that was modulated using the M-ary differential phase shift keying method. The input is a baseband representation of the modulated signal. The input and output for this block are discrete-time signals. This block accepts a scalar-valued or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 5-477 table on this page.

The **M-ary number** parameter, M, is the number of possible output symbols that can immediately follow a given output symbol. The block compares the current symbol to the previous symbol. The block's first output is the initial condition of zero (or a group of zeros, if the **Output type** parameter is set to `Bit`) because there is no previous symbol.

Integer-Valued Signals and Binary-Valued Signals

If you set the **Output type** parameter to `Integer`, then the block demodulates a phase difference of

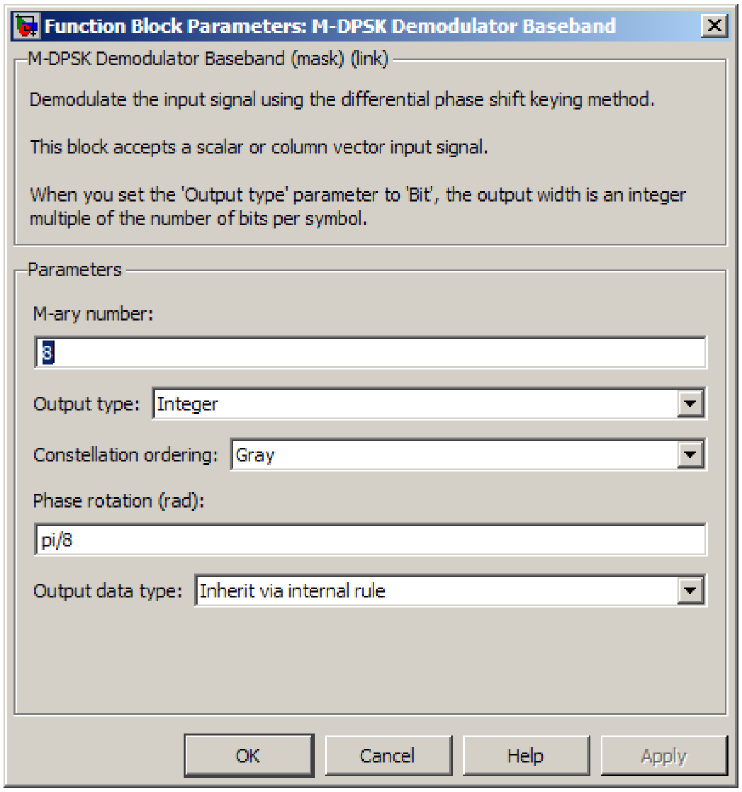
$$\theta + 2\pi k/M$$

to k, where θ represents the **Phase rotation** parameter and k represents an integer between `0` and M-1.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

In binary output mode, the symbols can be either binary-demapped or Gray-demapped. The **Constellation ordering** parameter indicates how the block maps an integer to a corresponding group of K output bits. See the reference pages for the M-DPSK Modulator Baseband and M-PSK Modulator Baseband blocks for details.

Dialog Box



Function Block Parameters: M-DPSK Demodulator Baseband

M-DPSK Demodulator Baseband (mask) (link)

Demodulate the input signal using the differential phase shift keying method.

This block accepts a scalar or column vector input signal.

When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol.

Parameters

M-ary number: 8

Output type: Integer

Constellation ordering: Gray

Phase rotation (rad): pi/8

Output data type: Inherit via internal rule

M-ary number

The number of possible modulated symbols that can immediately follow a given symbol.

Output type

Determines whether the output consists of integers or groups of bits.

Constellation ordering

Determines how the block maps each integer to a group of output bits.

Phase rotation (rad)

This phase difference between the current and previous modulated symbols that results in an output of zero.

Output data type

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`.

For integer outputs, this block can output the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, output can be `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	- Double-precision floating point - Single-precision floating point
Output	- Double-precision floating point - Single-precision floating point - Boolean when **Output type** set to `Bit` - 8-, 16-, and 32-bit signed integers - 8-, 16-, and 32-bit unsigned integers

Pair Block

M-DPSK Modulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

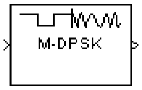
Blocks

DBPSK Demodulator Baseband | DQPSK Demodulator Baseband | M-DPSK Modulator Baseband | M-PSK Demodulator Baseband

Introduced before R2006a

M-DPSK Modulator Baseband

Modulate using M-ary differential phase shift keying method



Library

PM, in Digital Baseband sublibrary of Modulation

Description

The M-DPSK Modulator Baseband block modulates using the M-ary differential phase shift keying method. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the number of possible output symbols that can immediately follow a given output symbol.

The input must be a discrete-time signal. For integer inputs, the block can accept the data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit inputs, the block can accept `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, and `double`.

The input can be either bits or integers, which are binary-mapped or Gray-mapped into symbols.

This block accepts column vector input signals. For a bit input, the input width must be an integer multiple of the number of bits per symbol.

Integer-Valued Signals and Binary-Valued Signals

If you set the **Input type** parameter to `Integer`, then valid input values are integers between 0 and M-1. In this case, the input can be either a scalar or a frame-based column vector. If the first input is k_1, then the modulated symbol is

$$\exp\left(j\theta + j2\pi\frac{k_1}{m}\right)$$

where θ represents the **Phase rotation** parameter. If a successive input is *k*, then the modulated symbol is

$$\exp\left(j\theta + j2\pi\frac{k}{m}\right) \cdot \text{(previous modulated symbol)}$$

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of *K* bits.

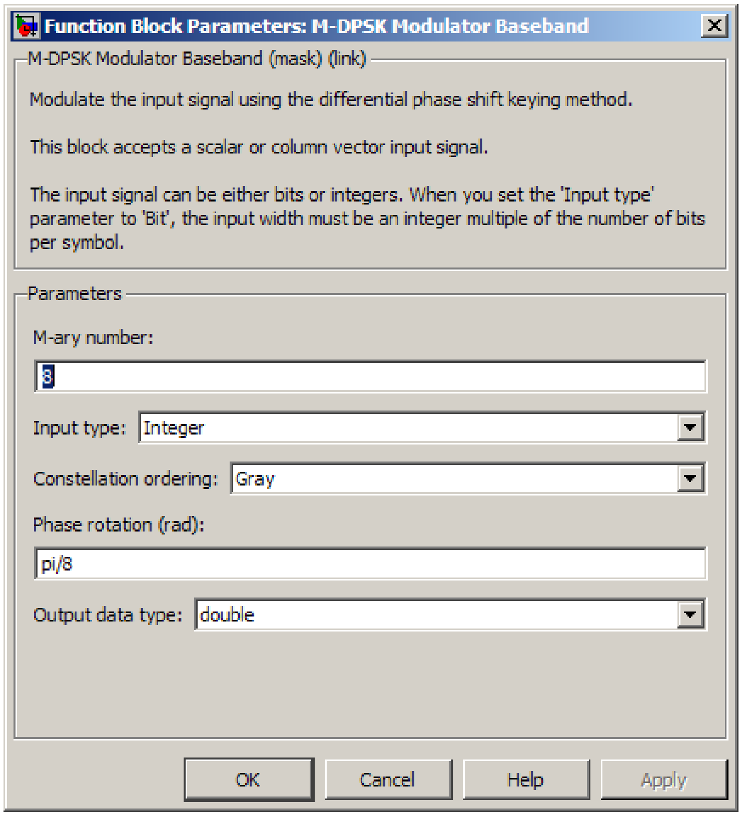
The input can be a column vector with a length that is an integer multiple of *K*.

In binary input mode, the **Constellation ordering** parameter indicates how the block maps a group of *K* input bits to a corresponding phase difference. The `Binary` option uses a natural binary-to-integer mapping, while the `Gray` option uses a Gray-coded assignment of phase differences. For example, the following table indicates the assignment of phase difference to three-bit inputs, for both the `Binary` and `Gray` options. θ is the **Phase rotation** parameter. The phase difference is between the previous symbol and the current symbol.

Current Input	Binary-Coded Phase Difference	Gray-Coded Phase Difference
[0 0 0]	jθ	jθ
[0 0 1]	jθ + jπ/4	jθ + jπ/4
[0 1 0]	jθ + jπ2/4	jθ + jπ3/4
[0 1 1]	jθ + jπ3/4	jθ + jπ2/4
[1 0 0]	jθ + jπ4/4	jθ + jπ7/4
[1 0 1]	jθ + jπ5/4	jθ + jπ6/4
[1 1 0]	jθ + jπ6/4	jθ + jπ4/4
[1 1 1]	jθ + jπ7/4	jθ + jπ5/4

For more details about the `Binary` and `Gray` options, see the reference page for the M-PSK Modulator Baseband block. The signal constellation for that block corresponds to the arrangement of phase differences for this block.

Dialog Box



M-DPSK Modulator Baseband (mask) (link)

Modulate the input signal using the differential phase shift keying method.

This block accepts a scalar or column vector input signal.

The input signal can be either bits or integers. When you set the 'Input type' parameter to 'Bit', the input width must be an integer multiple of the number of bits per symbol.

Parameters

M-ary number: 8

Input type: Integer

Constellation ordering: Gray

Phase rotation (rad): pi/8

Output data type: double

M-ary number

The number of possible output symbols that can immediately follow a given output symbol.

Input type

Indicates whether the input consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Constellation ordering

Determines how the block maps each group of input bits to a corresponding integer.

Phase rotation (rad)

The phase difference between the previous and current modulated symbols when the input is zero.

Output data type

The output data type can be either `single` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean (binary input mode only) • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Output	• Double-precision floating point • Single-precision floating point

Pair Block

M-DPSK Demodulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

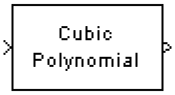
Blocks

DBPSK Modulator Baseband | DQPSK Modulator Baseband | M-DPSK Demodulator Baseband | M-PSK Modulator Baseband

Introduced before R2006a

Memoryless Nonlinearity

Apply memoryless nonlinearity to complex baseband signal
Library: Communications Toolbox / RF Impairments



Description

The Memoryless Nonlinearity block applies memoryless nonlinear impairments to a complex baseband signal. Use this block to model memoryless nonlinear impairments caused by signal amplification in the radio frequency (RF) transmitter or receiver. For more information, see “Memoryless Nonlinear Impairments” on page 5-486.

Note All values of power assume a nominal impedance of 1 ohm.

Ports

Input

`In1` — Input RF baseband signal
scalar | column vector

Input RF baseband signal, specified as a scalar or column vector. Values in this input must be complex.

Data Types: `double`
Complex Number Support: Yes

Output

`Out1` — Output RF baseband signal
scalar | column vector

Output RF baseband signal, returned as a scalar or column vector. The output is of the same data type as the input.

Parameters

`Method` — Nonlinearity modeling method
`Cubic polynomial` (default) | `Hyperbolic tangent` | `Saleh model` | `Ghorbani model` | `Rapp model`

Nonlinearity modeling method, specified as `Cubic polynomial`, `Hyperbolic tangent`, `Saleh model`, `Ghorbani model`, `Rapp model`, or `Lookup table`. For more information, see “Memoryless Nonlinear Impairments” on page 5-486.

`Linear gain (dB)` — Linear gain
`0` (default) | scalar

Linear gain in decibels, specified as a scalar. This parameter scales the power gain of the output signal.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Cubic polynomial`, `Hyperbolic tangent`, or `Rapp model`.

Data Types: `double`

`IIP3 (dBm)` — Third-order input intercept point

`30` (default) | scalar

Third-order input intercept point in dBm, specified as a scalar.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Cubic polynomial` or `Hyperbolic tangent`.

Data Types: `double`

`AM/PM conversion (degrees per dB)` — AM/PM conversion factor

`10` (default) | scalar

AM/PM conversion factor in degrees per decibel, specified as a scalar. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 5-487.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Cubic polynomial` or `Hyperbolic tangent`.

Data Types: `double`

`Lower input power limit for AM/PM conversion (dBm)` — Input power lower limit

`10` (default) | scalar

Input power lower limit in dBm, specified as a scalar less than the **Upper input power limit for AM/PM conversion (dBm)** parameter value. The AM/PM conversion scales linearly for input power values in the range [**Lower input power limit for AM/PM conversion (dBm)**, **Upper input power limit for AM/PM conversion (dBm)**]. If the input signal power is below the input power lower limit, the phase shift resulting from AM/PM conversion is zero. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 5-487.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Cubic polynomial` or `Hyperbolic tangent`.

Data Types: `double`

`Upper input power limit for AM/PM conversion (dBm)` — Input power upper limit

`inf` (default) | scalar

Input power upper limit in dBm, specified as a scalar greater than the **Lower input power limit for AM/PM conversion (dBm)** parameter value. The AM/PM conversion scales linearly for input power values in the range [**Lower input power limit for AM/PM conversion (dBm)**, **Upper input power limit for AM/PM conversion (dBm)**]. If the input signal power is below the input power lower limit, the phase shift resulting from AM/PM conversion is zero. For more information, see “Cubic Polynomial and Hyperbolic Tangent Model Methods” on page 5-487.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Cubic polynomial` or `Hyperbolic tangent`.

Data Types: `double`

`Input scaling (dB)` — Input signal scaling factor

`0` (default) | scalar

Input signal scaling factor in decibels, specified as a scalar. This parameter scales the power gain of the input signal.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Saleh model` or `Ghorbani model`.

Data Types: `double`

`AM/AM parameters [alpha beta]` — AM/AM parameters for Saleh model

`[2.1587 1.1517]` (default) | two-element vector

AM/AM parameters for Saleh model, used to compute the amplitude gain for an input signal, specified as a two-element vector. For more information, see “Saleh Model Method” on page 5-488.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Saleh model`.

Data Types: `double`

`AM/PM parameters [alpha beta]` — AM/PM parameters for Saleh model

`[4.0033 9.1040]` (default) | two-element vector

AM/PM parameters for Saleh model, used to compute the phase change for an input signal, specified as a two-element vector. For more information, see “Saleh Model Method” on page 5-488.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Saleh model`.

`AM/AM parameters [x1 x2 x3 x4]` — AM/AM parameters for Ghorbani model

`[8.1081 1.5413 6.5202 -0.0718]` (default) | four-element vector

AM/AM parameters for Ghorbani model, used to compute the amplitude gain for an input signal, specified as a four-element vector. For more information, see "Ghorbani Model Method" on page 5-488.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Ghorbani model`.

Data Types: `double`

`AM/PM parameters [y1 y2 y3 y4]` — AM/PM parameters for Ghorbani model

`[4.6645 2.0965 10.88 -0.003]` (default) | four-element vector

AM/PM parameters for Ghorbani model, used to compute the phase change for an input signal, specified as a four-element vector. For more information, see "Ghorbani Model Method" on page 5-488.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Ghorbani model`.

Data Types: `double`

`Output scaling (dB)` — Output signal scaling factor

`0` (default) | scalar

Output signal scaling factor in decibels, specified as a scalar. This parameter scales the power gain of the output signal.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Saleh model` or `Ghorbani model`.

Data Types: `double`

`Smoothness factor` — Smoothness factor

`0.5` (default) | scalar

Smoothness factor, specified as a scalar. For more information, see "Rapp Model Method" on page 5-489.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Rapp model`.

Data Types: `double`

`Output saturation level` — Output saturation level

`1` (default) | scalar

Output saturation level, specified as a scalar. For more information, see “Rapp Model Method” on page 5-489.

Tunable: Yes

Dependencies

To enable this parameter, set the **Method** to `Rapp model`.

Data Types: `double`

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Memoryless Nonlinear Impairments

Memoryless nonlinear impairments distort the input signal amplitude and phase. The amplitude distortion is amplitude-to-amplitude modulation (AM/AM) and the phase distortion is amplitude-to-phase modulation (AM/PM).

Model Method	Memoryless Nonlinear Impairment
Cubic polynomial	AM/AM and AM/PM
Hyperbolic tangent	
Saleh model	
Ghorbani model	
Rapp model	AM/AM only

The modeled impairments apply the AM/AM and AM/PM distortions differently, according to the model method you specify. The models apply the memoryless nonlinear impairment to the input signal by following these steps.

1. Multiply the signal by an input gain factor.

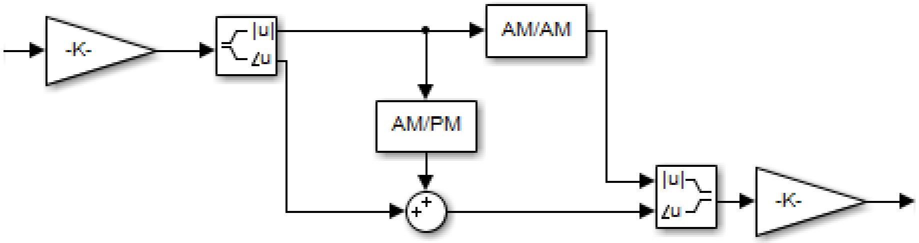
   Note You can normalize the signal to 1 by setting the input scaling gain to the inverse of the input signal amplitude.

2. Split the complex signal into its magnitude and angle components.
3. Apply an AM/AM distortion to the magnitude of the signal, according to the selected model method, to produce the magnitude of the output signal.
4. Apply an AM/PM distortion to the phase of the signal, according to the selected model method, to produce the angle of the output signal.

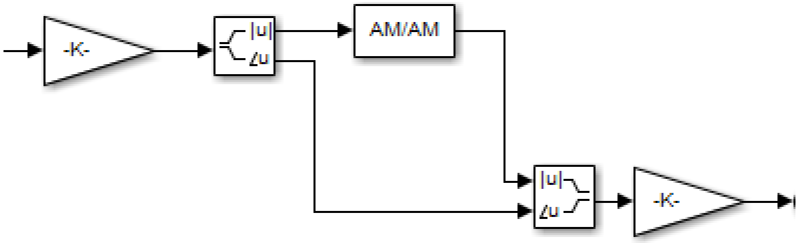
   Note This step does not apply for the Rapp model.

5 Combine the new magnitude and angle components into a complex signal. Then, multiply the result by an output gain factor.

The first four model methods (cubic polynomial, hyperbolic tangent, Saleh model, and Ghorbani model) apply AM/AM and AM/PM impairments as shown in this figure.

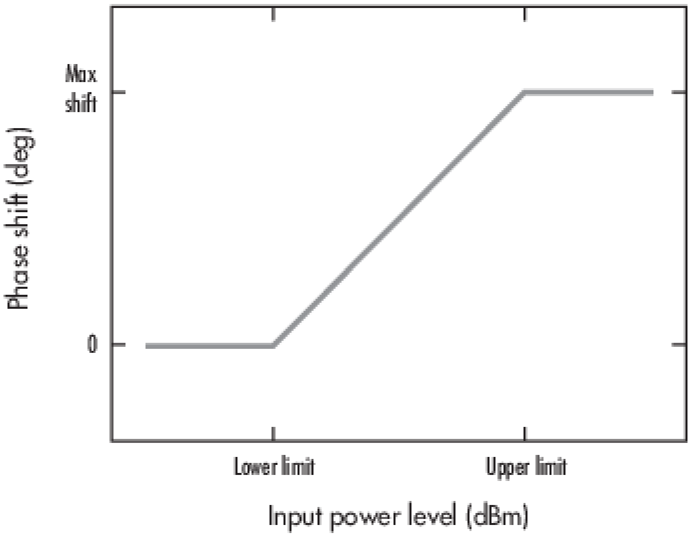


The Rapp model method applies AM/AM distortion as shown in this figure.



Cubic Polynomial and Hyperbolic Tangent Model Methods

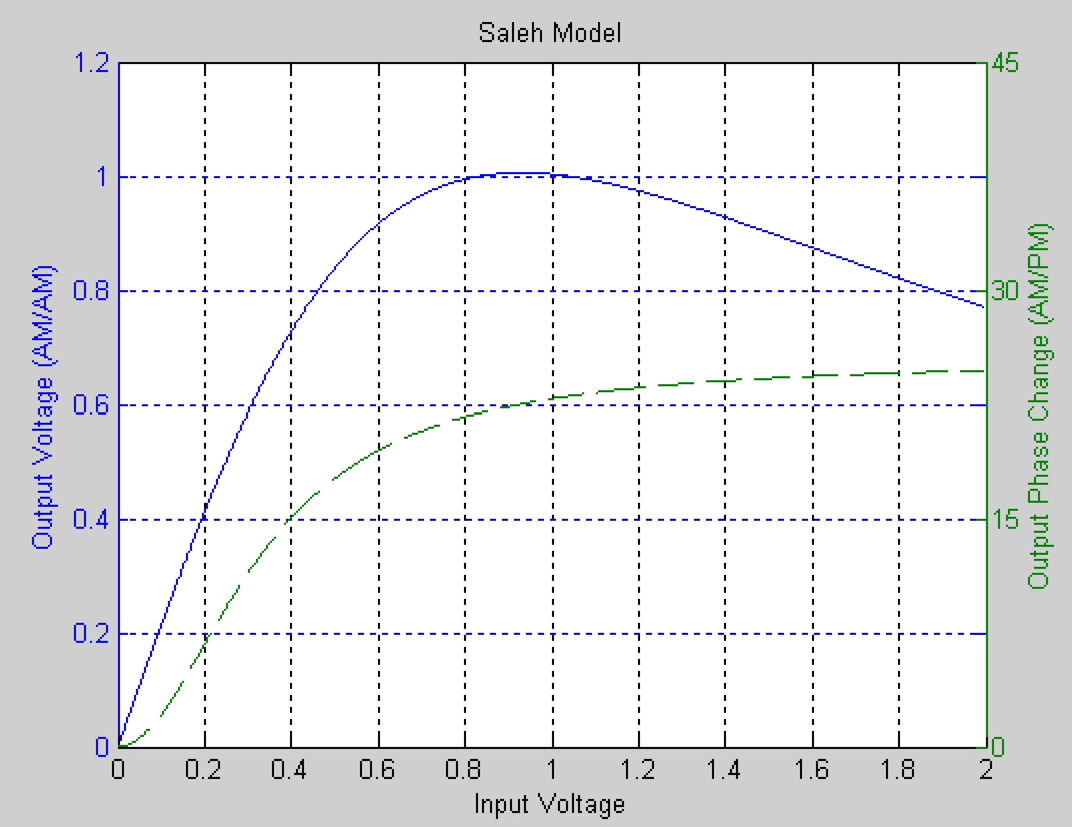
This figure shows the AM/PM conversion behavior for the cubic polynomial and hyperbolic tangent model methods.



The AM/PM conversion scales linearly with an input power value between the lower and upper limits of the input power level. Outside this range, the AM/PM conversion is constant at the values corresponding to the lower and upper input power limits, which are zero and (*AM/PM conversion*) × (*upper input power limit – lower input power limit*), respectively.

Saleh Model Method

This figure shows the AM/AM behavior (output voltage versus input voltage for the AM/AM distortion) and the AM/PM behavior (output phase versus input voltage for the AM/PM distortion) for the Saleh model method.



The AM/AM parameters, α_{AMAM} and β_{AMAM}, are used to compute the amplitude distortion of the input signal by using

$$F_{\mathrm{AMAM}}(u) = \frac{\alpha_{\mathrm{AMAM}} \times u}{1 + \beta_{\mathrm{AMAM}} \times u^2},$$

where u is the magnitude of the scaled signal.

The AM/PM parameters, α_{AMPM} and β_{AMPM}, are used to compute the phase distortion of the input signal by using

$$F_{\mathrm{AMPM}}(u) = \frac{\alpha_{\mathrm{AMPM}} \times u^2}{1 + \beta_{\mathrm{AMPM}} \times u^2},$$

where u is the magnitude of the scaled signal. The α and β parameters for AM/AM and AM/PM are similarly named but distinct.

Ghorbani Model Method

The Ghorbani model method applies AM/AM and AM/PM distortion as described in this section.

The AM/AM parameters (x_1, x_2, x_3, and x_4) are used to compute the amplitude distortion of the input signal by using

$$F_{\mathrm{AMAM}}(u) = \frac{x_1 u^{x_2}}{1 + x_3 u^{x_2}} + x_4 u,$$

where u is the magnitude of the scaled signal.

The AM/PM parameters (y_1, y_2, y_3, and y_4) are used to compute the phase distortion of the input signal by using

$$F_{\mathrm{AMPM}}(u) = \frac{y_1 u^{y_2}}{1 + y_3 u^{y_2}} + y_4 u,$$

where u is the magnitude of the scaled signal.

Rapp Model Method

The Rapp model method applies AM/AM distortion as described in this section. The Rapp model does not apply AM/PM distortion to the input signal.

The smoothness factor and output saturation level are used to compute the amplitude distortion of the input signal given by

$$F_{\mathrm{AMAM}}(u) = \frac{u}{\left(1 + \left(\frac{u}{O_{\mathrm{sat}}}\right)^{2\mathrm{S}}\right)^{1/2\mathrm{S}}},$$

where

- u is the magnitude of the scaled signal.
- S is the smoothness factor.
- O_{sat} is the output saturation level.

References

[1] Saleh, A.A.M. “Frequency-Independent and Frequency-Dependent Nonlinear Models of TWT Amplifiers.” *IEEE Transactions on Communications* 29, no. 11 (November 1981): 1715–20. https://doi.org/10.1109/TCOM.1981.1094911.

[2] Ghorbani, A., and M. Sheikhan. "The Effect of Solid State Power Amplifiers (SSPAs) Nonlinearities on MPSK and M-QAM Signal Transmission." In *1991 Sixth International Conference on Digital Processing of Signals in Communications*, 193–97, 1991.

[3] Rapp, Ch. "Effects of HPA-Nonlinearity on a 4-DPSK/OFDM-Signal for a Digital Sound Broadcasting System." In *Proceedings Second European Conf. on Sat. Comm. (ESA SP-332)*, 179–84. Liege, Belgium, 1991. https://elib.dlr.de/33776/.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

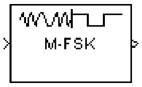
Blocks
I/Q Imbalance

Objects
`comm.MemorylessNonlinearity`

Introduced before R2006a

M-FSK Demodulator Baseband

Demodulate FSK-modulated data



Library

FM, in Digital Baseband sublibrary of Modulation

Description

The M-FSK Demodulator Baseband block demodulates a signal that was modulated using the M-ary frequency shift keying method. The input is a baseband representation of the modulated signal. The input and output for this block are discrete-time signals. This block accepts a scalar value or column vector input signal of type `single` or `double`. For information about the data types each block port supports, see "Supported Data Types" on page 5-494.

The **M-ary number** parameter, M, is the number of frequencies in the modulated signal. The **Frequency separation** parameter is the distance, in Hz, between successive frequencies of the modulated signal.

The M-FSK Demodulator Baseband block implements a non-coherent energy detector. To obtain the same BER performance as that of coherent FSK demodulation, use the CPFSK Demodulator Baseband block.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, the block outputs integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where *K* represents the number of bits per symbol. The output vector length must be an integer multiple of *K*.

The **Symbol set ordering** parameter indicates how the block maps a symbol to a group of *K* output bits. When you set the parameter to `Binary`, the block maps the integer, I, to [*u*(1) *u*(2) ... *u*(K)] bits, where the individual u(i) are given by

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

u(1) is the most significant bit.

For example, if *M* = 8, you set **Symbol set ordering** to `Binary`, and the demodulated integer symbol value is 6, then the binary output word is [1 1 0].

When you set **Symbol set ordering** to `Gray`, the block assigns binary outputs from points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
```

to the *P*th integer.

The typical Binary to Gray mapping for *M* = 8 is shown in the following tables.

Binary to Gray Mapping for Bits

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Binary to Gray Mapping for Integers

Binary Code	Gray Code
0	0
1	1
2	3
3	2
4	6
5	7
6	5
7	4

Whether the output is an integer or a binary representation of an integer, the block maps the highest frequency to the integer 0 and maps the lowest frequency to the integer M-1. In baseband simulation, the lowest frequency is the negative frequency with the largest absolute value.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is *K* times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

To run the M-FSK Demodulator block in multirate mode, clear the **Treat each discrete rate as a separate task** checkbox (in **Simulation > Configuration Parameters > Solver**).

Parameters

M-ary number

The number of frequencies in the modulated signal.

Output type

Determines whether the output consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Symbol set ordering

Determines how the block maps each integer to a group of output bits.

Frequency separation (Hz)

The distance between successive frequencies in the modulated signal.

Samples per symbol

The number of input samples that represent each modulated symbol.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample times. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Output data type

The output type of the block can be specified here as `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Pair Block

M-FSK Modulator Baseband

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications.* Upper Saddle River, NJ: Prentice-Hall, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

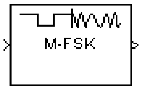
Blocks

CPFSK Demodulator Baseband | M-FSK Modulator Baseband

Introduced before R2006a

M-FSK Modulator Baseband

Modulate using M-ary frequency shift keying method



Library

FM, in Digital Baseband sublibrary of Modulation

Description

The M-FSK Modulator Baseband block modulates using the M-ary frequency shift keying method. The output is a baseband representation of the modulated signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-498.

To prevent aliasing from occurring in the output signal, set the sampling frequency greater than the product of *M* and the **Frequency separation** parameter. Sampling frequency is **Samples per symbol** divided by the input symbol period (in seconds).

Integer-Valued Signals and Binary-Valued Signals

The input and output signals for this block are discrete-time signals.

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol, oversampled by the **Samples per symbol** parameter value, for each group of *K* bits.

The **Symbol set ordering** parameter indicates how the block maps a group of *K* input bits to a corresponding symbol. When you set the parameter to `Binary`, the block maps [*u*(1) *u*(2) ... *u*(K)] to the integer

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

and assumes that this integer is the input value. *u*(1) is the most significant bit.

If you set M = 8, **Symbol set ordering** to `Binary`, and the binary input word is [1 1 0], the block converts [1 1 0] to the integer 6. The block produces the same output when the input is 6 and the **Input type** parameter is `Integer`.

When you set **Symbol set ordering** to `Gray`, the block uses a Gray-coded arrangement and assigns binary inputs to points of a predefined Gray-coded signal constellation. The predefined M-ary Gray-coded signal constellation assigns the binary representation

```
M = 8; P = [0:M-1]';
de2bi(bitxor(P,floor(P/2)), log2(M),'left-msb')
```

to the P^{th} integer.

The following tables show the typical Binary to Gray mapping for M = 8.

Binary to Gray Mapping for Bits

Binary Code	Gray Code
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

Binary to Gray Mapping for Integers

Binary Code	Gray Code
0	0
1	1
2	3
3	2
4	6
5	7
6	5
7	4

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of K, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

To run the M-FSK Modulator block in multirate mode, clear the **Treat each discrete rate as a separate task** checkbox (in **Simulation > Configuration Parameters > Solver**).

Parameters

M-ary number

The number of frequencies in the modulated signal.

Input type

Indicates whether the input consists of integers or groups of bits. If you set this parameter to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Symbol set ordering

Determines how the block maps each group of input bits to a corresponding integer.

Frequency separation (Hz)

The distance between successive frequencies in the modulated signal.

Phase continuity

Determines whether the modulated signal changes phases in a continuous or discontinuous way.

If you set the **Phase continuity** parameter to `Continuous`, then the modulated signal maintains its phase even when it changes its frequency. If you set the **Phase continuity** parameter to `Discontinuous`, then the modulated signal comprises portions of *M* sinusoids of different frequencies. Thus, a change in the input value sometimes causes a change in the phase of the modulated signal.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Output data type

You can specify the output type of the block as either a `double` or a `single`. By default, the block sets this value to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (bit input mode only) • 8-, 16-, and 32-bit signed integers (integer input mode only) • 8-, 16-, and 32-bit unsigned integers (integer input mode only)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

M-FSK Demodulator Baseband

References

[1] Sklar, Bernard. *Digital Communications: Fundamentals and Applications.* Upper Saddle River, NJ: Prentice-Hall, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CPFSK Modulator Baseband | M-FSK Demodulator Baseband

Introduced before R2006a

MIL-188 QAM Demodulator Baseband

MIL-STD-188-110 B/C standard-specific quadrature amplitude demodulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / AM
Communications Toolbox / Modulation / Digital Baseband Modulation / Standard-Compliant

Description

The MIL-188 QAM Demodulator Baseband block demodulates the input signal using “MIL-STD-188-110” on page 5-502 standard-specific quadrature amplitude modulation (QAM). For a description of MIL-STD-188 compliant demodulation, see “MIL-STD-188-110 QAM Hard Demodulation” on page 5-502 and “MIL-STD-188-110 QAM Soft Demodulation” on page 5-503.

This icon shows the block with all ports enabled:

In
MIL-188-QAM
Var

Ports

Input

In — MIL-STD-188 standard-specific QAM modulated signal

scalar | vector | matrix

MIL-STD-188 standard-specific QAM modulated signal, specified as a scalar, vector, or matrix. When this input is a matrix, each column is treated as an independent channel. This port is unnamed until the `Var` port is enabled.

Data Types: `single` | `double`
Complex Number Support: Yes

`Var` — Noise variance

positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values. When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “MIL-STD-188-110 QAM Soft Demodulation” on page 5-503 for demodulation decision type considerations.

Dependencies

To enable this port set the Noise variance source parameter to `Input port`.

Output

`Out` — Demodulated signal
scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The dimensions of the demodulated signal depend on the specified Output type and Decision type parameter values. This port is unnamed on the block.

Output type	Decision type	Demodulated Signal Description	Dimensions of Demodulated Signal
`Integer`	—	Demodulated integer values in the range [0, (M – 1)]	The output signal has the same dimensions as input signal.
`Bit`	`Hard decision`	Demodulated bits	The number of rows in the output signal is $\log_2(M)$ times the number of rows in the input signal. Each demodulated symbol is mapped to a group of $\log_2(M)$ elements in a column, where the first element represents the MSB, and the last element represents the LSB.
	`Log-likelihood ratio`	Log-likelihood ratio value for each bit	
	`Approximate log-likelihood ratio`	Approximate log-likelihood ratio value for each bit	
M is the value of Modulation order.			

Use Output data type to specify the output data type.

Parameters

`Modulation order` — Modulation order
`16` (default) | `32` | `64` | `256`

Modulation order, M, specified as `16`, `32`, `64`, or `256`. The modulation order specifies the total number of points in the constellation of the input signal.

`Constellation scaling` — Constellation scaling
`As specified in standard` (default) | `Unit average power`

Constellation scaling preference, specified as:

- `As specified in standard` – The block scales the constellation based on specifications in the relevant standard [1].
- `Unit average power` – The block scales the constellation to an average power of 1 watt referenced to 1 ohm.

`Output type` — Input type
`Integer` (default) | `Bit`

Output type, specified as `Integer` or `Bit`. To use `Integer`, the input signal must consist of integers in the range [0, (M – 1)]. To use `Bit`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2(M)$, where M is the Modulation order.

Decision type — Demodulation decision type

Hard decision (default) | Log-likelihood ratio | Approximate log-likelihood ratio

Demodulation decision type, specified as Hard decision, Log-likelihood ratio, or Approximate log-likelihood ratio. See “MIL-STD-188-110 QAM Soft Demodulation” on page 5-503 for algorithm selection considerations.

Dependencies

This parameter applies when Output type is set to Bit.

Noise variance source — Noise variance source

Property (default) | Input port

Noise variance source, specified as:

- Property — The noise variance is set using the Noise variance parameter.
- Input port — The noise variance is set using the Var input port.

Dependencies

This parameter applies only when Decision type is set to either Log-likelihood ratio or Approximate log-likelihood ratio.

Noise variance — Noise variance

1 (default) | positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values.

- When specified as a scalar, that value is used on all elements in the input signal.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal. Each noise variance vector element is applied to its corresponding column in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “MIL-STD-188-110 QAM Soft Demodulation” on page 5-503 for demodulation decision type considerations.

Dependencies

This parameter applies only when Noise variance is set to Property and Decision type is set to either Log-likelihood ratio or Approximate log-likelihood ratio.

Data Types: double

Output data type — Output data type

double (default) | ...

Output data type, specified as one of the acceptable values from this table. Acceptable Output data type values depend on the Output type and Decision type parameter values.

Output type	Decision type	Output data type Options
Integer	Not applicable	double, single, int8, uint8, int16, uint16, int32, or uint32

Output type	Decision type	Output data type Options
`Bit`	`Hard decision`	`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `logical`
	`Log-likelihood ratio` or `Approximate log-likelihood ratio`	The output signal is the same data type as the input signal.

Dependencies

This parameter applies only when Output type is set to `Integer` or when Output type is set to `Bit` and Decision type is set to `Hard decision`.

`Simulate using` — Type of simulation to run

`Interpreted execution` (default) | `Code generation`

Type of simulation to run, specified as:

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

For information on execution speed, see "Tips" on page 5-503.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`

More About

MIL-STD-188-110

MIL-STD-188-110 is a US Department of Defense standard for HF communications using serial PSK mode of both data and voice signals.

The standard specifies physical layer modulation schemes for tactical and long-haul communications. The modulation scheme specified by the standard is a mix of QAM and APSK. For a detailed description of the modulation scheme, see [1].

MIL-STD-188-110 QAM Hard Demodulation

The hard demodulation algorithm uses optimum decision region-based demodulation. Since all the constellation points are equally probable, maximum a posteriori probability (MAP) detection reduces to a maximum likelihood (ML) detection. The ML detection rule is equivalent to choosing the closest constellation point to the received symbol. The decision region for each constellation point is designed by drawing perpendicular bisectors between adjacent points. A received symbol is mapped to the proper constellation point based on which decision region it lies in.

Since all MIL-STD constellations are quadrant-based symmetric, for each symbol the optimum decision region-based demodulation:

- Maps the received symbol into the first quadrant
- Chooses the decision region for the symbol
- Maps the constellation point back to its original quadrant using the sign of real and imaginary parts of the received symbol

MIL-STD-188-110 QAM Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

Tips

- For faster execution of the MIL-188 QAM Demodulator Baseband block, set the Simulate using parameter to:
  - `Code generation` when using hard decision demodulation.
  - `Interpreted execution` when using soft decision demodulation.

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems." Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
DVBS-APSK Demodulator Baseband | M-APSK Demodulator Baseband | MIL188-QAM Modulator Baseband

Functions
`mil188qamdemod`

Topics
“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018b

MIL-188 QAM Modulator Baseband

MIL-STD-188-110 B/C standard-specific quadrature amplitude modulation (QAM)

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / AM
Communications Toolbox / Modulation / Digital Baseband Modulation / Standard-Compliant

Description

The MIL-188 QAM Modulator Baseband block modulates the input signal using “MIL-STD-188-110” on page 5-507 standard-specific quadrature amplitude modulation (QAM).

Ports

Input

In — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The input signal must be binary values or integers in the range [0, (*M* – 1)], where *M* is the Modulation order. This port is unnamed on the block.

Note To process the input signal as binary elements, set the Input type parameter value to `Bit`. For binary inputs, the number of rows must be an integer multiple of $\log_2(M)$. Groups of $\log_2(M)$ bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Out` — MIL-STD-188 standard-specific QAM modulated signal
scalar | vector | matrix

MIL-STD-188 standard-specific QAM modulated signal, returned as a complex scalar, vector, or matrix. The output signal dimensions depend on the specified Input type parameter value. This port is unnamed on the block.

Input type	Dimensions of Output Signal
`Integer`	The output signal has the same dimensions as the input signal.

Input type	Dimensions of Output Signal
`Bit`	The number of rows in the output signal equals the number of rows in the input signal divided by $\log_2(M)$, where M is the Modulation order.

Parameters

`Modulation order` — Modulation order
16 (default) | 32 | 64 | 256

Modulation order, M, specified as `16`, `32`, `64`, or `256`. The modulation order specifies the total number of points in the constellation of the output signal.

`Constellation scaling` — Constellation scaling
`As specified in standard` (default) | `Unit average power`

Constellation scaling preference, specified as:

- `As specified in standard` – The block scales the constellation based on specifications in the relevant standard [1].
- `Unit average power` – The block scales the constellation to an average power of 1 watt referenced to 1 ohm.

`Input type` — Input type
`Integer` (default) | `Bit`

Input type, specified as `Integer` or `Bit`. To use `Integer`, the input signal must consist of integers in the range [0, (M – 1)]. To use `Bit`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2(M)$, where M is the Modulation order.

`Output data type` — Output data type
`double` (default) | `single`

Output data type, specified as `double` or `single`.

`View Constellation` — Plot reference constellation
`button`

To plot the reference constellation, click the **View Constellation** button.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as:

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time and has a simulation speed comparable to `Code generation`. In `Interpreted execution` mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`

More About

MIL-STD-188-110

MIL-STD-188-110 is a US Department of Defense standard for HF communications using serial PSK mode of both data and voice signals.

The standard specifies physical layer modulation schemes for tactical and long-haul communications. The modulation scheme specified by the standard is a mix of QAM and APSK. For a detailed description of the modulation scheme, see [1].

References

[1] MIL-STD-188-110B & C: "Interoperability and Performance Standards for Data Modems." Department of Defense Interface Standard, USA.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DVBS-APSK Modulator Baseband | M-APSK Modulator Baseband | MIL188-QAM Demodulator Baseband

Functions

`mil188qammod`

Introduced in R2018b

MIMO Fading Channel

Filter input signal through MIMO multipath fading channel

Library: Communications Toolbox / Channels
Communications Toolbox / MIMO

Description

The MIMO Fading Channel block filters an input signal using a multi-input/multi-output (MIMO) multipath fading channel. This block models both Rayleigh and Rician fading and employs the Kronecker model for modeling the spatial correlation between the links. For processing details, see the Algorithms on page 5-517 section.

Signal Dimensions

The availability and dimensions of input and output port signals depends on:

- The Antenna selection parameter setting on the **Main** tab
- The Initial time source parameter setting on the **Realization** tab
- The Output channel path gains selection on the **Realization** tab

Antenna Selection Parameter	Signal Input (in)	Transmit Selection Input (Tx Sel)	Receive Selection Input (Rx Sel)	Initial Time Offset Input (Init Time)	Signal Output (Out1)	Optional Channel Gain Output (Gain)
`Off`	N_S-by-N_T	N/A	N/A	nonnegative scalar	N_S-by-N_R	N_S-by-N_P-by-N_T-by-N_R
`Tx`	N_S-by-N_{ST}	1-by-N_T	N/A		N_S-by-N_R	
`Rx`	N_S-by-N_T	N/A	1-by-N_R		N_S-by-N_{SR}	
`Tx and Rx`	N_S-by-N_{ST}	1-by-N_T	1-by-N_R		N_S-by-N_{SR}	

- N_S represents the number of samples in the input signal.
- N_T represents the number of transmit antennas, as determined by:
  - Transmit spatial correlation when Specify spatial correlation is set to `Separate Tx Rx`
  - Number of transmit antennas when Specify spatial correlation is set to `None` or `Combined`
- N_R represents the number of receive antennas, as determined by:
  - Receive spatial correlation when Specify spatial correlation is set to `Separate Tx Rx`
  - Number of receive antennas when Specify spatial correlation is set to `None`
  - Combined spatial correlation and Number of transmit antennas when Specify spatial correlation is set to `Combined`

- N_P represents the number of channel paths, as determined by the Discrete path delays (s) or Average path gains (dB).
- N_{ST} represents the number of selected transmit antennas, as determined by the number of elements set to `1` in the vector provided to the Tx Sel input port.
- N_{SR} represents the number of selected receive antennas, as determined by the number of elements set to `1` in the vector provided to the Rx Sel input port.

Ports

Input

`in` — Input data signal

vector

Input data signal, specified as an N_S-by-N_T or N_S-by-N_{ST} matrix.

- N_S represents the number of samples in the input signal.
- N_T represents the number of transmit antennas.
- N_{ST} represents the number of selected transmit antennas.

Data Types: `double` | `single`
Complex Number Support: Yes

`Tx Sel` — Select active transmit antennas

binary vector

Select active transmit antennas, specified as a 1-by-N_T binary vector. N_T represents the number of transmit antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Dependencies

To enable this port, on the **Main** tab, set Antenna selection to `Tx` or `Tx and Rx`.

Data Types: `double`

`Rx Sel` — Select active receive antennas

binary vector

Select active receive antennas, specified as a 1-by-N_R binary vector. N_R represents the number of receive antennas. Elements set to `1` identify selected antenna indices and `0` identify nonselected antenna indices.

Dependencies

To enable this port, on the **Main** tab, set Antenna selection to `Rx` or `Tx and Rx`.

Data Types: `double`

`Init Time` — Initial time offset

nonnegative scalar

Initial time offset for the fading model in seconds, specified as a nonnegative scalar.

Init Time must be greater than the last frame end time. When **Init Time** is not a multiple of `1/Sample Rate (Hz)`, it is rounded up to the nearest sample position.

Dependencies

To enable this port, on the **Realization** tab, set Initial time source to `Input port`.

Data Types: `double`

Output

`Out1` — Output data signal for fading channel
vector

Output data signal for the fading channel, returned as an N_S-by-N_R or N_S-by-N_{SR} matrix.

- N_S represents the number of samples in the input signal.
- N_R represents the number of receive antennas.
- N_{SR} represents the number of selected receive antennas.

`Gain` — Discrete path gains
4-D array

Discrete path gains of the underlying fading process, returned as an N_S-by-N_P-by-N_T-by-N_R array.

- N_S represents the number of samples in the input signal.
- N_P represents the number of channel paths.
- N_T represents the number of transmit antennas.
- N_R represents the number of receive antennas.

Entries for nonselected paths are filled with `NaN`.

Dependencies

To enable this port, on the **Realization** tab, select Output channel path gains.

Parameters

Main Tab

Multipath parameters (frequency selectivity)

`Inherit sample rate from input` — Option to inherit the sample rate from input
on (default) | off

Select this parameter to use the sample rate of the input signal when processing. When `Inherit sample rate from input` is selected, the sample rate is N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

`Sample rate (Hz)` — Input signal sample rate
1 (default) | positive scalar

Input signal sample rate, specified in hertz as a positive scalar. To match the model settings, set the sample rate to N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

Dependencies

This parameter appears when Inherit sample rate from input is not selected.

Data Types: `double`

`Discrete path delays (s)` — Delays for each discrete path

`0` (default) | nonnegative scalar | row vector

Delays for each discrete path in seconds, specified as a nonnegative scalar or row vector.

- When you set **Discrete path delays (s)** to a scalar, the MIMO channel is frequency flat.
- When you set **Discrete path delays (s)** to a vector, the MIMO channel is frequency selective.

Data Types: `double`

`Average path gains (dB)` — Average gain for each discrete path

`0` (default) | scalar | row vector

Average gain for each discrete path in decibels, specified as a scalar or row vector. **Average path gains (dB)** must have the same size as Discrete path delays (s).

Data Types: `double`

`Normalize average path gains to 0 dB` — Option to normalize average path gains to 0 dB

on (default) | off

Select this parameter to normalize the fading processes so that the total power of the path gains, averaged over time, is 0 dB.

`Fading distribution` — Fading distribution of channel

`Rayleigh` (default) | `Rician`

Select the fading distribution of the channel, either `Rayleigh` or `Rician`.

`K-factors` — K-factor of Rician fading channel

`3` (default) | positive scalar | row vector of nonnegative values

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of nonnegative values. N_P equals the value of the Discrete path delays (s) parameter.

- If you set **K-factors** to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of **K-factors**. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set **K-factors** to a row vector, the discrete path corresponding to a positive element of the **K-factors** vector is a Rician fading process with a Rician K-factor specified by that element. The discrete path corresponding to any zero-valued elements of the **K-factors** vector are Rayleigh fading processes. At least one element value must be nonzero.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path Doppler shifts (Hz)` — Doppler shifts for line-of-sight components

`0` (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path Doppler shifts (Hz)** to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set **LOS path Doppler shifts (Hz)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of **LOS path Doppler shifts (Hz)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

`LOS path initial phases (rad)` — Initial phases for line-of-sight components

`0` (default) | scalar | row vector

Initial phases for the line-of-sight component of the Rician fading channel in radians, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path initial phases (rad)** to a scalar, it is the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set **LOS path initial phases (rad)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the elements of **LOS path initial phases (rad)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

Doppler parameters (time dispersion)

`Maximum Doppler shift (Hz)` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

Maximum Doppler shift (Hz) must be smaller than (Sample Rate (Hz)/10)/f_c for each path, where f_c is the cutoff frequency factor of the path. For more information, see “Cutoff Frequency Factor” on page 5-517.

Data Types: `double`

`Doppler spectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this parameter is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.

- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter applies when Maximum Doppler shift (Hz) is greater than zero.

If the Technique for generating fading samples parameter is set to `Sum of sinusoids`, Doppler spectrum must be `doppler('Jakes')`.

Antenna parameters (spatial dispersion)

`Specify spatial correlation` — Spatial correlation mode

`None` (default) | `Separate Tx Rx` | `Combined`

Select the spatial correlation mode: `None`, `Separate Tx Rx`, or `Combined`.

- Choose `'None'` to specify the number of transmit and receive antennas.
- Choose `'Spatial Tx Rx'` to specify the transmit and receive spatial correlation matrices separately. The number of transmit (N_T) and receive (N_R) antennas are derived from the dimensions of the Transmit spatial correlation and Receive spatial correlation parameters, respectively.
- Choose `'Combined'` to specify a single correlation matrix for the whole channel. The product of N_T and N_R is derived from the dimension of Combined spatial correlation.

`Number of transmit antennas` — Number of transmit antennas

2 (default) | positive integer

Number of transmit antennas, specified as a positive integer.

Dependencies

This parameter appears when Specify spatial correlation is `None` or `Combined`.

Data Types: `double`

`Number of receive antennas` — Number of receive antennas

2 (default) | positive integer

Number of receive antennas, specified as a positive integer.

Dependencies

This parameter appears when Specify spatial correlation is `None`.

Data Types: `double`

`Transmit spatial correlation` — Spatial correlation of transmitter

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the transmitter as an N_T-by-N_T matrix or N_T-by-N_T-by-N_P array. N_T is the number of transmit antennas, and N_P equals the value of the Discrete path delays (s) parameter.

- If **Discrete path delays (s)** is a scalar, the channel is frequency flat, and **Transmit spatial correlation** is an N_T-by-N_T Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.

- If **Discrete path delays (s)** is a vector, the channel is frequency selective, and you can specify **Transmit spatial correlation** as a matrix. Each path has the same transmit spatial correlation matrix.
- Alternatively, you can specify **Transmit spatial correlation** as an N_T-by-N_T-by-N_P array, where each path can have its own different transmit spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Separate Tx Rx`.

Data Types: `double`
Complex Number Support: Yes

`Receive spatial correlation` — Spatial correlation of receiver

`[1 0; 0 1]` (default) | matrix | 3-D array

Specify the spatial correlation of the receiver as an N_R-by-N_R matrix or N_R-by-N_R-by-N_P array. N_R is the number of receive antennas, and N_P equals the value of the Discrete path delays (s) parameter.

- If **Discrete path delays (s)** is a scalar, the channel is frequency flat, and **Receive spatial correlation** is an N_R-by-N_R Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If **Discrete path delays (s)** is a vector, the channel is frequency selective, and you can specify **Receive spatial correlation** as a matrix. Each path has the same receive spatial correlation matrix.
- Alternatively, you can specify **Receive spatial correlation** as an N_R-by-N_R-by-N_P array, where each path can have its own different receive spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Separate Tx Rx`.

Data Types: `double`
Complex Number Support: Yes

`Combined spatial correlation` — Combined spatial correlation matrix

`[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]` (default) | matrix | 3-D array

Specify the combined spatial correlation matrix as an N_{TR}-by-N_{TR} matrix or N_{TR}-by-N_{TR}-by-N_P array, where N_{TR} = ($N_T \times N_R$), and N_P equals the number of delay paths specified by the Discrete path delays (s) parameter.

- If Discrete path delays (s) is a scalar, the channel is frequency flat, and **Combined spatial correlation** is an N_{TR}-by-N_{TR} Hermitian matrix. The magnitude of any off-diagonal element must be no larger than the geometric mean of the two corresponding diagonal elements.
- If Discrete path delays (s) is a vector, the channel is frequency selective, and you can specify **Combined spatial correlation** as a matrix. Each path has the same spatial correlation matrix.
- Alternatively, you can specify **Combined spatial correlation** as an N_{TR}-by-N_{TR}-by-N_P array, where each path can have its own different combined spatial correlation matrix.

Dependencies

This parameter appears when Specify spatial correlation is `Combined`.

Data Types: `double`

`Normalize outputs by number of receive antennas` — Normalize channel output

on (default) | off

Select this parameter to normalize the channel outputs by the number of receive antennas.

`Simulate using` — Compilation type

`Interpreted execution` (default) | `Code generation`

Compilation type, specified as `Interpreted execution` or `Code generation`.

`Antenna selection` — Antenna mode

`Off` (default) | `Tx` | `Rx` | `Tx and Rx`

The antenna mode you select corresponds to additional input ports on the block.

Antenna selection Setting	Input Ports Added
`Off`	**None**
`Tx`	**Tx Sel**
`Rx`	**Rx Sel**
`Tx and Rx`	**Tx Sel**, **Rx Sel**

Realization Tab

`Technique for generating fading samples` — Channel modeling technique

`Filtered Gaussian noise` (default) | `Sum of sinusoids`

Select the channel modeling technique, either `Filtered Gaussian noise` or `Sum of sinusoids`.

`Number of sinusoids` — Number of sinusoids used

`48` (default) | positive integer

Number of sinusoids used to model the fading process, specified as a positive integer.

Dependencies

This parameter appears when Technique for generating fading samples is `Sum of sinusoids`.

`Initial time source` — Source of initial time offset

`Property` (default) | `Input port`

Indicate the source of the initial time offset for the fading model, either `Property` or `Input port`.

- When you set **Initial time source** to `Property`, use Initial time (s) to set the initial time offset.
- When you set **Initial time source** to `Input port`, use the input port Init Time to set the initial time offset.

Dependencies

This parameter appears when Technique for generating fading samples is `Sum of sinusoids`.

`Initial time (s)` — Initial time offset

`0` (default) | nonnegative scalar

Initial time offset for the fading model, specified as a nonnegative scalar.

When `Initial time (s)` is not a multiple of `1/`Sample Rate (Hz), it is rounded up to the nearest sample position.

Dependencies

This parameter appears when Technique for generating fading samples is `Sum of sinusoids` and Initial time source is set to `Property`.

`Initial seed` — Random number generator initial seed

`73` (default) | nonnegative integer

Random number generator initial seed for this block, specified as a nonnegative integer.

`Output channel path gains` — Option to output channel path gains

off (default) | on

Select this parameter to add the Gain output port to the block and output the channel path gains of the underlying fading process.

Visualization Tab

`Channel visualization` — Select the channel visualization

`Off` (default) | `Impulse response` | `Frequency response` | `Doppler spectrum` | `Impulse and frequency responses`

Select the channel visualization: `Off`, `Impulse response`, `Frequency response`, `Doppler spectrum`, or `Impulse and frequency responses`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

`Antenna pair to display` — Transmit-receive antenna pair to display

`[1,1]` (default) | vector

Transmit-receive antenna pair to display, specified as a 1-by-2 vector, where the first element corresponds to the desired transmit antenna and the second corresponds to the desired receive antenna. At this time, only a single pair can be displayed.

Dependencies

This parameter appears when Channel visualization is not `Off`.

`Percentage of samples to display` — Percentage of samples to display

`25%` (default) | `10%` | `50%` | `100%`

Select the percentage of samples to display: `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This parameter appears when Channel visualization is `Impulse response`, `Frequency response`, or `Impulse and frequency responses`.

`Path for Doppler spectrum display` — Path for which Doppler spectrum is displayed

`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_P, where N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter appears when Channel visualization is `Doppler spectrum`.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`yes`

Algorithms

The fading processing per link is described in Methodology for Simulating Multipath Fading Channels and assumes the same parameters for all ($N_T \times N_R$) links of the MIMO channel. Each link comprises all multipaths for that link.

The Kronecker Model

The Kronecker model assumes that the spatial correlations at the transmit and receive sides are separable. Equivalently, the direction of departure (DoD) and directions of arrival (DoA) spectra are assumed to be separable. The full correlation matrix is:

$$R_H = E[R_t \otimes R_r]$$

- The ⊗ symbol represents the Kronecker product.
- R_t represents the correlation matrix at the transmit side: $R_t = E\left[H^H H\right]$, of size N_T-by-N_T.
- R_r represents the correlation matrix at the receive side: $R_r = E\left[H H^H\right]$, of size N_R-by-N_R.

You can obtain a realization of the MIMO channel matrix as:

$$H = R_r^{\frac{1}{2}} A R_t^{\frac{1}{2}}$$

A is an N_R-by-N_T matrix of independent identically distributed complex Gaussian variables with zero mean and unit variance.

Cutoff Frequency Factor

The cutoff frequency factor, f_c, is determined for different Doppler spectrum types.

- For any Doppler spectrum type other than Gaussian and biGaussian, f_c equals 1.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation` × $\sqrt{2\log2}$.
- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log2}$.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)` × $\sqrt{2\log2}$.

- If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
- In all other cases, f_c equals 1.

Antenna Selection

When the object is in antenna selection mode, it uses the following algorithms to process an input signal:

- All random path gains are always generated and keep evolving for each link, whether or not a given link is selected. The path gain values output for the non-selected links are populated with `NaN`.
- The spatial correlation only applies to the selected transmit and/or receive antennas, and the correlation coefficients are the corresponding entries in the transmit, receive, or combined correlation matrices. In other words, the spatial correlation matrix for the selected transmit or receive antennas is a submatrix of the transmit, receive, or combined spatial correlation matrix property value.
- For signal paths associated with nonactive antennas, a signal with zero power is transmitted to the channel filter.
- Channel output normalization happens over the number of selected receive antennas.

References

[1] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*. Academic Press, 2007.

[2] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*. Academic Press, 2006.

[3] Kermoal, J. P., L. Schumacher, K. I. Pedersen, P. E. Mogensen, and F. Frederiksen. "A stochastic MIMO radio channel model with experimental validation." *IEEE Journal on Selected Areas of Communications*. Vol. 20, Number 6, 2002, pp. 1211–1226.

[4] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*. Second Edition. New York: Kluwer Academic/Plenum, 2000.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

AWGN Channel | SISO Fading Channel

Functions
`doppler`

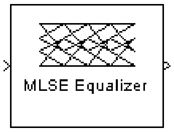
Objects
`comm.MIMOChannel`

Topics
Channel Visualization

Introduced in R2013b

MLSE Equalizer

Equalize using Viterbi algorithm



Library

Equalizer Block

Description

The MLSE Equalizer block uses the Viterbi algorithm to equalize a linearly modulated signal through a dispersive channel. The block processes input frames and outputs the maximum likelihood sequence estimate (MLSE) of the signal, using an estimate of the channel modeled as a finite input response (FIR) filter.

This block supports `single` and `double` data types.

Channel Estimates

The channel estimate takes the form of a column vector containing the coefficients of an FIR filter in descending order of powers. The length of this vector is the channel memory, which must be a multiple of the block's **Samples per input symbol** parameter.

To specify the channel estimate vector, use one of these methods:

- Set **Specify channel via** to `Dialog` and enter the vector in the **Channel coefficients** field.
- Set **Specify channel via** to `Input port` and the block displays an additional input port, labeled `Ch`, which accepts a column vector input signal.

Signal Constellation

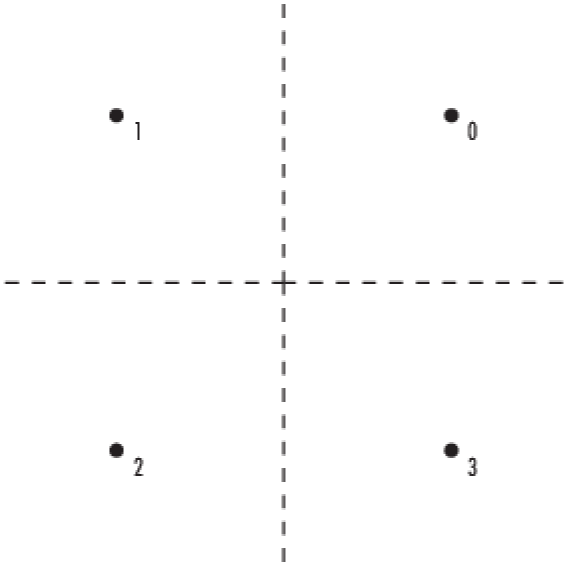
The **Signal constellation** parameter specifies the constellation for the modulated signal, as determined by the modulator in your model. **Signal constellation** is a vector of complex numbers, where the kth complex number in the vector is the constellation point to which the modulator maps the integer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

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

For example, to specify the constellation given by the mapping

$$
\begin{aligned}
0 &\rightarrow +1+i\\
1 &\rightarrow -1+i\\
2 &\rightarrow -1-i\\
3 &\rightarrow +1-i
\end{aligned}
$$

set **Constellation points** to `[1+i, -1+i, -1-i, 1-i]`. Note that the sequence of numbers in the vector indicates how the modulator maps integers to the set of constellation points. The labeled constellation is shown below.



Preamble and Postamble

If your data is accompanied by a preamble (prefix) or postamble (suffix), then configure the block accordingly:

- If you select **Input contains preamble**, then the **Expected preamble** parameter specifies the preamble that you expect to precede the data in the input signal.
- If you check the **Input contains postamble**, then the **Expected postamble** parameter specifies the postamble that you expect to follow the data in the input signal.

The **Expected preamble** or **Expected postamble** parameter must be a vector of integers between 0 and M-1, where M is the number of constellation points. An integer value of k-1 in the vector corresponds to the kth entry in the **Constellation points** vector and, consequently, to a modulator input of k-1.

The preamble or postamble must already be included at the beginning or end, respectively, of the input signal to this block. If necessary, you can concatenate vectors in Simulink software using the Matrix Concatenation block.

To learn how the block uses the preamble and postamble, see “"Reset Every Frame" Operation Mode” on page 5-521 below.

"Reset Every Frame" Operation Mode

One way that the Viterbi algorithm can transition between successive frames is called `Reset every frame` mode. You can choose this mode using the **Operation mode** parameter.

In `Reset every frame` mode, the block decodes each frame of data independently, resetting the state metric at the end of each frame. The traceback decoding always starts at the state with the minimum state metric.

The initialization of state metrics depends on whether you specify a preamble and/or postamble:

- If you do not specify a preamble, the decoder initializes the metrics of all states to 0 at the beginning of each frame of data.
- If you specify a preamble, the block uses it to initialize the state metrics at the beginning of each frame of data. More specifically, the block decodes the preamble and assigns a metric of 0 to the decoded state. If the preamble does not decode to a unique state -- that is, if the length of the preamble is less than the channel memory -- the decoder assigns a metric of 0 to all states that can be represented by the preamble. Whenever you specify a preamble, the traceback path ends at one of the states represented by the preamble.
- If you do not specify a postamble, the traceback path starts at the state with the smallest metric.
- If you specify a postamble, the traceback path begins at the state represented by the postamble. If the postamble does not decode to a unique state, the decoder identifies the smallest of all possible decoded states that are represented by the postamble and begins traceback decoding at that state.

Note In `Reset every frame` mode, the input to the MLSE Equalizer block must contain at least T symbols, not including an optional preamble, where T is the **Traceback depth** parameter.

Continuous Operation Mode

An alternative way that the Viterbi algorithm can transition between successive frames is called `Continuous with reset option` mode. You can choose this mode using the **Operation mode** parameter.

In `Continuous with reset option` mode, the block initializes the metrics of all states to 0 at the beginning of the simulation. At the end of each frame, the block saves the internal state metric for use in computing the traceback paths in the next frame.

If you select **Enable the reset input port**, the block displays another input port, labeled `Rst`. In this case, the block resets the state metrics whenever the scalar value at the `Rst` port is nonzero.

Decoding Delay

The MLSE Equalizer block introduces an output delay equal to the **Traceback depth** in the `Continuous with reset option` mode, and no delay in the `Reset every frame` mode.

Parameters

Specify channel via

The method for specifying the channel estimate. If you select `Input port`, the block displays a second input port that receives the channel estimate. If you select `Dialog`, you can specify the channel estimate as a vector of coefficients for an FIR filter in the **Channel coefficients** field.

Channel coefficients

Vector containing the coefficients of the FIR filter that the block uses for the channel estimate. This field is visible only if you set **Specify channel via** to `Dialog`.

Signal constellation

Vector of complex numbers that specifies the constellation for the modulation.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode

The operation mode of the Viterbi decoder. Choices are `Continuous with reset option` and `Reset every frame`.

Input contains preamble

When checked, you can set the preamble in the **Expected preamble** field. This option appears only if you set **Operation mode** to `Reset every frame`.

Expected preamble

Vector of integers between 0 and M-1 representing the preamble, where M is the size of the constellation. This field is visible and active only if you set **Operation mode** to `Reset every frame` and then select **Input contains preamble**.

Input contains postamble

When checked, you can set the postamble in the **Expected postamble** field. This option appears only if you set **Operation mode** to `Reset every frame`.

Expected postamble

Vector of integers between 0 and M-1 representing the postamble, where M is the size of the constellation. This field is visible and active only if you set **Operation mode** to `Reset every frame` and then select **Input contains postamble**.

Samples per input symbol

The number of input samples for each constellation point.

Enable the reset input port

When you check this box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous with reset option`.

References

[1] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

[2] Steele, Raymond, Ed., *Mobile Radio Communications*, Chichester, England, Wiley, 1996.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CMA Equalizer | LMS Decision Feedback Equalizer | LMS Linear Equalizer | RLS Decision Feedback Equalizer | RLS Linear Equalizer

Functions

`mlseeq`

Objects

`comm.MLSEEqualizer`

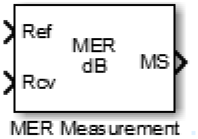
Topics

“MLSE Equalizers”

Introduced before R2006a

MER Measurement

Measure signal-to-noise ratio (SNR) in digital modulation applications



Library

Utility Blocks

Description

The MER Measurement block outputs the modulation error ratio (MER). MER is a measure of the signal-to-noise ratio (SNR) in digital modulation applications. The block measures all outputs in dB.

The MER Measurement block accepts a received signal at input port `Rcv`. It may use an ideal input signal at reference port `Ref` or, optionally, a reference constellation. The MER block then outputs a measure of the modulation accuracy by comparing these inputs. The modulation error ratio is the ratio of the average reference signal power to the mean square error. This ratio corresponds to the SNR of the AWGN channel.

The block output always outputs MER in dB, with an option to output minimum MER and *X*-percentile MER values. The minimum MER represents the best-case MER value per burst. For the *X*-percentile option, you can select to output the number of symbols processed in the percentile computations.

The table shows the output type, the parameter that selects the output type, the computation units, and the corresponding measurement interval.

Output	Activation Parameter	Units	Measurement Interval
MER	None (output by default)	dB	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Minimum MER	**Output minimum MER**	dB	`Current length` | `Entire history` | `Custom` | `Custom with periodic reset`
Percentile MER	**Output X-percentile EVM**	dB	`Entire history`
Number of symbols	**Output X-percentile EVM** and **Output the number of symbols processed**	None	`Entire history`

Data Type

The block accepts double, single, and fixed-point data types. The output of the block is always double.

Algorithms

Parameters

Reference signal

Specifies the reference signal source as either Input port or Estimated from reference constellation.

Reference constellation

Specifies the reference constellation points as a vector. This parameter is available only when **Reference signal** is Estimated from reference constellation. The default is constellation(comm.QPSKModulator).

Measurement interval

Specify the measurement interval as: Input length, Entire history, Custom, or Custom with periodic reset. This parameter affects the RMS and minimum MER outputs only.

- To calculate MER using only the current samples, set this parameter to 'Input length'.
- To calculate MER for all samples, set this parameter to 'Entire history'.
- To calculate MER over an interval you specify and to use a sliding window, set this parameter to 'Custom'.
- To calculate MER over an interval you specify and to reset the object each time the measurement interval is filled, set this parameter to 'Custom with periodic reset'.

Custom measurement interval

Specify the custom measurement interval in samples as a real positive integer. This is the interval over which the MER is calculated. This parameter is available when **Measurement interval** is Custom or Custom with periodic reset. The default is 100.

Averaging dimensions

Specify the dimensions over which to average the MER measurements as a scalar or row vector whose elements are positive integers. For example, to average across the rows, set this parameter to 2. The default is 1.

This block supports var-size inputs of the dimensions in which the averaging takes place. However, the input size for the nonaveraged dimensions must be constant. For example, if the input size is [1000 3 2] and **Averaging dimensions** is [1 3], then the output size is [1 3 1]. The number of elements in the second dimension is fixed at 3.

Output minimum MER

Outputs the minimum MER of an input vector or frame.

Output X-percentile MER

Enables an output *X*-percentile MER measurement. When you select this option, specify **X-percentile value (%)**.

X-Percentile value (%)

This parameter is available only when you select **Output X-percentile MER**. The Xth percentile is the MER value above which X% of all the computed MER values lie. The parameter defaults to the 95th percentile. That is, 95% of all MER values are above this output.

Output the number of symbols processed

Outputs the number of symbols that the block uses to compute the **Output X-percentile MER**. This parameter is available only when you select **Output X-percentile MER**.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

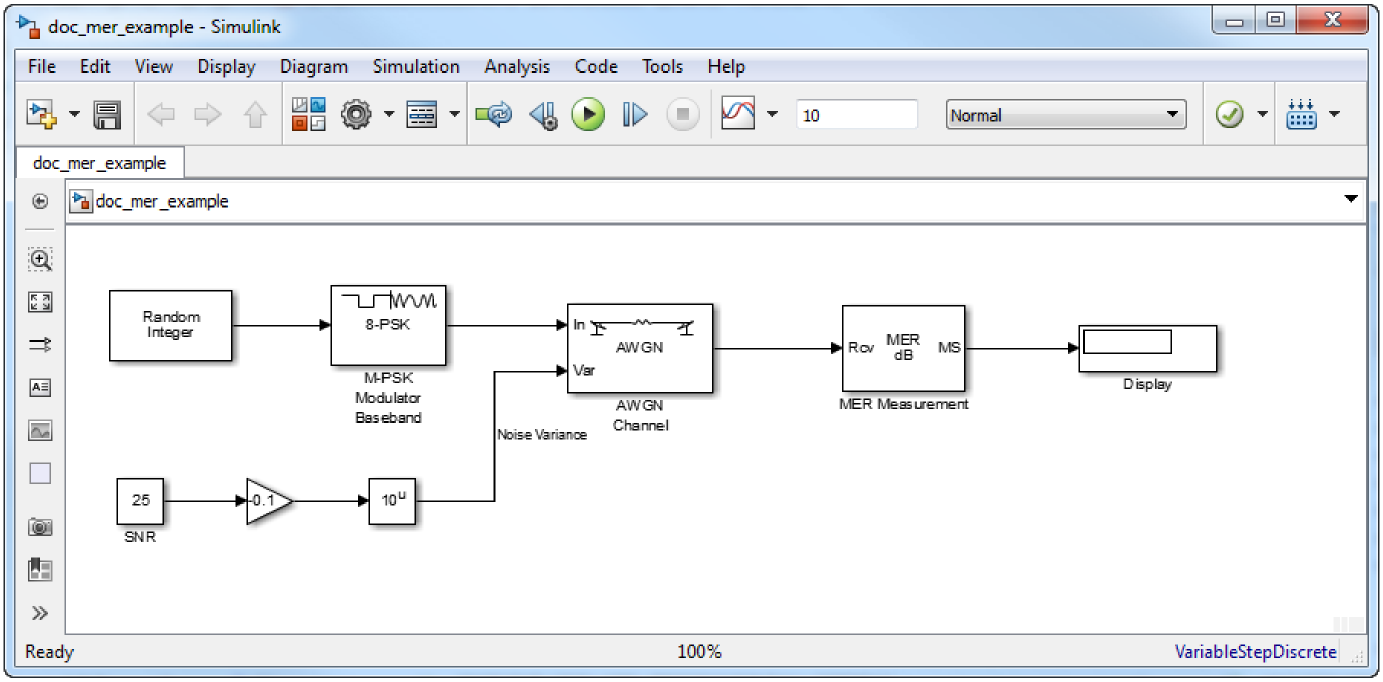
Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

Examples

Measure MER of Noisy PSK Signal

Measure the MER of a noisy 8-PSK signal.

Load the model by typing doc_mer_example at the command line.



Run the model. The MER is shown in the Display block and is approximately equal to the SNR, which is set by using the Constant block. Experiment with different SNR values, and observe the effect on the estimated MER.

Algorithms

MER is a measure of the SNR in a modulated signal calculated in dB. The MER over *N* symbols is

$$\text{MER} = 10 \cdot \log_{10}\left(\frac{\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{\sum_{n=1}^{N}(e_k)}\right)\text{dB},$$

The MER for the *k*th symbol is

$$MER_k = 10 * \log_{10}\left(\frac{\frac{1}{N}\sum_{n=1}^{N}\left(I_k^2 + Q_k^2\right)}{e_k}\right)\text{dB}.$$

The minimum MER represents the minimum MER value in a burst, or

$$MER_{\min} = \min_{k \in [1, \ldots, N]}\{MER_k\},$$

where:

- $e_k = e_k = (I_k - \tilde{I}_k)^2 + (Q_k - \tilde{Q}_k)^2$
- I_k = In-phase measurement of the *k*th symbol in the burst
- Q_k = Quadrature phase measurement of the *k*th symbol in the burst
- I_k and Q_k represent ideal (reference) values. $\tilde{I}_k$ and $\tilde{Q}_k$ represent measured (received) symbols.

The block computes the *X*-percentile MER by creating a histogram of all the incoming MER_k values. The output provides the MER value above which X% of the MER values fall.

References

[1] DVB (ETSI) Standard ETR290. *Digital Video Broadcasting (DVB): Measurement guidelines for DVB systems*. May 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

To generate code in a model using this block, you must enable **Dynamic Memory Allocation in MATLAB Functions**. For more information, see “Dynamic memory allocation in MATLAB functions” (Simulink).

See Also

Blocks
EVM Measurement

Objects
`comm.MER`

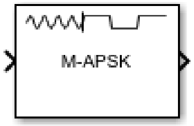
Topics
“EVM and MER Measurements with Simulink”
“Modulation Error Ratio (MER)”

Introduced in R2009b

M-APSK Demodulator Baseband

M-ary amplitude phase shift keying (APSK) demodulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / APM



Description

The M-APSK Demodulator Baseband block demodulates a baseband representation of an *M*-ary amplitude phase shift keying (APSK) modulated signal. *M* is the “Modulation Order for M-APSK” on page 5-534. For a description of *M*-APSK demodulation, see “APSK Hard Demodulation” on page 5-534 and “APSK Soft Demodulation” on page 5-535.

Note M-APSK Demodulator Baseband specifically applies to multiple ring PSK constellations. For a single ring PSK constellation, use M-PSK Demodulator Baseband.

This icon shows the block with all ports enabled:

In
M-APSK
Var

Ports

Input

In — APSK modulated signal
scalar | vector | matrix

APSK modulated signal, specified as a scalar, vector, or matrix. When this input is a matrix, each column is treated as an independent channel. This port is unnamed until the `Var` port is enabled.

Data Types: `double` | `single`
Complex Number Support: Yes

`Var` — Noise variance
positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values. When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “APSK Soft Demodulation” on page 5-535 for demodulation decision type considerations.

Dependencies

To enable this port, set Noise variance source to `Input port`.

Data Types: `double` | `single`

Output

`Out` — Demodulated signal

scalar | vector | matrix

Demodulated signal, returned as a scalar, vector, or matrix. The data type and dimensions of the demodulated signal depend on the values specified by the Output type and Decision type parameters. This port is unnamed on the block.

Output type	Decision type	Demodulated Signal Description	Dimensions of Demodulated Signal
`Integer`	—	Demodulated integer values in the range [0, (M – 1)]	The output signal has the same dimensions as the input signal.
`Bit`	`Hard decision`	Demodulated bits	The number of rows in the output signal is $\log_2(M)$ times the number of rows in the input signal. Each demodulated symbol is mapped to a group of $\log_2(M)$ elements in a column, where the first element represents the MSB and the last element represents the LSB.
	`Log-likelihood ratio`	Log-likelihood ratio value for each bit	
	`Approximate log-likelihood ratio`	Approximate log-likelihood ratio value for each bit	
M is the "Modulation Order for M-APSK" on page 5-534.			

Use Output data type to specify the output data type.

Data Types: `single` | `double`

Parameters

`Constellation points per circle` — Constellation points per PSK ring

`[4,12]` (default) | vector

Constellation points per PSK ring, specified as a vector with more than one element. Each vector element indicates the number of constellation points in its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. The sum of the elements in **Constellation points per circle** determines the modulation order. Element values must be multiples of four, and the modulation order must be a power of two.

Example: `[4,12,16]` specifies a three PSK ring constellation with a modulation order of 32.

`Radius of each circle` — Radius per PSK ring

`[0.5,1]` (default) | vector

Radius per PSK ring, specified as a vector with the same length as Constellation points per circle. Each vector element indicates the radius of its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. These element values must be positive values arranged in increasing order.

Example: `[0.5,1,2]` defines radii for three constellation PSK rings. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

`Phase offset of each circle (rad)` — Phase offset per PSK ring

`[pi/4,pi/12]` (default) | scalar | vector

Phase offset per PSK ring in radians, specified as a scalar or vector with the same length as Constellation points per circle. Each vector element indicates the phase offset of its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. The **Phase offset of each circle (rad)** can be a scalar only if all the elements of **Constellation points per circle** are the same value.

Example: `[pi/4,pi/12,pi/16]` defines phase offsets for three constellation PSK rings. The inner ring has a phase offset of π/4, the second ring has a phase offset of π/12, and the outer ring has a phase offset of π/16.

`Symbol mapping` — Symbol mapping

`Auto` (default) | `Contourwise-gray` | `Gray` | `User-defined`

Symbol mapping, specified as one of the following:

- `Contourwise-gray` — Uses Gray mapping along the contour in the phase dimension for each PSK ring.
- `Gray` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all elements in Constellation points per circle must be equal, and all elements in Phase offset of each circle (rad) must be equal. For a description of the Gray mapping used, see [2].
- `User-defined` — See Custom symbol mapping.

The default symbol mapping depends on the **Constellation points per circle** and **Phase offset of each circle (rad)** parameters. When all elements in **Constellation points per circle** are equal, and all elements in **Phase offset of each circle (rad)** are equal, the default is `Gray`. For all other cases, the default is `Contourwise-gray`.

`Custom symbol mapping` — Custom symbol mapping

`[0,4,12,8,1,3,2,6,7,5,13,15,14,10,11,9]` (default) | integer vector

Custom symbol mapping, specified as an integer vector. This vector must consist of *M* unique elements with values in the range [0, (*M* – 1)], where *M* is the “Modulation Order for M-APSK” on page 5-534. The first element in **Custom symbol mapping** corresponds to the constellation point in the first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

Example: The default value, `[0,4,12,8,1,3,2,6,7,5,13,15,14,10,11,9]`, specifies contourwise-gray symbol mapping. The distribution of constellation points is nonuniform on all contours.

Dependencies

To enable this parameter, set Symbol mapping to `User-defined`.

`Output type` — Output type

`Integer` (default) | `Bit`

Output type, specified as `Integer` or `Bit`.

Data Types: `char` | `string`

`Decision type` — Demodulation decision type

`Hard decision` (default) | `Log-likelihood ratio` | `Approximate log-likelihood ratio`

Demodulation decision type, specified as `Hard decision`, `Log-likelihood ratio`, or `Approximate log-likelihood ratio`. See “APSK Soft Demodulation” on page 5-535 for algorithm selection considerations.

Dependencies

This parameter applies only when Output type is set to `Bit`.

`Noise variance source` — Noise variance source

`Property` (default) | `Input port`

Noise variance source, specified as:

- `Property` — The noise variance is set using the Noise variance parameter.
- `Input port` — The noise variance is set using the Var input port.

Dependencies

This parameter applies only when Decision type is set to either `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

`Noise variance` — Noise variance

`1` (default) | positive scalar | vector of positive values

Noise variance, specified as a positive scalar or vector of positive values.

- When specified as a scalar, that value is used on all elements in the input signal.
- When specified as a vector, the vector length must be equal to the number of columns in the input signal. Each noise variance vector element is applied to its corresponding column in the input signal.

When the noise variance or signal power result in computations involving extreme positive or negative magnitudes, see “APSK Soft Demodulation” on page 5-535 for Decision type specification considerations.

Dependencies

This parameter applies only when Noise variance source is set to `Property` and Decision type is set to either `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

Data Types: `double`

`Output data type` — Output data type

`double` (default) | `...`

Output data type, specified as one of the acceptable values from this table. Acceptable **Output data type** values depend on the Output type and Decision type parameter values.

Output type	Decision type	Output data type Options
`Integer`	Not applicable	`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`
`Bit`	`Hard decision`	`double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `logical`
	`Log-likelihood ratio` or `Approximate log-likelihood ratio`	The output signal is the same data type as the input signal.

Dependencies

This parameter applies only when Output type is set to `Integer` or when Output type is set to `Bit` and Decision type is set to `Hard decision`.

`Simulate using` — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

Type of simulation to run, specified as:

- `Code generation` -– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` -– Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

For information on execution speed, see “Tips” on page 5-536.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`

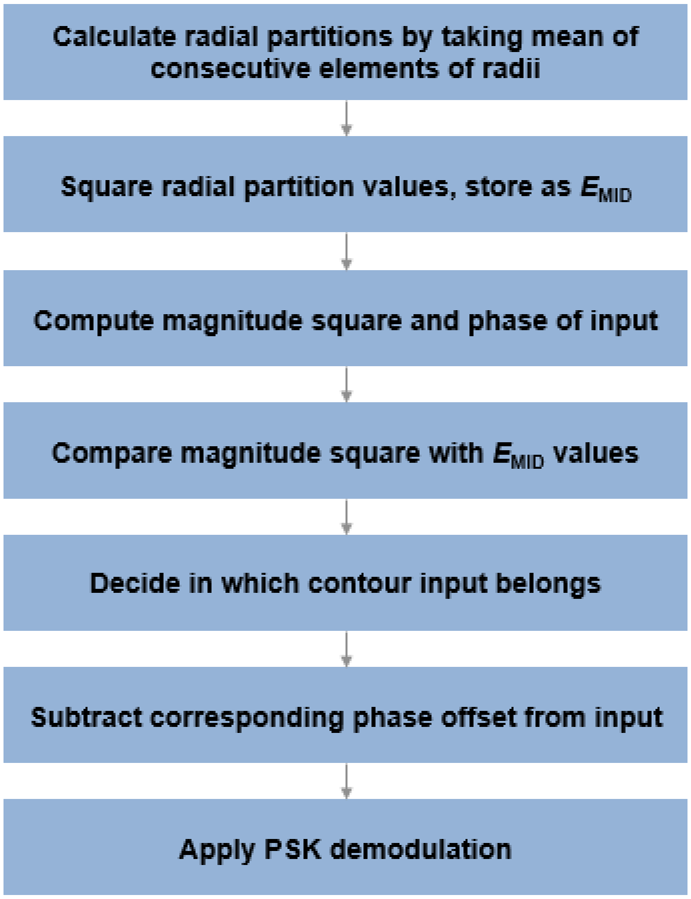
More About

Modulation Order for *M*-APSK

The modulation order, *M*, for *M*-APSK equals the sum of the vector elements in the Constellation points per circle parameter and is the total number of points in the signal constellation. Element values in **Constellation points per circle** must be multiples of four, and *M* must be a power of two.

APSK Hard Demodulation

The hard demodulation algorithm applies amplitude phase decoding, as described in [1].


Calculate radial partitions by taking mean of consecutive elements of radii
Square radial partition values, store as E_{MID}
Compute magnitude square and phase of input
Compare magnitude square with E_{MID} values
Decide in which contour input belongs
Subtract corresponding phase offset from input
Apply PSK demodulation

APSK Soft Demodulation

For soft demodulation, two soft-decision log-likelihood ratio (LLR) algorithms are available: exact LLR and approximate LLR. This table compares these algorithms.

Algorithm	Accuracy	Execution Speed
Exact LLR	more accurate	slower execution
Approximate LLR	less accurate	faster execution

For further description of these algorithms, see “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

Tips

- For faster execution of the M-APSK Demodulator Baseband block, set the Simulate using parameter to:
  - `Code generation` when using hard decision demodulation.
  - `Interpreted execution` when using soft decision demodulation.

References

[1] Sebesta, J. “Efficient Method for APSK Demodulation.” *Selected Topics on Applied Mathematics, Circuits, Systems, and Signals* (P. Pardalos, N. Mastorakis, V. Mladenov, and Z. Bojkovic, eds.). Vouliagmeni, Athens, Greece: WSEAS Press, 2009.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DVBS-APSK Demodulator Baseband | M-APSK Modulator Baseband | M-PSK Demodulator Baseband | MIL188-QAM Demodulator Baseband

Functions

`apskdemod`

Topics

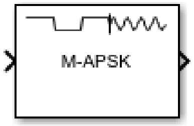
“Exact LLR Algorithm”
“Approximate LLR Algorithm”

Introduced in R2018b

M-APSK Modulator Baseband

M-ary amplitude phase shift keying (APSK) modulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / APM



Description

The M-APSK Modulator Baseband block modulates the input signal using *M*-ary amplitude phase shift keying (APSK) modulation. The output is a baseband representation of the modulated signal. *M*, the “Modulation Order for M-APSK” on page 5-540, equals the sum of the elements in Constellation points per circle. For a description of *M*-APSK modulation, see “Algorithms” on page 5-540.

Note M-APSK Modulator Baseband specifically applies to multiple ring PSK constellations. For a single ring PSK constellation, use M-PSK Modulator Baseband.

Ports

Input

`In` — Input signal
scalar | vector | matrix

Input signal, specified as a scalar, vector, or matrix. The input signal must consist of binary values or integers in the range [0, (*M* – 1)], where *M* is the “Modulation Order for M-APSK” on page 5-540. This port is unnamed on the block.

Note To process the input signal as binary elements, set the Input type parameter value to `Bit`. For binary inputs, the number of rows must be an integer multiple of $\log_2(M)$. Groups of $\log_2(M)$ bits in a column are mapped onto a symbol, with the first bit representing the MSB and the last bit representing the LSB.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Out` — APSK modulated signal
scalar | vector | matrix

APSK modulated signal, returned as a complex scalar, vector, or matrix. The output signal dimensions depend on the specified Input type value. This port is unnamed on the block.

Input type	Dimensions of Output Signal
`Integer`	The output signal has the same dimensions as the input signal.
`Bit`	The number of rows in the output signal equals the number of rows in the input signal divided by $\log_2(M)$, where M is the “Modulation Order for M-APSK” on page 5-540.

Use Output data type to specify the output data type.

Parameters

`Constellation points per circle` — Constellation points per PSK ring

`[4,12]` (default) | vector

Constellation points per PSK ring, specified as a vector with more than one element. Each vector element indicates the number of constellation points in its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. The sum of the elements in **Constellation points per circle** determines the modulation order. Element values must be multiples of four, and the modulation order must be a power of two.

Example: `[4,12,16]` specifies a three PSK ring constellation with a modulation order of 32.

`Radius of each circle` — Radius per PSK ring

`[0.5,1]` (default) | vector

Radius per PSK ring, specified as a vector with the same length as Constellation points per circle. Each vector element indicates the radius of its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. These element values must be positive and arranged in increasing order.

Example: `[0.5,1,2]` defines radii for three constellation PSK rings. The inner ring has a radius of 0.5, the second ring has a radius of 1.0, and the outer ring has a radius of 2.0.

`Phase offset of each circle (rad)` — Phase offset per PSK ring

`[pi/4,pi/12]` (default) | scalar | vector

Phase offset per PSK ring in radians, specified as a scalar or vector with the same length as Constellation points per circle. Each vector element indicates the phase offset of its corresponding PSK ring. The first element corresponds to the innermost circle, and so on until the last element, which corresponds to the outermost circle. The **Phase offset of each circle (rad)** can be a scalar only if all the elements of **Constellation points per circle** are the same value.

Example: `[pi/4,pi/12,pi/16]` defines phase offsets for three constellation PSK rings. The inner ring has a phase offset of π/4, the second ring has a phase offset of π/12, and the outer ring has a phase offset of π/16.

`Symbol mapping` — Symbol mapping

`Auto` (default) | `Contourwise-gray` | `Gray` | `User-defined`

Symbol mapping, specified as one of the following:

- `Contourwise-gray` — Uses Gray mapping along the contour in the phase dimension for each PSK ring.

- `Gray` — Uses Gray mapping along the contour in both the amplitude and phase dimensions. For Gray symbol mapping, all the values for Constellation points per circle must be equal and all the values for Phase offset of each circle (rad) must be equal. For a description of the Gray mapping used, see [2].
- `User-defined` — See Custom symbol mapping.

The default symbol mapping depends on **Constellation points per circle** and **Phase offset of each circle (rad)**. When all the elements of **Constellation points per circle** are equal and all the elements of **Phase offset of each circle (rad)** are equal, the default is `Gray`. For all other cases, the default is `Contourwise-gray`.

`Custom symbol mapping` — Custom symbol mapping

`[0,4,12,8,1,3,2,6,7,5,13,15,14,10,11,9]` (default) | integer vector

Custom symbol mapping, specified as an integer vector. This vector must consist of *M* unique elements with values in the range [0, (*M* – 1)], where *M* is the "Modulation Order for M-APSK" on page 5-540. The first element in **Custom symbol mapping** corresponds to the constellation point in the first quadrant of the innermost circle, with subsequent elements positioned counterclockwise around the PSK rings.

Example: The default value, `[0,4,12,8,1,3,2,6,7,5,13,15,14,10,11,9]`, specifies contourwise-gray symbol mapping. The distribution of constellation points is nonuniform on all contours.

Dependencies

To enable this parameter, set Symbol mapping to `User-defined`.

`Input type` — Input type

`Integer` (default) | `Bit`

Input type, specified as `Integer` or `Bit`. To use `Integer`, the input signal must consist of integers in the range [0, (*M* – 1)]. To use `Bit`, the input signal must contain binary values, and the number of rows must be an integer multiple of $\log_2(M)$, where *M* is the "Modulation Order for M-APSK" on page 5-540.

`Output data type` — Output data type

`double` (default) | `single`

Output data type, specified as `double` or `single`.

`View Constellation` — Plot reference constellation

`button`

To plot the reference constellation, click the **View Constellation** button.

Tip Click **Apply** before clicking the **View Constellation** to view latest parameter values.

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as:

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`no`

More About

Modulation Order for *M*-APSK

The modulation order, *M*, for *M*-APSK is the total number of points in the signal constellation. *M* equals the sum of the elements in Constellation points per circle. The element values in **Constellation points per circle** must be multiples of four. *M* must be a power of two.

Algorithms

The block implements a pure APSK constellation.

A pure M-APSK constellation is composed of N_C concentric rings or contours, each with uniformly spaced PSK points. The M-APSK constellation set is

$$\chi = \begin{cases} R_1 \exp\left(j\left(\frac{2\pi}{M_1} i + \theta_1\right)\right), & i = 0, \ldots, M_1 - 1, \\ R_2 \exp\left(j\left(\frac{2\pi}{M_2} i + \theta_2\right)\right), & i = 0, \ldots, M_2 - 1, \\ \quad \vdots & \quad \vdots \\ R_{N_C} \exp\left(j\left(\frac{2\pi}{M_{N_C}} i + \theta_{\mathrm{Nc}}\right)\right), & i = 0, \ldots, M_{N_C} - 1, \end{cases}$$

where

- The modulation order is equal to the sum of all M_l for $l = 1, 2, \ldots, N_C$.
- N_C is the number of concentric rings. $N_C \geq 2$.
- M_l is the number of constellation points in the *l*th ring.
- R_l is the radius of the *l*th ring.
- θ_l is the phase offset of the *l*th ring.
- $j = \sqrt{-1}$

References

[1] Corazza, Giovanni E. *Digital Satellite Communications*. New York: Springer Science Business Media, LLC, 2007.

[2] Liu, Z., Q. Xie, K. Peng, and Z. Yang. "APSK Constellation with Gray Mapping." *IEEE Communications Letters*. Vol. 15, Number 12, December 2011, pp. 1271–1273.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DVBS-APSK Modulator Baseband | M-APSK Demodulator Baseband | M-PSK Modulator Baseband | MIL188-QAM Modulator Baseband

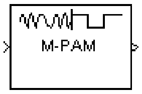
Functions

`apskmod`

Introduced in R2018b

M-PAM Demodulator Baseband

Demodulate PAM-modulated data



Library

AM, in Digital Baseband sublibrary of Modulation

Description

The M-PAM Demodulator Baseband block demodulates a signal that was modulated using M-ary pulse amplitude modulation. The input is a baseband representation of the modulated signal.

The signal constellation has M points, where M is the **M-ary number** parameter. M must be an even integer. The block scales the signal constellation based on how you set the **Normalization method** parameter. For details on the constellation and its scaling, see the reference page for the M-PAM Modulator Baseband block.

This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-547.

Note All values of power assume a nominal impedance of 1 ohm.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Output type** parameter to `Integer`, the block outputs integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Output type** parameter to `Bit`, the block outputs binary-valued signals that represent integers. The block represents each integer using a group of $K = \log_2(M)$ bits, where K represents the number of bits per symbol. The output vector length must be an integer multiple of K.

The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation. More details are on the reference page for the M-PAM Modulator Baseband block.

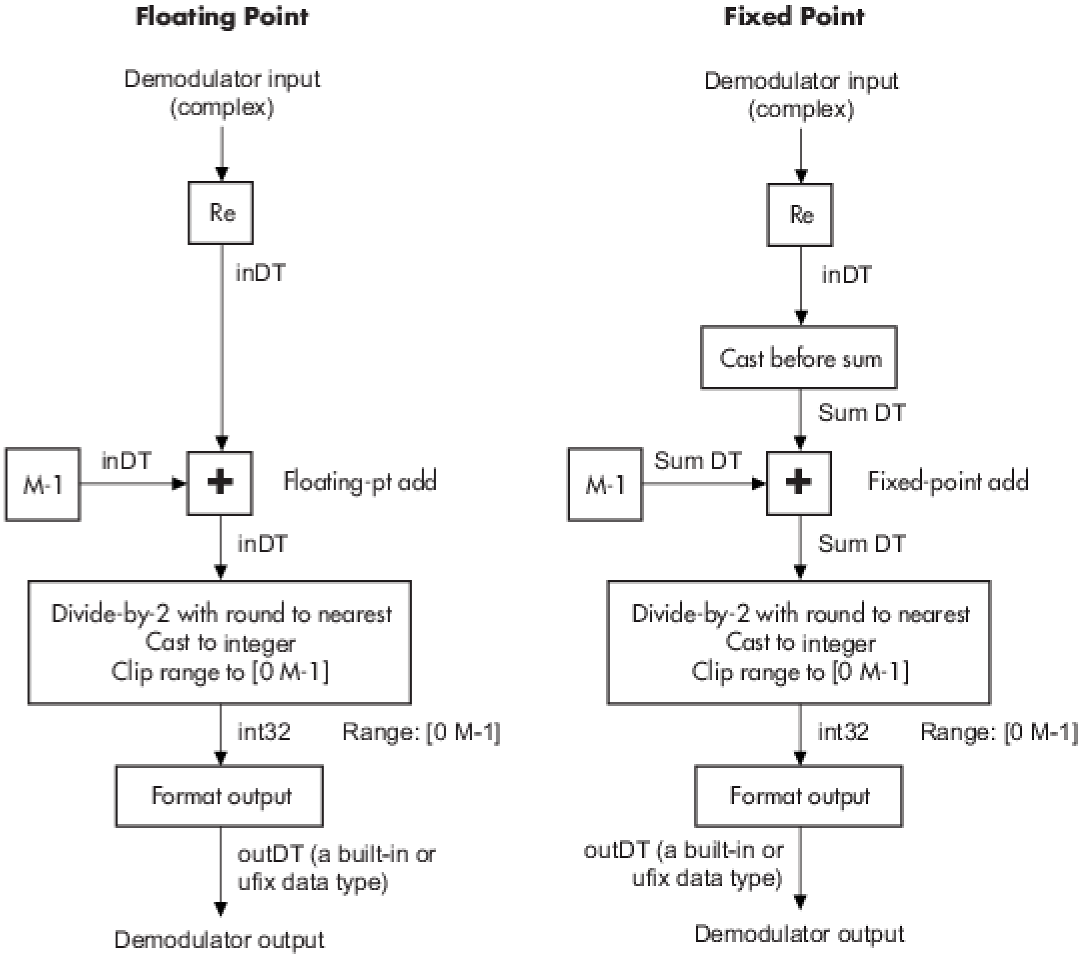
Algorithm

The demodulator algorithm maps received input signal constellation values to M-ary integer symbol indices between 0 and M-1 and then maps these demodulated symbol indices to formatted output values.

The integer symbol index computation is performed by first scaling the real part of the input signal constellation (possibly with noise) by a denormalization factor derived from the **Normalization**

method and related parameters. This denormalized value is added to M-1 to translate it into an approximate range between 0 and 2 x (M-1) plus noise. The resulting value is then rescaled via a divide-by-two (or, equivalently, a right-shift by one bit for fixed-point operation) to obtain a range approximately between 0 and M-1 (plus noise). The noisy index value is rounded to the nearest integer and clipped, via saturation, to the exact range of [0 M-1]. Finally, based on other block parameters, the integer index is mapped to a symbol value that is formatted and cast to the selected **Output data type**.

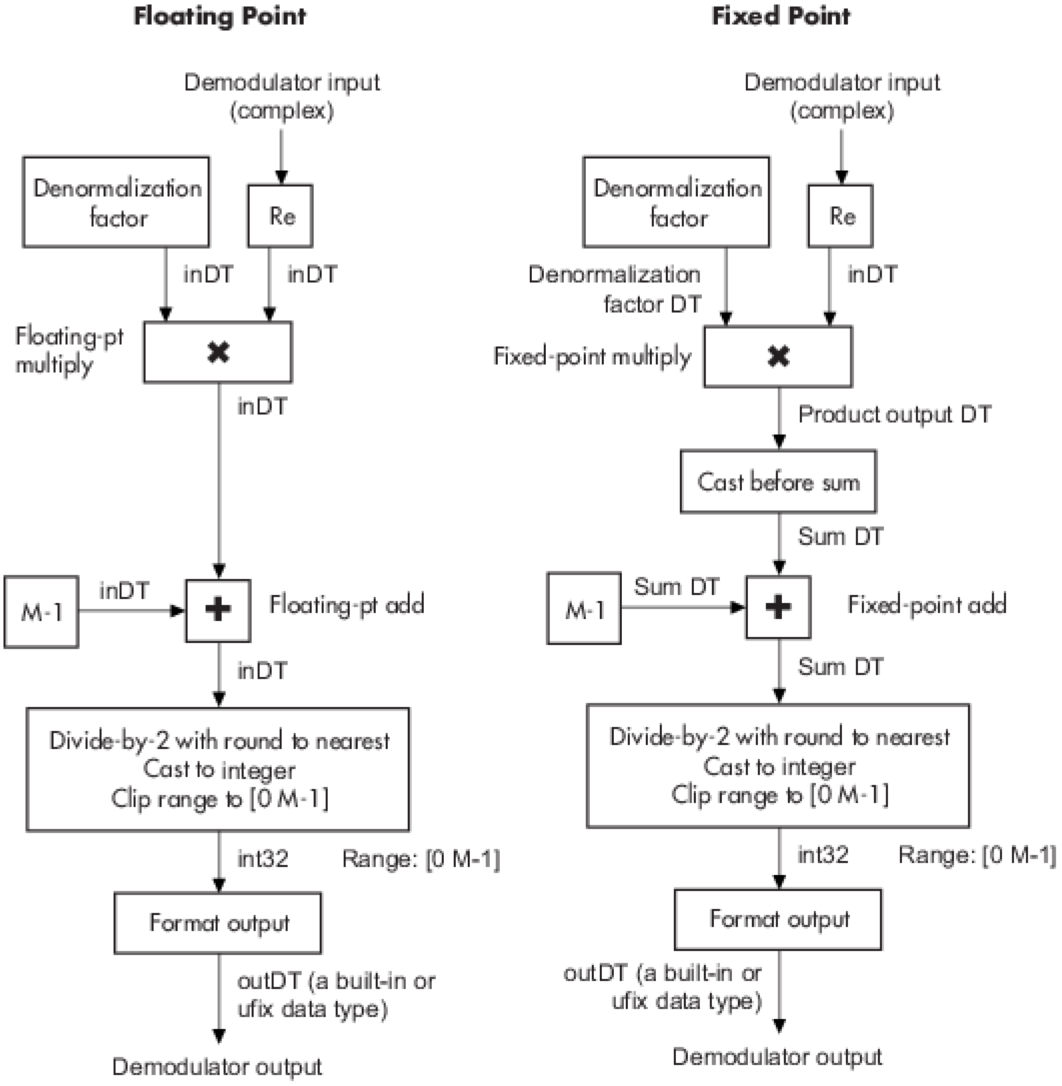
The following figures contains signal flow diagrams for floating-point and fixed-point algorithm operation. The floating-point diagrams apply when the input signal data type is `double` or `single`. The fixed-point diagrams apply when the input signal is a signed fixed-point data type. Note that the diagram is simplified when using normalized constellations (i.e., denormalization factor is 1).



Floating Point

Fixed Point

Signal-Flow Diagrams with Denormalization Factor Equal to 1



Signal-Flow Diagrams with Nonunity Denormalization Factor

Parameters

M-ary number

The number of points in the signal constellation. It must be an even integer.

Output type

Determines whether the output consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Constellation ordering

Determines how the block maps each integer to a group of output bits.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

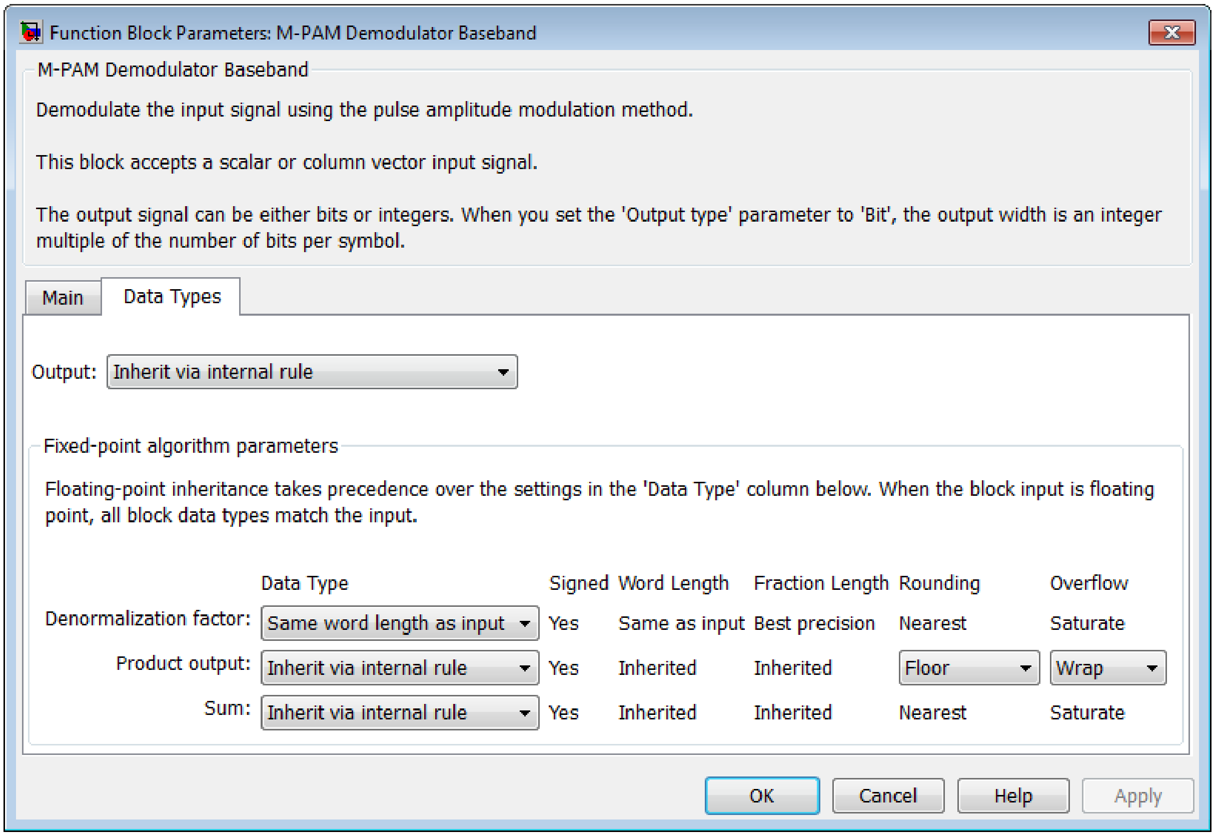
The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.



M-PAM Demodulator Baseband

Demodulate the input signal using the pulse amplitude modulation method.

This block accepts a scalar or column vector input signal.

The output signal can be either bits or integers. When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol.

Output: Inherit via internal rule

Fixed-point algorithm parameters

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Denormalization factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Product output:	Inherit via internal rule	Yes	Inherited	Inherited	Floor	Wrap
Sum:	Inherit via internal rule	Yes	Inherited	Inherited	Nearest	Saturate

Output

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`. Otherwise, the output data type will be as if this parameter is set to `'Smallest unsigned integer'`.

When the parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., `ufix(1)` for bit outputs, and `ufix(ceil(log2(M)))` for integer outputs. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., `uint8`).

For integer outputs, this parameter can be set to `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, the options are `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

Denormalization factor

This parameter applies when a fixed-point input is not normalized. It can be set to `Same word length as input` or `Specify word length`, in which case a field is enabled for user input. A best-precision fraction length is always used.

Product output

This parameter only applies when the input is a fixed-point signal and there is a nonunity (not equal to 1) denormalized factor. It can be set to `Inherit via internal rule` or `Specify word length`, which enables a field for user input.

Setting to `Inherit via internal rule` computes the full-precision product word length and fraction length. Internal Rule for Product Data Types in *DSP System Toolbox User's Guide* describes the full-precision Product output internal rule.

Setting to `Specify word length` allows you to define the word length. The block computes a best-precision fraction length based on the word length specified and the pre-computed worst-case (min/max) real world value **Product output** result. The worst-case **Product output** result is precomputed by multiplying the denormalized factor with the worst-case (min/max) input signal range, purely based on the input signal data type.

The block uses the **Rounding** method when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. For more information, see “Rounding Modes” or “Rounding Mode: Simplest” (Fixed-Point Designer).

Sum

This parameter only applies when the input is a fixed-point signal. It can be set to `Inherit via internal rule`, `Same as product output`, or `Specify word length`, in which case a field is enabled for user input

Setting `Inherit via internal rule` computes the full-precision sum word length and fraction length, based on the two inputs to the Sum in the fixed-point Hard Decision Algorithm on page 5-542 signal flow diagram. The rule is the same as the fixed-point inherit rule of the internal **Accumulator data type** parameter in the Simulink Sum (Simulink) block.

Setting `Specify word length` allows you to define the word length. A best precision fraction length is computed based on the word length specified in the pre-computed maximum range necessary for the demodulated algorithm to produce accurate results. The signed fixed-point data type that has the best precision fully contains the values in the range 2 * (M-1) for the specified word length.

Setting to `Same as product output` allows the Sum data type to be the same as the **Product output** data type (when **Product output** is used). If the **Product output** is not used, then this setting will be ignored and the `Inherit via internal rule` Sum setting will be used.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • $ufix(\lceil \log_2 M \rceil)$ in ASIC/FPGA when **Output type** is `Integer`

Pair Block

M-PAM Modulator Baseband

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

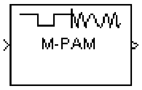
See Also

Blocks
General QAM Demodulator Baseband | M-PAM Modulator Baseband

Introduced before R2006a

M-PAM Modulator Baseband

Modulate using M-ary pulse amplitude modulation



Library

AM, in Digital Baseband sublibrary of Modulation

Description

The M-PAM Modulator Baseband block modulates using M-ary pulse amplitude modulation. The output is a baseband representation of the modulated signal. The **M-ary number** parameter, M, is the number of points in the signal constellation. It must be an even integer.

Note All values of power assume a nominal impedance of 1 ohm.

Constellation Size and Scaling

Baseband M-ary pulse amplitude modulation using the block's default signal constellation maps an integer m between 0 and M-1 to the complex value

2m - M + 1

Note This value is actually a real number. The block's output signal is a complex data-type signal whose imaginary part is zero.

The block scales the default signal constellation based on how you set the **Normalization method** parameter. The following table lists the possible scaling conditions.

Value of Normalization Method Parameter	Scaling Condition
`Min. distance between symbols`	The nearest pair of points in the constellation is separated by the value of the **Minimum distance** parameter
`Average Power`	The average power of the symbols in the constellation is the **Average power** parameter
`Peak Power`	The maximum power of the symbols in the constellation is the **Peak power** parameter

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar or column vector input signal.

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of *K* bits.

The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation.

- If **Constellation ordering** is set to `Binary`, then the block uses a natural binary-coded constellation.
- If **Constellation ordering** is set to `Gray`, then the block uses a Gray-coded constellation.

  For details about the Gray coding, see the reference page for the M-PSK Modulator Baseband block.

Constellation Visualization

The M-PAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications Toolbox User's Guide*.

Parameters

M-ary number

The number of points in the signal constellation. It must be an even integer.

Input type

Indicates whether the input consists of integers or groups of bits. If this parameter is set to `Bit`, then the **M-ary number** parameter must be 2^K for some positive integer K.

Constellation ordering

Determines how the block maps each group of input bits to a corresponding integer.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Output data type

The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Output fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter or when you select `User-defined` and the specified output data type is a fixed-point data type.

Output fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, 32-bit signed integers • 8-, 16-, 32-bit unsigned integers • $ufix(\lceil\log_2 M\rceil)$ when **Input type** is `Integer`

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

M-PAM Demodulator Baseband

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

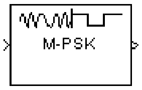
Blocks
General QAM Modulator Baseband | M-PAM Demodulator Baseband

Introduced before R2006a

M-PSK Demodulator Baseband

Demodulate PSK-modulated data

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM
Communications Toolbox HDL Support / Modulation / PM



Description

The M-PSK Demodulator Baseband block demodulates a baseband representation of a PSK-modulated signal. The modulation order, *M*, is equivalent to the number of points in the signal constellation and is determined by the **M-ary number** parameter. The block accepts scalar or column vector input signals.

Input/Output Ports

Input

`Port_1` — Input signal
scalar | vector

Input port accepting a baseband representation of a PSK-modulated signal.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Output signal
scalar | vector

Output signal, returned as a scalar or vector. The output is a demodulated version of the PSK-modulated signal.

Data Types: `single` | `double` | `fixed point`

Parameters

`M-ary number` — Modulation order of the PSK constellation
8 (default) | scalar

Specify the modulation order as a positive integer power of two.

Example: 2 | 16

`Output type` — Output signal data type
`Integer` (default) | `Bit`

Specify the elements of the input signal as integers or bits. If **Output type** is `Bit`, the number of samples per frame is an integer multiple of the number of bits per symbol, $\log_2(M)$.

`Decision type` — Demodulator output

`Hard decision` (default) | `Log-likelihood ratio` | `Approximate log-likelihood ratio`

Specify the demodulator output to be hard decision, log-likelihood ratio (LLR), or approximate LLR. The LLR and approximate LLR outputs are used with error decoders that support soft-decision inputs such as a Viterbi decoder, to achieve superior performance. This parameter is available when **Output type** is `Bit`.

See “Phase Modulation” for algorithm details. The output values for `Log-likelihood ratio` and `Approximate log-likelihood ratio` decision types are of the same data type as the input values

`Noise variance source` — Source of noise variance

`Dialog` (default) | `Port`

Specify the source of the noise variance estimate. This parameter is available when **Decision type** is `Log-likelihood ratio` or `Approximate log-likelihood ratio`.

- To specify the noise variance from the dialog box, select `Dialog`.
- To input the noise variance from an input port, select `Port`.

`Noise variance` — Estimate of noise variance

`1` (default) | positive scalar

Specify the estimate of the noise variance as a positive scalar. This parameter is available when **Noise variance source** is `Dialog`.

This parameter is tunable in all simulation modes. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. Avoiding recompilation is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

Note The exact LLR algorithm computes exponentials using finite precision arithmetic. Computation of exponentials with very large positive or negative magnitudes might yield:

- `Inf` or `-Inf` if the noise variance is a very large value
- `NaN` if both the noise variance and signal power are very small values

When the output returns any of these values, try using the approximate LLR algorithm because it does not compute exponentials.

`Constellation ordering` — Symbol mapping

`Gray` (default) | `Binary` | `User-defined`

Specify how the integer or group of $\log_2(M)$ bits is mapped to the corresponding symbol.

- When **Constellation ordering** is set to `Gray`, the output symbol is mapped to the input signal using a Gray-encoded signal constellation.

- When **Constellation ordering** is set to `Binary`, the modulated symbol is exp($j\phi+j2\pi m/M$), where ϕ is the phase offset in radians, m is the integer output such that $0 \leq m \leq M - 1$, and M is the modulation order.
- When **Constellation ordering** is `User-defined`, specify a vector of size M, which has unique integer values in the range [0, M–1]. The first element of this vector corresponds to the constellation point having a value of $e^{j\phi}$ with subsequent elements running counterclockwise.

Example: `[0 3 2 1]`

`Constellation mapping` — User-defined symbol mapping

`[0:7]` (default) | vector

Specify the order in which input integers are mapped to output integers. The parameter is available when **Constellation ordering** is `User-defined`, and must be a row or column vector of size M having unique integer values in the range [0, M – 1].

The first element of this vector corresponds to the constellation point at 0 + **Phase offset** angle, with subsequent elements running counterclockwise. The last element corresponds to the -2π/M + **Phase offset** constellation point.

`Phase offset (rad)` — Phase offset in radians

`pi/8` (default) | scalar

Specify, in radians, the phase offset of the initial constellation as a real scalar.

Example: `pi/4`

`Output data type` — Output data type

`Inherit via internal rule` (default) | `Smallest unsigned integer` | `double` | `single` | `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32`

Specify the data type of the demodulated output signal.

Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`[abc] | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

[a] M = 2, 4, 8 only.

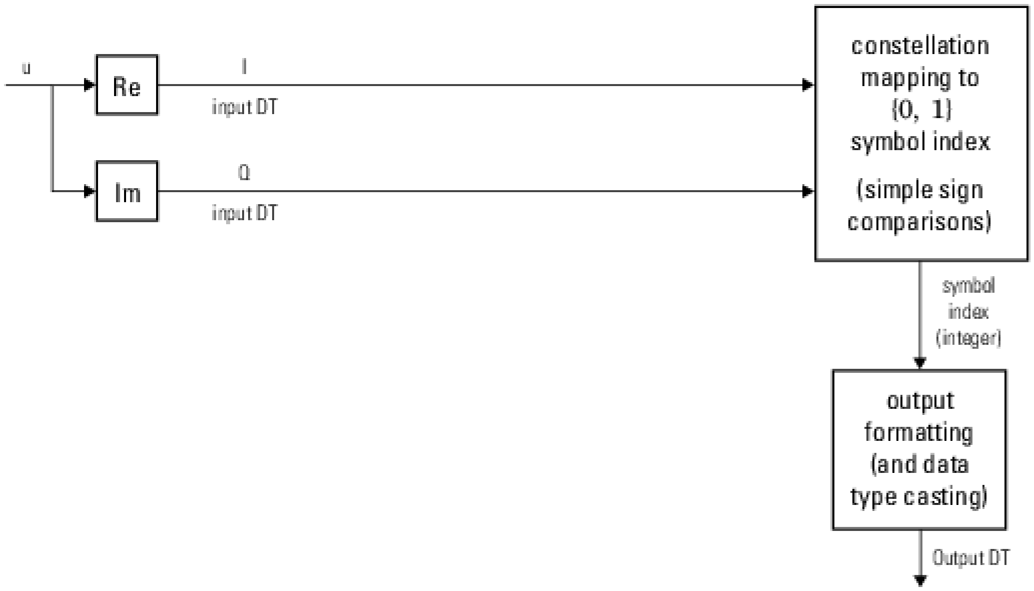
[b] Fixed-point inputs must be signed.

[c] When ASIC/FPGA is selected in the Hardware Implementation Pane, output is ufix(1) for bit outputs, and ufix(ceil(log2(M))) for integer outputs.

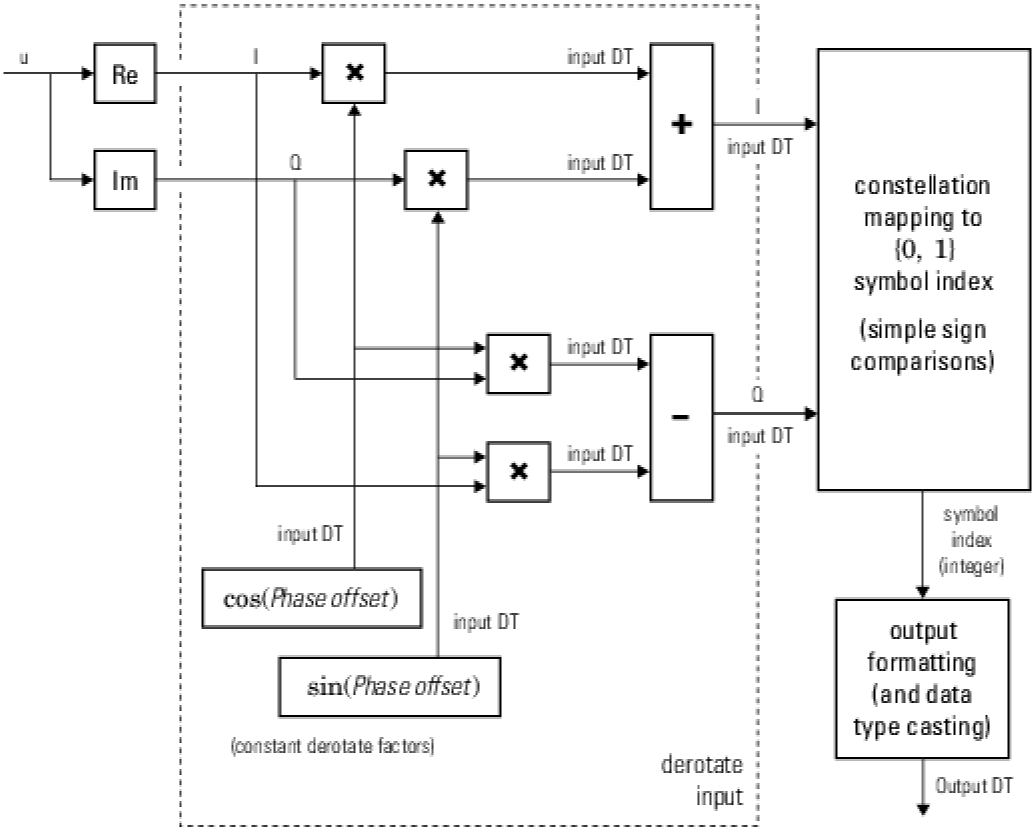
Algorithms

BPSK

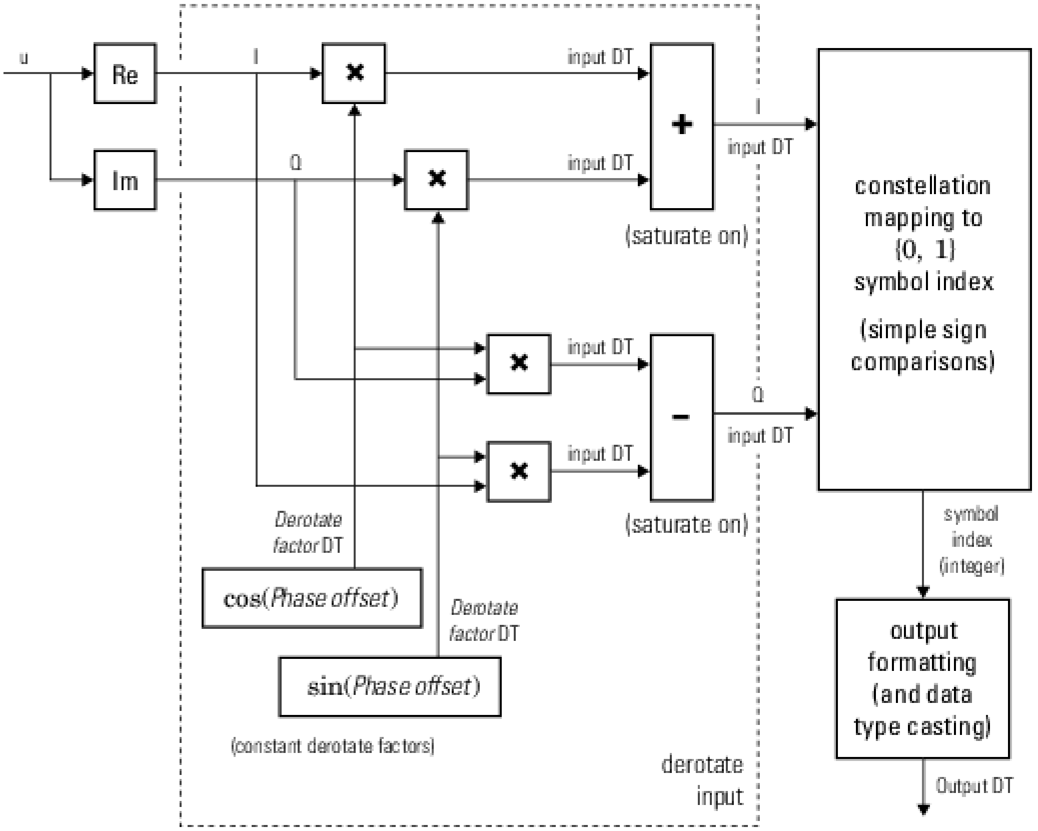
Diagrams for hard-decision demodulation of BPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1} symbol index (simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision BPSK Demodulator Signal Diagram for Trivial Phase Offset (multiple of π/2)



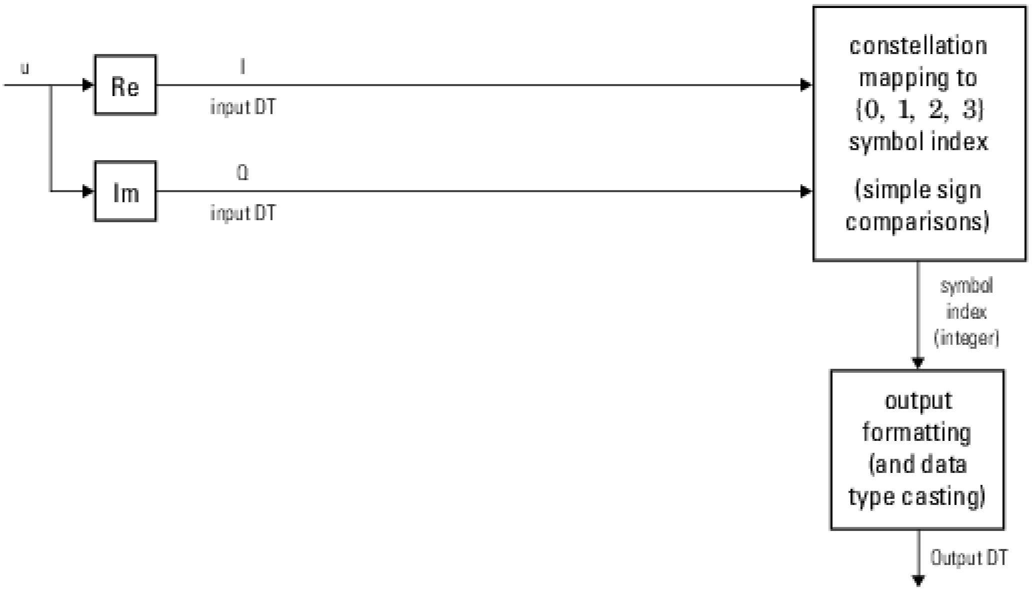
Hard-Decision BPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset



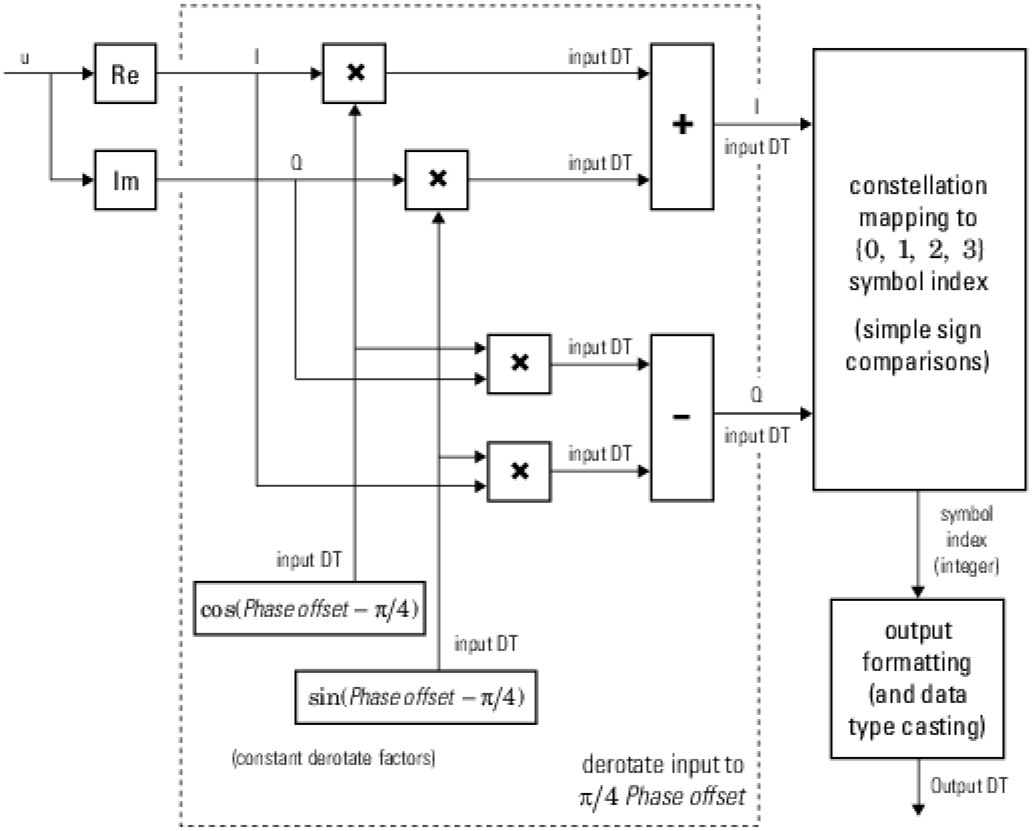
Hard-Decision BPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

QPSK

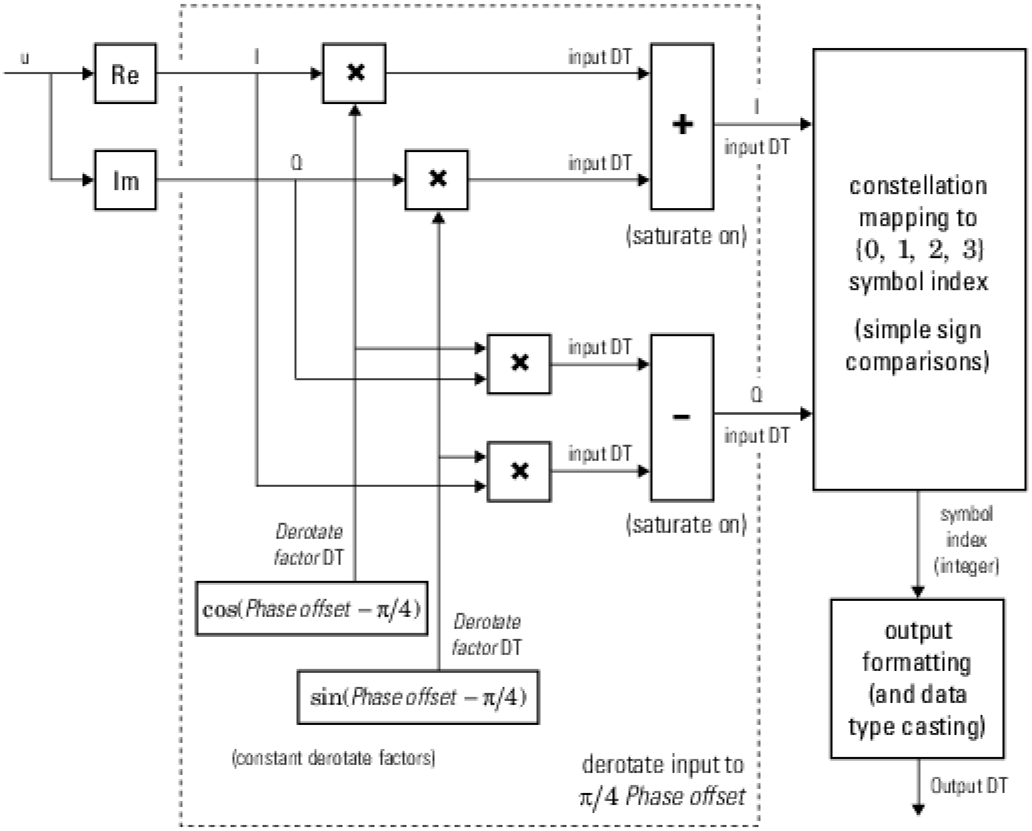
Diagrams for hard-decision demodulation of QPSK signals follow.


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of π/4)


u
Re
Im
I
Q
×
input DT
+
−
I
input DT
Q
input DT
cos(Phase offset − π/4)
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

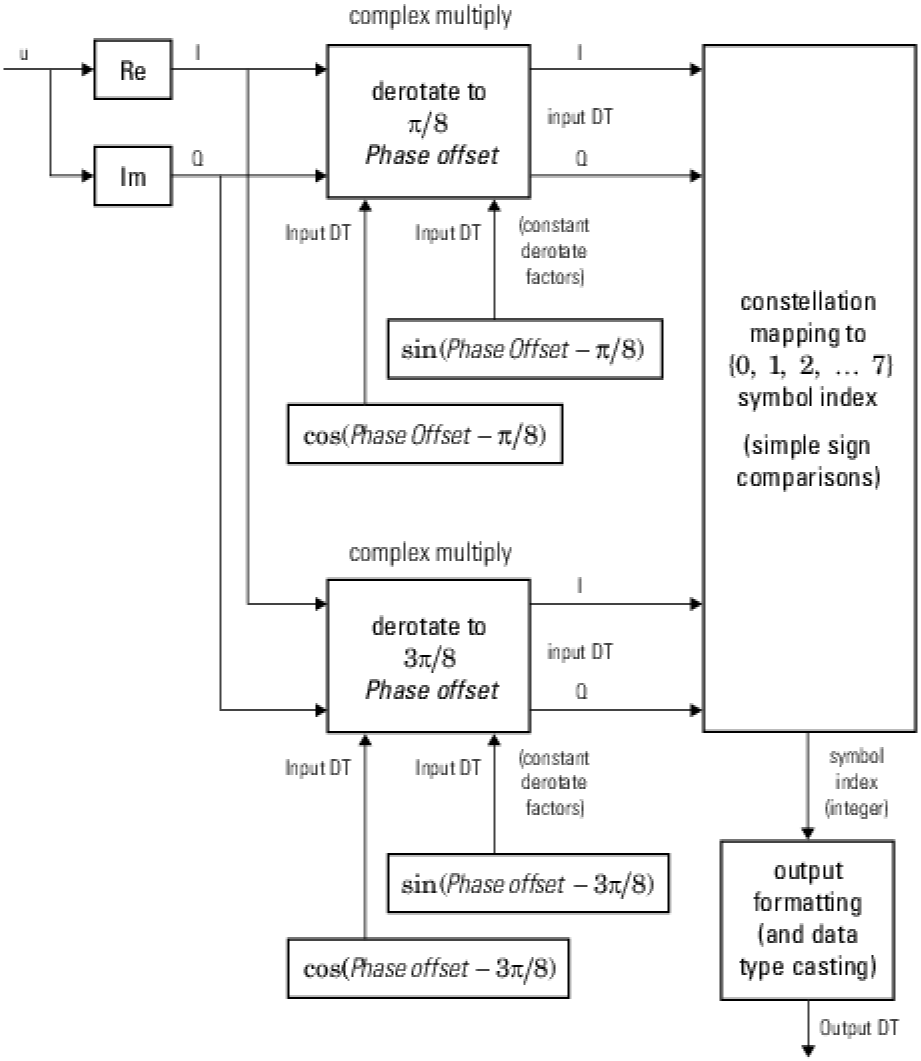
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset


u
Re
Im
I
Q
×
input DT
+
I
input DT
(saturate on)
–
Q
(saturate on)
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
Derotate factor DT
cos(Phase offset − π/4)
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset

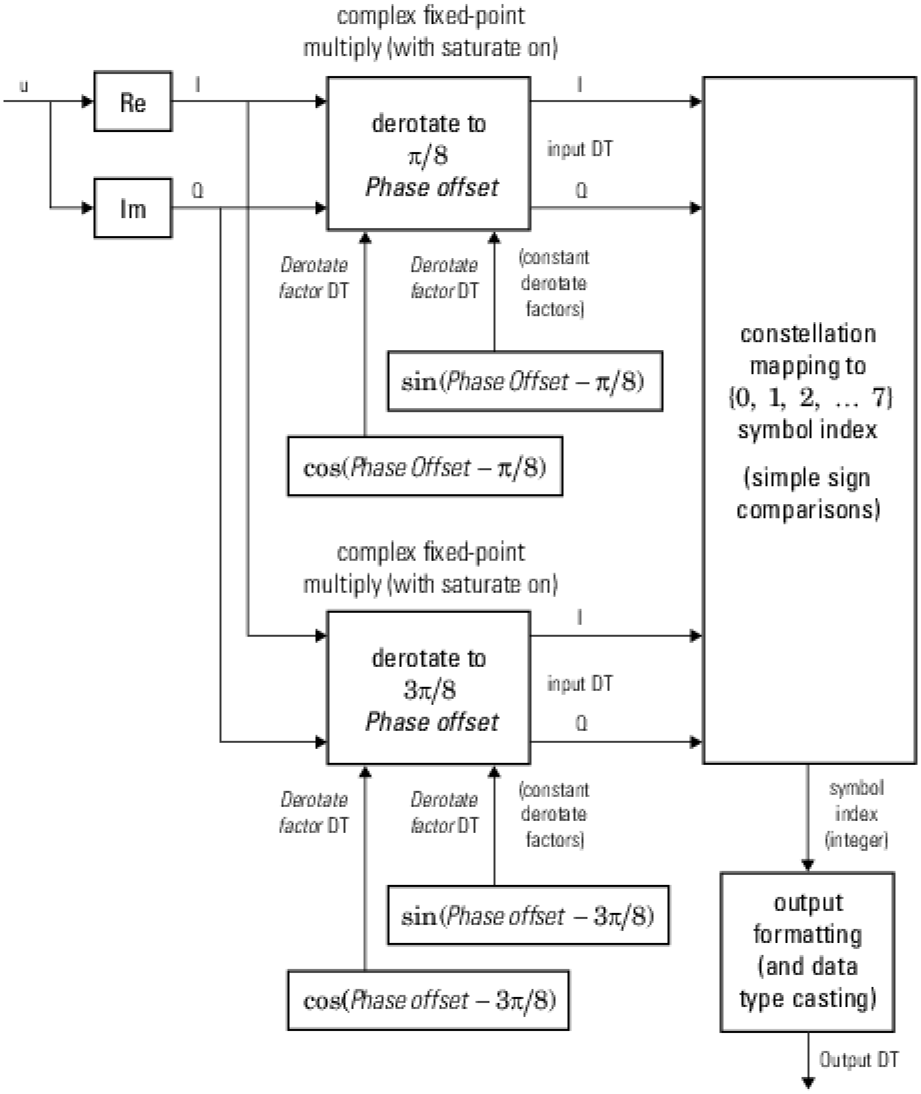
Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

Higher-Order PSK

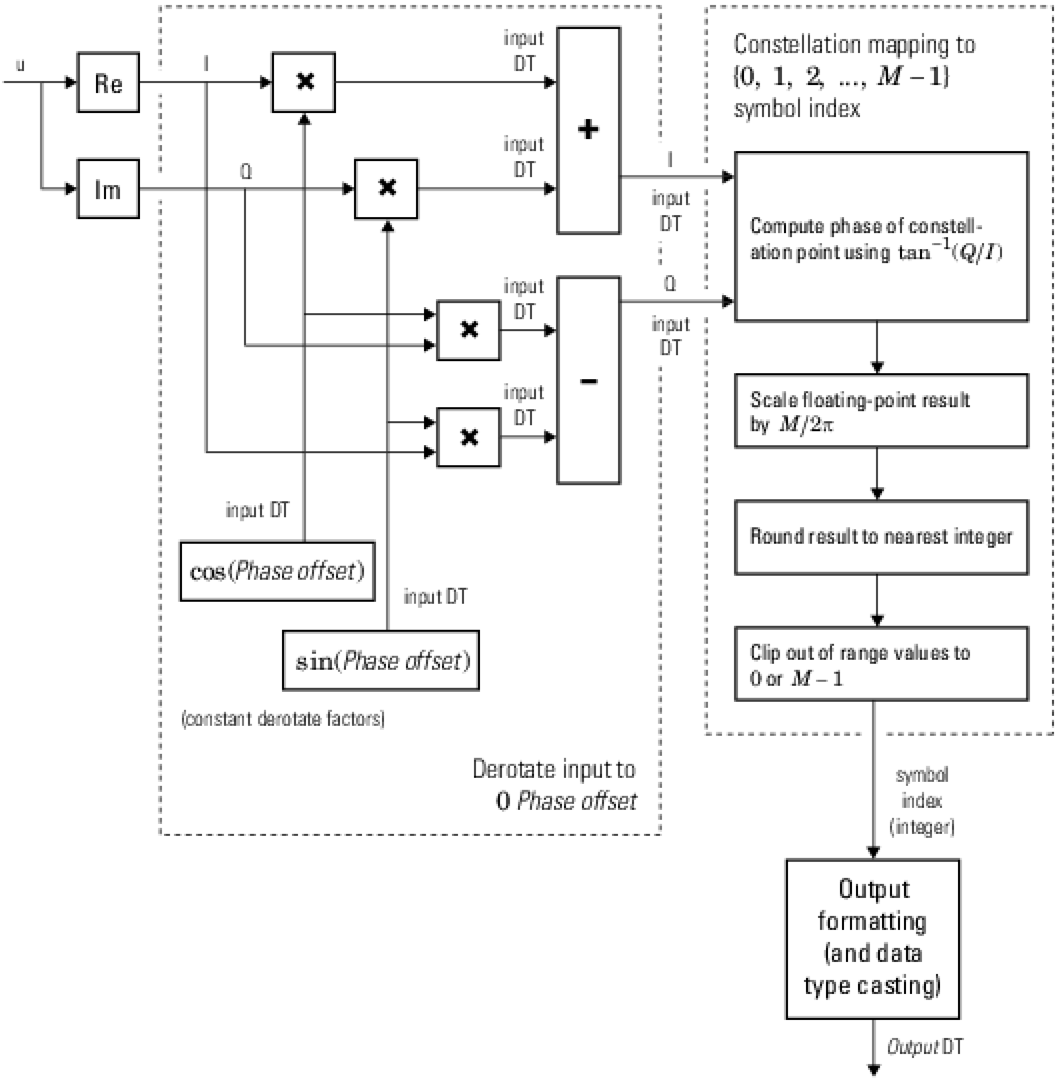
Diagrams for hard-decision demodulation of higher-order ($M \geq 8$) signals follow.



Hard-Decision 8-PSK Demodulator Floating-Point Signal Diagram



Hard-Decision 8-PSK Demodulator Fixed-Point Signal Diagram


u
Re
Im
I
Q
input DT
+
−
×
Constellation mapping to
{0, 1, 2, ..., M − 1}
symbol index
Compute phase of constellation point using $\tan^{-1}(Q/I)$
Scale floating-point result by $M/2\pi$
Round result to nearest integer
Clip out of range values to 0 or $M-1$
symbol index (integer)
Output formatting (and data type casting)
Output DT
cos(Phase offset)
sin(Phase offset)
(constant derotate factors)
Derotate input to 0 Phase offset

Hard-Decision M-PSK Demodulator (M > 8) Floating-Point Signal Diagram for Nontrivial Phase Offset

For $M > 8$, to improve speed and implementation costs, no derotation arithmetic is performed when **Phase offset** is 0, $\pi/2$, π, or $3\pi/2$ (that is, when it is trivial).

Also, for $M > 8$, this block only supports `double` and `single` input types.

Log-Likelihood Ratio and Approximate Log-Likelihood Ratio

The exact LLR and approximate LLR algorithms (soft-decision) are described in “Phase Modulation”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is 0. For more details, see "ConstrainedOutputPipeline" (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "InputPipeline" (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is 0. For more details, see "OutputPipeline" (HDL Coder).

See Also

M-DPSK Demodulator Baseband | M-PSK Modulator Baseband

Topics

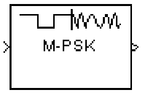
"Phase Modulation"
"Gray Coded 8-PSK"

Introduced before R2006a

M-PSK Modulator Baseband

Modulate using M-ary phase shift keying

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM
Communications Toolbox HDL Support / Modulation / PM



Description

The M-PSK Modulator Baseband block modulates an input signal using M-ary phase shift keying (PSK) and returns a complex baseband output. The modulation order, *M*, which is equivalent to the number of points in the signal constellation, is determined by the **M-ary number** parameter. The block accepts scalar or column vector input signals.

Input/Output Ports

Input

`Port_1` — Input signal
scalar | vector

Specify the input signal as an integer scalar, integer vector, or binary vector.

- When **Input type** is `Integer`, specify the input signal elements as integers from 0 to *M* – 1.
- When **Input type** is `Bit`, specify the input signal as a binary vector in which the number of elements is an integer multiple of the bits per symbol. The bits per symbol is equal to $\log_2(M)$.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16` | `uint32` | `Boolean`

Output

`Port_1` — Output signal
scalar | vector

Output signal, returned as a complex scalar or vector. The output is the complex baseband representation of the PSK-modulated signal.

Data Types: `single` | `double` | `fixed point`

Parameters

`M-ary number` — Modulation order of the PSK constellation
8 (default) | scalar

Specify the modulation order as a positive integer power of two.

Example: 2 | 16

Input type — Type of input signal

Integer (default) | Bit

Specify the elements of the input signal as integers or bits. If **Input type** is Bit, the number of samples per frame must be an integer multiple of the number of bits per symbol. The number of bits per symbol is $\log_2(M)$.

Constellation ordering — Symbol mapping

Gray (default) | Binary | User-defined

Specify how the integer or group of $\log_2(M)$ bits is mapped to the corresponding symbol.

- When **Constellation ordering** is set to Gray, the input signal is mapped to the output symbols using a Gray-encoded signal constellation.
- When **Constellation ordering** is set to Binary, the modulated symbol is $\exp(j\phi+j2\pi m/M)$, where ϕ is the phase offset in radians, m is the integer input such that $0 \le m \le M - 1$, and M is the modulation order.
- When **Constellation ordering** is User-defined, specify a vector of size M, which has unique integer values in the range $[0, M\text{–}1]$. The first element of this vector corresponds to the constellation point having an value of $e^{j\phi}$ with subsequent elements running counterclockwise.

Example: [0 3 2 1]

Constellation mapping — User-defined symbol mapping

[0:7] (default) | vector

Specify the order in which input integers are mapped to output integers. The parameter is available when **Constellation ordering** is User-defined, and must be a row or column vector of size M having unique integer values in the range $[0, M - 1]$.

The first element of this vector corresponds to the constellation point at 0 + **Phase offset** angle, with subsequent elements running counterclockwise. The last element corresponds to the -2π/M + **Phase offset** constellation point.

Phase offset (rad) — Phase offset in radians

pi/8 (default) | scalar

Specify, in radians, the phase offset of the initial constellation as a real scalar.

Example: pi/4

Output data type — Output data type

double (default) | single | Inherit via back propagation | fixdt(1,16) | fixdt(1,16,0) | <data type expression>

Specify the data type of the modulated output signal. Set this parameter to one of the fixed point options or <data type expression> to enable parameters in which you specify additional details. Set this parameter to Inherit via back propagation, to match the output data type and scaling to the following block in the model.

Block Characteristics

Data Types	Boolean | double | fixed point[ab] | integer | single

Multidimensional Signals	no
Variable-Size Signals	yes

a. ufix(1) at the input if ''input type'' is set to ''bit''. ufix(ceil(log2(M))) at input if ''input type'' is set to ''integer'' for M-ary modulation.

b. Fixed-point outputs must be signed.

Tips

The M-PSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. Clicking the **View Constellation** button allows you to visualize a signal constellation for the specified block parameters.

Algorithms

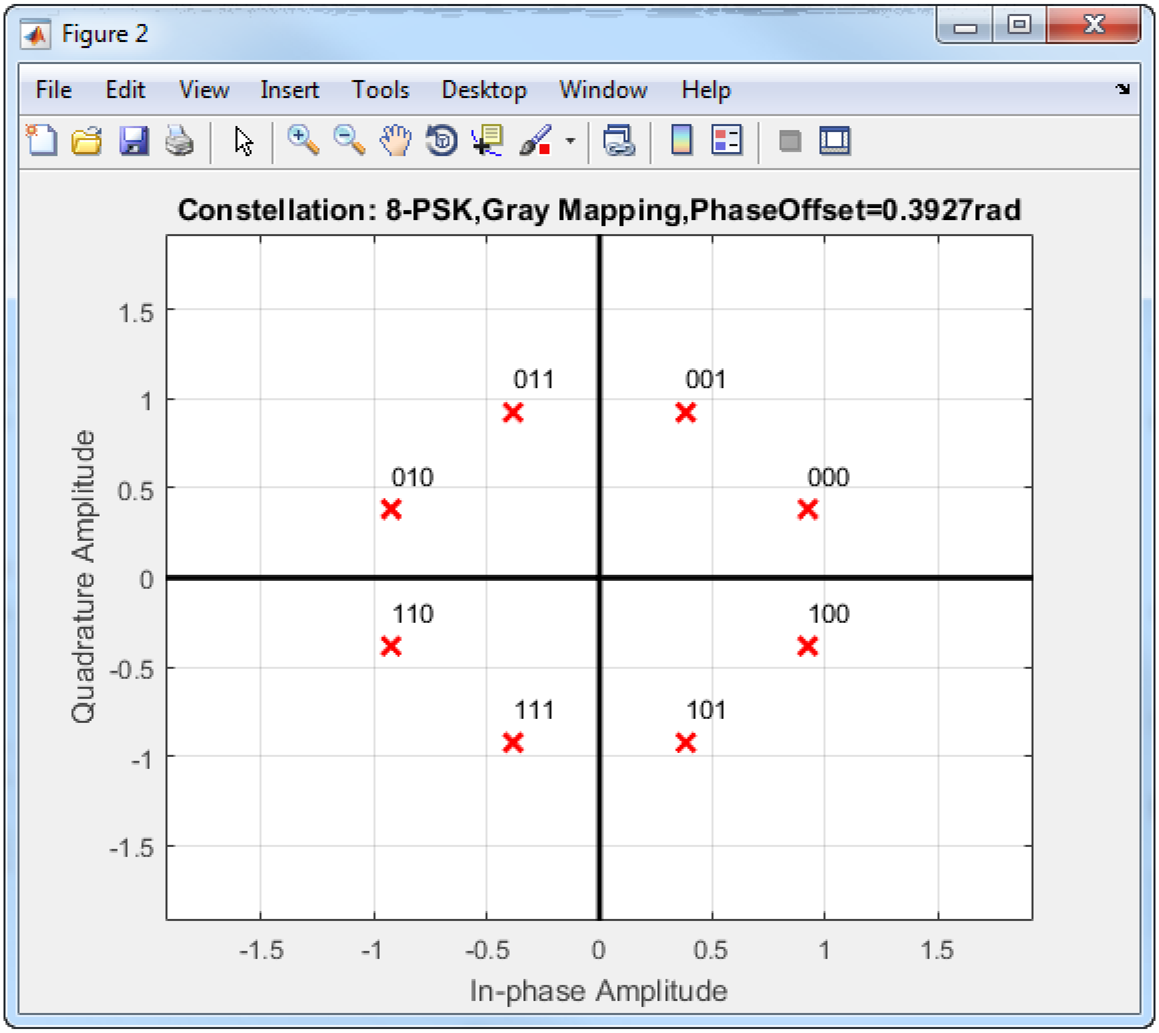
The block outputs a baseband signal by mapping input bits or integers to complex symbols according to the following:

$$s_n(t) = \exp\left(j\pi\left(\frac{2n+1}{M}\right)\right); \quad n \in \left\{0, 1, \ldots, M-1\right\}.$$

This applies when a natural binary ordering is used. Another common mapping is Gray coding, which has the advantage that only one bit changes between adjacent constellation points. This results in better bit error rate performance. For 8-PSK modulation with Gray coding, the mapping between the input and output symbols is shown.

Input	Output
0	0 *(000)*
1	1 *(001)*
2	3 *(011)*
3	2 *(010)*
4	6 *(110)*
5	7 *(111)*
6	5 *(101)*
7	4 *(100)*

The corresponding constellation diagram follows.



When the input signal is composed of bits, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $\log_2(M)$ bits.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

M-DPSK Modulator Baseband | M-PSK Demodulator Baseband

Topics

“Phase Modulation”
“Gray Coded 8-PSK”

Introduced before R2006a

M-PSK Phase Recovery

(Removed) Recover carrier phase using M-Power method

Note M-PSK Phase Recovery has been removed. Use the Carrier Synchronizer block instead.

Library

Carrier Phase Recovery sublibrary of Synchronization

Description

The M-PSK Phase Recovery block recovers the carrier phase of the input signal using the M-Power method. This feedforward, non-data-aided, clock-aided method is suitable for systems that use baseband phase shift keying (PSK) modulation. It is also suitable for systems that use baseband quadrature amplitude modulation (QAM), although the results are less accurate than those for comparable PSK systems. The alphabet size for the modulation must be an even integer.

For PSK signals, the **M-ary number** parameter represents the alphabet size. For QAM signals, the **M-ary number** should be 4 regardless of the alphabet size because the 4-power method is the most appropriate for QAM signals.

The M-Power method assumes that the carrier phase is constant over a series of consecutive symbols, and returns an estimate of the carrier phase for the series. The **Observation interval** parameter is the number of symbols for which the carrier phase is assumed constant. This number must be an integer multiple of the input signal's vector length.

Input and Output Signals

This block accepts a scalar or column vector input signal of type `double` or `single`. The input signal represents a baseband signal at the symbol rate, so it must be complex-valued and must contain one sample per symbol.

The outputs are as follows:

- The output port labeled `Sig` gives the result of rotating the input signal counterclockwise, where the amount of rotation equals the carrier phase estimate. The `Sig` output is thus a corrected version of the input signal, and has the same sample time and vector size as the input signal.
- The output port labeled `Ph` outputs the carrier phase estimate, in degrees, for all symbols in the observation interval. The `Ph` output is a scalar signal.

Note Because the block internally computes the argument of a complex number, the carrier phase estimate has an inherent ambiguity. The carrier phase estimate is between -180/M and 180/M degrees and might differ from the actual carrier phase by an integer multiple of 360/M degrees.

Delays and Latency

The block's algorithm requires it to collect symbols during a period of length **Observation interval** before computing a single estimate of the carrier phase. Therefore, each estimate is delayed by

Observation interval symbols and the corrected signal has a latency of **Observation interval** symbols, relative to the input signal.

Parameters

M-ary number

The number of points in the signal constellation of the transmitted PSK signal. This value as an even integer.

Observation interval

The number of symbols for which the carrier phase is assumed constant. The observation interval parameter must be an integer multiple of the input signal vector length.

When this parameter is exactly equal to the vector length of the input signal, then the block always works. When the integer multiple is not equal to `1`, on the **Simulation** tab, select **Model Settings**. Then in the **Solver > Solver selection** section, choose **Type:** `Fixed-step` and clear the **Treat each discrete rate as a separate task** checkbox.

Algorithm

If the symbols occurring during the observation interval are x(1), x(2), x(3),..., x(L), then the resulting carrier phase estimate is

$$\frac{1}{M}\arg\left\{\sum_{k=1}^{L}(x(k))^M\right\}$$

where the arg function returns values between -180 degrees and 180 degrees.

References

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

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

See Also

CPM Phase Recovery, M-PSK Modulator Baseband

Compatibility Considerations

M-PSK Phase Recovery has been removed

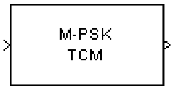
Errors starting in R2020a

M-PSK Phase Recovery has been removed. Use Carrier Synchronizer instead.

Introduced before R2006a

M-PSK TCM Decoder

Decode trellis-coded modulation data, modulated using PSK method



Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The M-PSK TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a PSK signal constellation.

The **M-ary number** parameter represents the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is the number of output bit streams from the convolutional encoder.)

The **Trellis structure** and **M-ary number** parameters in this block should match those in the M-PSK TCM Encoder block, to ensure proper decoding.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. The input signal must be `double` or `single`. The reset port signal must be `double` or `Boolean`. For information about the data types each block port supports, see “Supported Data Types” on page 5-571.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the M-PSK TCM Decoder block's output is a binary column vector whose length is *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

  If you select **Enable the reset input**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.
- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.

- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits, for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

M-ary number

The number of points in the signal constellation.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode

The operation mode of the Viterbi decoder. Choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port

When you check this box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous`.

Output data type

The output type of the block can be specified as a `boolean` or `double`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Reset	• Double-precision floating point • Boolean
Output	• Double-precision floating point • Boolean

Pair Block

M-PSK TCM Encoder

References

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

[2] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General TCM Decoder | M-PSK TCM Encoder

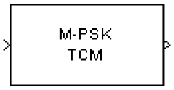
Functions

`poly2trellis`

Introduced before R2006a

M-PSK TCM Encoder

Convolutionally encode binary data and modulate using PSK method



Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The M-PSK TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a PSK signal constellation.

The **M-ary number** parameter is the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is equal to n for a rate k/n convolutional code.)

Input Signals and Output Signals

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the block input signal must be a binary column vector with a length of *L***k* for some positive integer *L*.

This block accepts a binary-valued input signal. The output signal is a complex column vector of length *L*.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code”. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink software to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

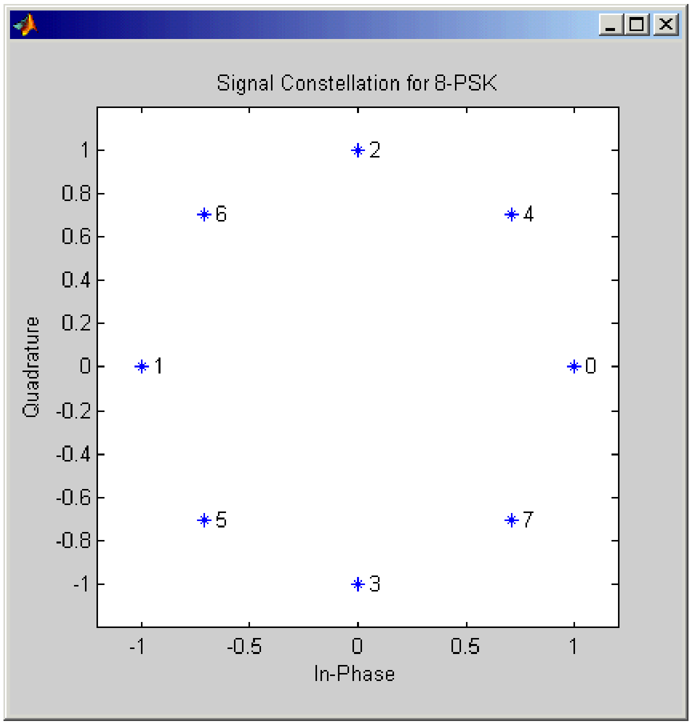
The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation`

mode to **Reset on nonzero input via port**. The block then opens a second input port, labeled Rst. The signal at the Rst port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

Signal Constellations

The trellis-coded modulation technique partitions the constellation into subsets called cosets, so as to maximize the minimum distance between pairs of points in each coset. This block internally forms a valid partition based on the value you choose for the **M-ary number** parameter.

The figure below shows the labeled set-partitioned signal constellation that the block uses when **M-ary number** is 8. For constellations of other sizes, see [1].



Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes [3].

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In Continuous mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In Truncated (reset every frame) mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by $y = n \cdot (x + s)/k$, where x is the number of input bits, and s = constraint length − 1 (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

M-ary number

The number of points in the signal constellation.

Output data type

The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Pair Block

M-PSK TCM Decoder

References

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

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General TCM Encoder | M-PSK TCM Decoder

Functions

`poly2trellis`

Introduced before R2006a

MSK Demodulator Baseband

Demodulate differentially encoded MSK-modulated data

Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The MSK Demodulator Baseband block demodulates a signal that was modulated using the differentially encoded minimum shift keying method. The block expects the input signal to be a baseband representation of a coherent modulated signal with no precoding. The **Phase offset** parameter represents the initial phase of the modulated waveform.

Pulse Shape Filtering

Differentially encoded minimum shift keying modulation uses pulse shaping to smooth the phase transitions of the modulated signal. The function $q(t)$ is the phase response obtained from the frequency pulse, $g(t)$, through this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to this rectangular pulse shape expression, $g(t)$.

Pulse Shape	Expression
`Rectangular`	$g(t) = \begin{cases} \frac{1}{2LT}, & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$

- L_{main} is the main lobe pulse duration in symbol intervals.
- The duration of the pulse, LT, is the pulse length in symbol intervals.

Integer-Valued Signals and Binary-Valued Signals

This block accepts a scalar-valued or column vector input signal with a data type of `single` or `double`. If you set the **Output type** parameter to `Integer`, then the block produces values of 1 and -1. If you set the **Output type** parameter to `Bit`, then the block produces values of 0 and 1.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. The input width must be an integer multiple of the **Samples per symbol** parameter value, and the input can be a column vector.

- When you set **Output type** to `Bit`, the output width is *K* times the number of input symbols.
- When you set **Output type** to `Integer`, the output width is the number of input symbols.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. The input must be a scalar. The output symbol time is the product of the input sample time and the **Samples per symbol** parameter value.

- When you set **Output type** to `Bit`, the output width equals the number of bits per symbol.
- When you set **Output type** to `Integer`, the output is a scalar.

Traceback Depth and Output Delays

Internally, this block creates a trellis description of the modulation scheme and uses the Viterbi algorithm. The **Traceback depth** parameter, D, in this block is the number of trellis branches used to construct each traceback path. D influences the output delay, which is the number of zero symbols that precede the first meaningful demodulated value in the output.

- When you set the **Rate options** parameter to `Allow multirate processing`, and the model uses a variable-step solver or a fixed-step solver with the **Tasking Mode** parameter set to `SingleTasking`, then the delay consists of D+1 zero symbols.
- When you set the **Rate options** parameter to `Enforce single-rate processing`, then the delay consists of D zero symbols.

The optimal **Traceback depth** parameter value is dependent on minimum squared Euclidean distance calculations. Alternatively, a typical value, dependent on the number of states, can be chosen using the “five-times-the-constraint-length” rule, which corresponds to 5×log2(*numStates*). The number of states is determined by the following equation:

$$numStates = \begin{Bmatrix} p \cdot 2^{(L-1)}, \ for\, even\, m \\ 2p \cdot 2^{(L-1)}, \ for\, odd\, m \end{Bmatrix}$$

where:

- *h* = m/p is the modulation index proper rational form
  - *m* = numerator of modulation index
  - *p* = denominator of modulation index
- *L* is the Pulse length

Parameters

Output type

Determines whether the output consists of bipolar or binary values.

Phase offset (rad)

The initial phase of the modulated waveform.

Samples per symbol

The number of input samples that represent each modulated symbol, which must be a positive integer. For more information, see “Upsample Signals and Rate Changes”.

Rate options

Select the rate processing method for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width is the number of symbols (which is given by dividing the input length by the **Samples per symbol** parameter value when the **Output type** parameter is set to `Integer`).
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output period is the same as the symbol period and equals the product of the input period and the **Samples per symbol** parameter value.

For more information, see Single-Rate Processing and Multirate Processing in the Description section of this page.

Traceback depth

The number of trellis branches that the MSK Demodulator Baseband block uses to construct each traceback path.

Output data type

The output data type can be `boolean`, `int8`, `int16`, `int32`, or `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Boolean (When **Output type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (When **Output type** set to `Integer`)

Pair Block

MSK Modulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

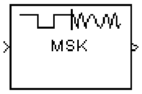
Blocks

CPM Demodulator Baseband | MSK Modulator Baseband | Viterbi Decoder

Introduced before R2006a

MSK Modulator Baseband

Modulate using differentially encoded minimum shift keying method



Library

CPM, in Digital Baseband sublibrary of Modulation

Description

The MSK Modulator Baseband block modulates using the differentially encoded minimum shift keying method. The output is a baseband representation of the modulated signal.

This block accepts a scalar-valued or column vector input signal. For a column vector input signal, the width of the output equals the product of the number of symbols and the value for the **Samples per symbol** parameter.

Pulse Shape Filtering

Differentially encoded minimum shift keying modulation uses pulse shaping to smooth the phase transitions of the modulated signal. The function *q*(*t*) is the phase response obtained from the frequency pulse, *g*(*t*), through this relation: $q(t) = \int_{-\infty}^{t} g(t)dt$

The specified frequency pulse shape corresponds to this rectangular pulse shape expression, *g*(*t*).

Pulse Shape	Expression
`Rectangular`	$g(t) = \begin{cases} \frac{1}{2LT}, & 0 \le t \le LT \\ 0 & \text{otherwise} \end{cases}$

- L_{main} is the main lobe pulse duration in symbol intervals.
- The duration of the pulse, *LT*, is the pulse length in symbol intervals.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, then the block accepts values of 1 and -1.

When you set the **Input type** parameter to `Bit`, then the block accepts values of 0 and 1.

For information about the data types each block port supports, see the “Supported Data Types” on page 5-582 table on this page.

Single-Rate Processing

In single-rate processing mode, the input and output signals have the same port sample time. The block implicitly implements the rate change by making a size change at the output when compared to the input. In this mode, the input to the block can be multiple symbols.

- When you set **Input type** to `Integer`, the input can be a column vector, the length of which is the number of input symbols.
- When you set **Input type** to `Bit`, the input width must be an integer multiple of *K*, the number of bits per symbol.

The output width equals the product of the number of input symbols and the **Samples per symbol** parameter value.

Multirate Processing

In multirate processing mode, the input and output signals have different port sample times. In this mode, the input to the block must be one symbol.

- When you set **Input type** to `Integer`, the input must be a scalar.
- When you set **Input type** to `Bit`, the input width must equal the number of bits per symbol.

The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Parameters

Input type

Indicates whether the input consists of bipolar or binary values.

Phase offset (rad)

The initial phase of the output waveform, measured in radians.

Samples per symbol

The number of output samples that the block produces for each integer or binary word in the input, which must be a positive integer. For all non-binary schemes, as defined by the pulse shapes, this value must be greater than 1.

For more information, see “Upsample Signals and Rate Changes”.

Rate options

Select the rate processing option for the block.

- `Enforce single-rate processing` — When you select this option, the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. The output width equals the product of the number of symbols and the **Samples per symbol** parameter value.
- `Allow multirate processing` — When you select this option, the input and output signals have different port sample times. The output sample time equals the symbol period divided by the **Samples per symbol** parameter value.

Output data type

Specify the block output data type as `double` and `single`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Boolean (when **Input type** set to `Bit`) • 8-, 16-, and 32-bit signed integers (when **Input type** set to `Integer`)
Output	• Double-precision floating point • Single-precision floating point

Pair Block

MSK Demodulator Baseband

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

CPM Modulator Baseband | MSK Demodulator Baseband

Topics

"Compare Filtered QPSK and MSK Signals in Simulink"
"Compare GMSK and MSK Signals in Simulink"

Introduced before R2006a

MSK-Type Signal Timing Recovery

Recover symbol timing phase using fourth-order nonlinearity method

Library

Timing Phase Recovery sublibrary of Synchronization

Description

The MSK-Type Signal Timing Recovery block recovers the symbol timing phase of the input signal using a fourth-order nonlinearity method. This block implements a general non-data-aided feedback method that is independent of carrier phase recovery but requires prior compensation for the carrier frequency offset. This block is suitable for systems that use baseband minimum shift keying (MSK) modulation or Gaussian minimum shift keying (GMSK) modulation.

Inputs

By default, the block has one input port. The input signal could be (but is not required to be) the output of a receive filter that is matched to the transmitting pulse shape, or the output of a lowpass filter that limits the amount of noise entering this block.

This block accepts a scalar-valued or column vector input signal. The input uses *N* samples to represent each symbol, where $N > 1$ is the **Samples per symbol** parameter.

- For a column vector input signal, the block operates in single-rate processing mode. In this mode, the output signal inherits its sample rate from the input signal. The input length must be a multiple of *N*.
- For a scalar input signal, the block operates in multirate processing mode. In this mode, the input and output signals have different sample rates. The output sample rate equals *N* multiplied by the input sample rate.
- This block accepts input signals of type Double or Single

If you set the **Reset** parameter to `On nonzero input via port`, then the block has a second input port, labeled `Rst`. The `Rst` input determines when the timing estimation process restarts, and must be a scalar.

- If the input signal is a scalar value, the sample time of the `Rst` input equals the symbol period
- If the input signal is a column vector, the sample time of the `Rst` input equals the input port sample time
- This block accepts reset signals of type Double or Boolean

Outputs

The block has two output ports, labeled `Sym` and `Ph`:

- The `Sym` output is the result of applying the estimated phase correction to the input signal. This output is the signal value for each symbol, which can be used for decision purposes. The values in the `Sym` output occur at the symbol rate:
  - For a column vector input signal of length *N***R*, the `Sym` output is a column vector of length *R* having the same sample rate as the input signal.
  - For a scalar input signal, the sample rate of the `Sym` output equals *N* multiplied by the input sample rate.
- The `Ph` output gives the phase estimate for each symbol in the input.

  The `Ph` output contains nonnegative real numbers less than *N*. Noninteger values for the phase estimate correspond to interpolated values that lie between two values of the input signal. The sample time of the `Ph` output is the same as that of the `Sym` output.

  Note If the `Ph` output is very close to either zero or **Samples per symbol**, or if the actual timing phase offset in your input signal is very close to zero, then the block's accuracy might be compromised by small amounts of noise or jitter. The block works well when the timing phase offset is significant rather than very close to zero.
- The output signal inherits its data type from the input signal.

Delays

When the input signal is a vector, this block incurs a delay of two symbols. When the input signal is a scalar, this block incurs a delay of three symbols.

Parameters

Modulation type

The type of modulation in the system. Choices are `MSK` and `GMSK`.

Samples per symbol

The number of samples, N, that represent each symbol in the input signal. This must be greater than 1.

Error update gain

A positive real number representing the step size that the block uses for updating successive phase estimates. Typically, this number is less than 1/N, which corresponds to a slowly varying phase.

This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. For more information, see Tunable Parameters (Simulink).

Reset

Determines whether and under what circumstances the block restarts the phase estimation process. Choices are `None`, `Every frame`, and `On nonzero input via port`. The last option causes the block to have a second input port, labeled `Rst`.

Algorithm

This block's algorithm extracts timing information by passing the sampled baseband signal through a fourth-order nonlinearity followed by a digital differentiator whose output is smoothed to yield an error signal. The algorithm then uses the error signal to make the sampling adjustments.

More specifically, this block uses a timing error detector whose result for the kth symbol is e(k), given in [2] by

$$
\begin{aligned}
&e(k) = (^{-}\mathrm{Re}\left\{r^2(kT - T_s + d_{k-1})r^{*2}((k-1)T - T_s + d_{k-2})\right\} \\
&-(^{-}\mathrm{Re}\left\{r^2(kT + T_s + d_{k-1})r^{*2}((k-1)T + T_s + d_{k-1})\right\}
\end{aligned}
$$

where

- r is the block's input signal
- T is the symbol period
- T_s is the sampling period
- * means complex conjugate
- d_k is the phase estimate for the kth symbol
- D is 1 for MSK and 2 for Gaussian MSK modulation

References

[1] D'Andrea, A. N., U. Mengali, and R. Reggiannini, "A Digital Approach to Clock Recovery in Generalized Minimum Shift Keying," *IEEE Transactions on Vehicular Technology*, Vol. 39, No. 3, August 1990, pp. 227-234.

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

See Also

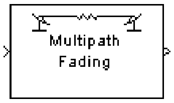
Blocks
Symbol Synchronizer

Introduced before R2006a

Multipath Rayleigh Fading Channel

(To be removed) Simulate multipath Rayleigh fading propagation channel

Note Multipath Rayleigh Fading Channel will be removed in a future release. Use SISO Fading Channel instead.



Multipath Rayleigh Fading Channel

Library

Channels

Description

The Multipath Rayleigh Fading Channel block implements a baseband simulation of a multipath Rayleigh fading propagation channel. You can use this block to model mobile wireless communication systems. For details about fading channels, see the references listed below.

This block accepts a scalar value or column vector input signal. The block inherits sample time from the input signal. The input signal must have a discrete sample time greater than 0.

Relative motion between the transmitter and receiver causes Doppler shifts in the signal frequency. You can specify the Doppler spectrum of the Rayleigh process using the **Doppler spectrum type** parameter. For channels with multiple paths, you can assign each path a different Doppler spectrum, by entering a vector of `doppler` objects in the **Doppler spectrum** field.

Because a multipath channel reflects signals at multiple places, a transmitted signal travels to the receiver along several paths, each of which may have differing lengths and associated time delays. In the block's parameter dialog box, the **Discrete path delay vector** specifies the time delay for each path. If you do not check **Normalize gain vector to 0 dB overall gain**, then the **Average path gain vector** specifies the gain for each path. When you check the box, the block uses a multiple of **Average path gain vector** instead of the **Average path gain vector** itself, choosing the scaling factor so that the channel's effective gain, considering all paths, is 0 dB.

The number of paths is implicitly indicated via the number of elements in **Discrete path delay vector** or **Average path gain vector**. If both of these parameters are vectors, then they must have the same length; if exactly one of these parameters contains a scalar value, then the block expands it into a vector whose size matches that of the other vector parameter.

The block multiplies the input signal by samples of a Rayleigh-distributed complex random process. The scalar **Initial seed** parameter seeds the random number generator and the block generates random numbers using the Ziggurat method.

Double-clicking this block during simulation or selecting **Open channel visualization at start of simulation** plots the channel characteristics using the channel visualization tool. For more information, see “Channel Visualization”.

Parameters

Maximum Doppler shift (Hz)

A positive scalar value that indicates the maximum Doppler shift.

Doppler spectrum type

Specifies the Doppler spectrum of the Rayleigh process.

This parameter defaults to `Jakes` Doppler spectrum. Alternatively, you can also choose any of the following types:

- Flat on page 2-290
- Gaussian on page 2-292
- Rounded on page 2-300
- Restricted Jakes on page 2-297
- Asymmetrical Jakes on page 2-282
- Bi-Gaussian on page 2-287
- Bell on page 2-285

For all Doppler spectrum types except `Jakes` and `Flat`, you can choose one or more parameters to control the shape of the spectrum.

You can also select `Specify as dialog parameter` for the **Doppler spectrum type**. Specify the Doppler spectrum by entering an object in the **Doppler spectrum** field. See the `doppler` function reference for details on how to construct Doppler objects, and also for the meaning of the parameters associated with the various Doppler spectrum types.

Discrete path delay vector (s)

A vector that specifies the propagation delay for each path.

Average path gain vector (dB)

A vector that specifies the gain for each path.

Normalize gain vector to 0 dB overall gain

Checking this box causes the block to scale the **Gain vector** parameter so that the channel's effective gain (considering all paths) is 0 dB.

Initial seed

The scalar seed for the Gaussian noise generator.

Open channel visualization at start of simulation

Select this check box to open the channel visualization tool when a simulation begins.

Complex path gains port

Select this check box to create a port that outputs the values of the complex path gains for each path. In this *N*-by-*M* multichannel output, *N* represents the number of samples the input signal contains and *M* represents the number of discrete paths (number of delays).

Channel filter delay port

Select this check box to create a port that outputs the value of the delay (in samples) that results from the filtering operation of this block. This delay is zero if only one path is simulated, but can be greater than zero if more than one path is present. See “Methodology for Simulating Multipath Fading Channels” for a definition of this delay, where it is denoted as N_1.

Algorithm

This implementation is based on the direct-form simulator described in Reference [1]. A detailed explanation of the implementation, including a review of the different Doppler spectra, can be found in [4].

Some wireless applications prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed *v* making an angle of θ with the direction of wave motion, then the Doppler shift is

$f_\mathrm{d} = (vf/c)\cos\theta$

where *f* is the transmission carrier frequency and *c* is the speed of light. The Doppler frequency represents the maximum Doppler shift arising from motion of the mobile.

Example

Generating Ideal Theoretical BER Results for a Rayleigh Fading Channel

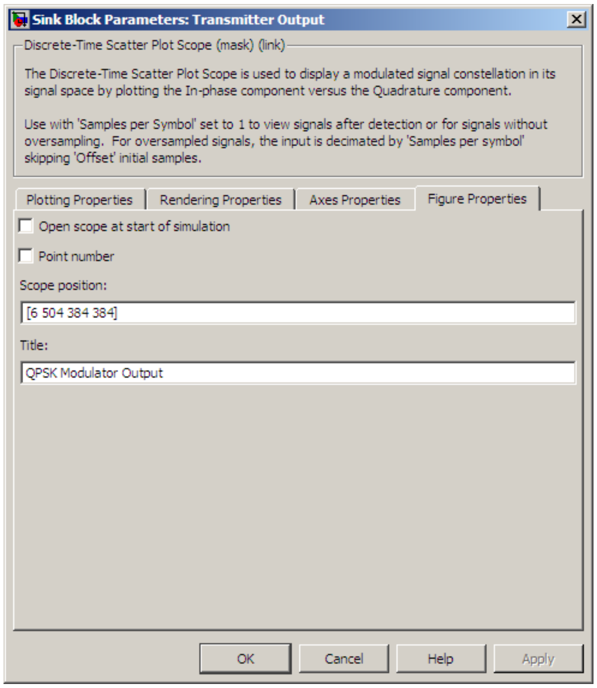
This example illustrates how to generate ideal theoretical BER results for a flat Rayleigh fading channel. The model reproduces known theoretical results and shows the correct BER performance for a flat Rayleigh fading channel. In this example, you will run the model and compare the simulation results to the BERTool theoretical results for verification purposes. Note that the EbNo value for the model's AWGN block is 5 dB. You can change the noise power by double-clicking the AWGN block and entering another numeric value in the EbNo parameter.

Opening the Model

You can open the model by clicking here in the MATLAB Help browser. Alternatively, you can type `doc_qpsk_rayleigh_derotated` at the MATLAB command line.

Running the Model and Comparing Results

1. To run the example, select **Run** in the **Simulate** of the **Simulation**, **Debug**, or **Modeling** tab.
2. After the model collects more than 5000 errors, click the stop button.
3. Close the three scopes.
4. In the Simulink model window, double-click the Transmitter Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then click **OK**.


Sink Block Parameters: Transmitter Output

Discrete-Time Scatter Plot Scope (mask) (link)

The Discrete-Time Scatter Plot Scope is used to display a modulated signal constellation in its signal space by plotting the In-phase component versus the Quadrature component.

Use with 'Samples per Symbol' set to 1 to view signals after detection or for signals without oversampling. For oversampled signals, the input is decimated by 'Samples per symbol' skipping 'Offset' initial samples.

□ Open scope at start of simulation

□ Point number

Scope position: [6 504 384 384]

Title: QPSK Modulator Output

5 In the Simulink model window, double-click the Rayleigh Channel Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then click **OK**.

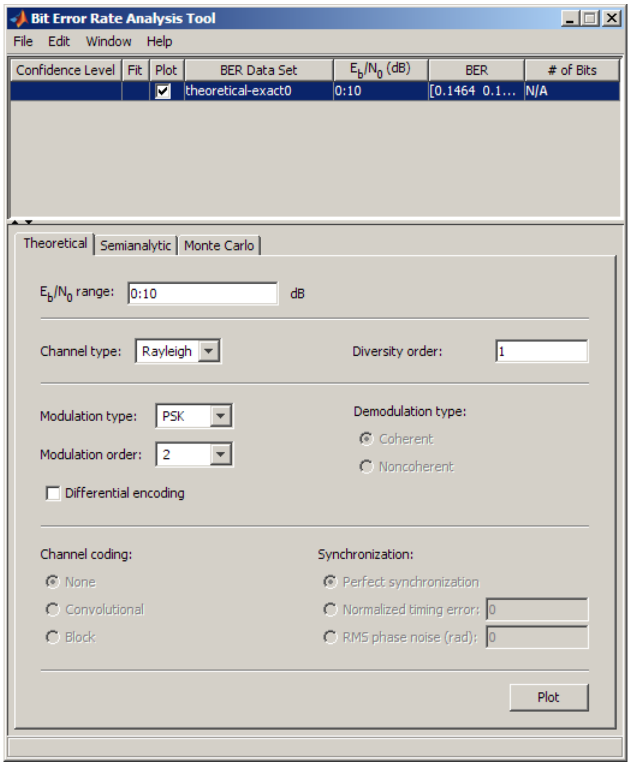
6 In the Simulink model window, double-click the Noisy Rayleigh Channel Output block. In the mask window, click the **Figure Properties** tab, uncheck **Open scope at start of Simulation**, then click **OK**.

7 In the Simulink model window, double-click the Error Rate Calculation block, check **Stop simulation**, enter `5000` for **Target number of error**, then click **OK**.

8 Click the play button to rerun the example.

9 Open BERTool by typing `bertool` at the MATLAB command line.

10 In BERTool, click the **Theoretical** tab and make the following selections:



Confidence Level	Fit	Plot	BER Data Set	E_b/N_0 (dB)	BER	# of Bits
		☑	theoretical-exact0	0:10	[0.1464 0.1...	N/A

E_b/N_0 range: 0:10 dB

Channel type: Rayleigh

Diversity order: 1

Modulation type: PSK

Modulation order: 2

□ Differential encoding

Demodulation type:
● Coherent
○ Noncoherent

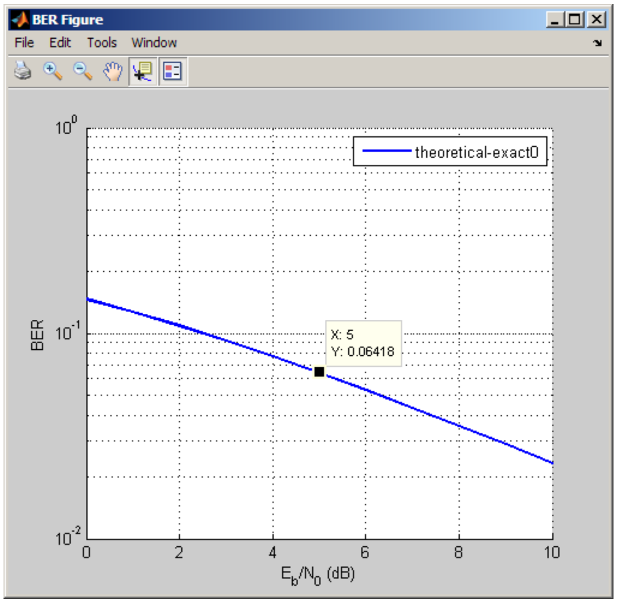
Channel coding:
● None
○ Convolutional
○ Block

Synchronization:
● Perfect synchronization
○ Normalized timing error: 0
○ RMS phase noise (rad): 0

- For **Eb/No range** enter `0:10`
- For **Channel type**, select `Rayleigh`
- For **Diversity Order** enter 1
- For **Modulation Type**, select PSK
- For **Modulation order**, select 4

11 Click **Plot**.

12 Since the Simulink model uses an EbNo value of 5 dB, verify the probability of error on the BERTool curve at 5 dB. The two values should be approximately equal.



Click the Data Cursor button (second from right) and click on the BERTool curve at 5dB.

See Also

Rayleigh Noise Generator, SISO Fading Channel, `doppler`

References

[1] Jeruchim, Michel C., Balaban, Philip, and Shanmugan, K. Sam, *Simulation of Communication Systems*, Second edition, New York, Kluwer Academic/Plenum, 2000.

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

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

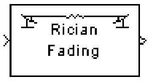
[4] Iskander, Cyril-Daniel, *A MATLAB-based Object-Oriented Approach to Multipath Fading Channel Simulation*, a MATLAB Central submission available from www.mathworks.com.

Introduced before R2006a

Multipath Rician Fading Channel

(To be removed) Simulate multipath Rician fading propagation channel

Note Multipath Rician Fading Channel will be removed in a future release. Use SISO Fading Channel instead.



Library

Channels

Description

The Multipath Rician Fading Channel block implements a baseband simulation of a multipath Rician fading propagation channel. You can use this block to model mobile wireless communication systems when the transmitted signal can travel to the receiver along a dominant line-of-sight or direct path. For more details, see “Fading Channels”.

This block accepts a scalar value or column vector input signal. The block inherits sample time from the input signal. The input signal must have a discrete sample time greater than 0.

Relative motion between the transmitter and receiver causes Doppler shifts in the signal frequency. You can specify the Doppler spectrum of the Rician process using the **Doppler spectrum type** pop-up menu. For channels with multiple paths, you can assign each path a different Doppler spectrum, by entering a vector of doppler objects in the **Doppler spectrum** field.

Because a multipath channel reflects signals at multiple places, a transmitted signal travels to the receiver along several paths, each of which may have differing lengths and associated time delays. In the block's parameter dialog box, the **Discrete path delay vector** specifies the time delay for each path. If you do not check the **Normalize gain vector to 0 dB overall gain** box, then the **Average path gain vector** specifies the gain for each path. When you check the box, the block uses a multiple of **Average path gain vector** instead of the **Average path gain vector** itself, choosing the scaling factor so that the channel's effective gain considering all paths is 0 dB.

The number of paths is implicitly indicated via the number of elements in **Discrete path delay vector** or **Average path gain vector**. If both of these parameters are vectors, they must have the same length; if exactly one of these parameters contains a scalar value, the block expands it into a vector whose size matches that of the other vector parameter.

Fading causes the signal to become diffuse. The **K-factor** parameter, which is part of the statistical description of the Rician distribution, represents the ratio between the power in the line-of-sight component and the power in the diffuse component. The ratio is expressed linearly, not in decibels. While the Average path gain vector parameter controls the overall gain through the channel, the **K-factor** parameter controls the gain's partition into line-of-sight and diffuse components.

You can specify the **K-factor** parameter as a scalar or a vector. If the **K-factor** parameter is a scalar, then the first discrete path of the channel is a Rician fading process (it contains a line-of-sight component) with the specified **K-factor**, while the remaining discrete paths indicate independent Rayleigh fading processes (with no line-of-sight component). If the **K-factor** parameter is a vector of the same size as **Discrete path delay vector**, then each discrete path is a Rician fading process with a **K-factor** given by the corresponding element of the vector. You can attribute the line-of-sight component a Doppler shift, through the **Doppler shift(s) of line-of-sight component(s)** parameter, and an initial phase, through the **Initial phase(s) of line-of-sight component(s)**. The **Doppler shift(s) of line-of-sight component(s)** and **Initial phase(s) of line-of-sight component(s)** parameters must be of the same size as the K-factor parameter.

The block multiplies the input signal by samples of a Rician-distributed complex random process. The scalar **Initial seed** parameter seeds the random number generator and the block generates random numbers using the Ziggurat method.

Double-clicking this block during simulation or selecting the block dialog's check box labeled **Open channel visualization at start of simulation** plots the channel characteristics using the channel visualization tool. For more information, see “Channel Visualization”.

Parameters

K-factor

The ratio of power in the line-of-sight component to the power in the diffuse component. The ratio is expressed linearly, not in decibels. If **K-factor** is a scalar value, then the first discrete path is a Rician fading process (it contains a line-of-sight component) with the specified K-factor, while the remaining discrete paths are independent Rayleigh fading processes (with no line-of-sight component). If **K-factor** is a vector of the same size as **Discrete path delay vector**, then each discrete path is a Rician fading process with a **K-factor** given by the corresponding element of the vector.

Doppler shift(s) of line-of-sight components(s) (Hz)

The Doppler shift of the line-of-sight component. It must be a scalar (if **K-factor** is a scalar) or a vector of the same size as **K-factor**. If this parameter contains a scalar value, then the line-of-sight component of the first discrete path has the specified Doppler shift, while the remaining discrete paths become independent Rayleigh fading processes. If the parameter contains a vector, then the line-of-sight component of each discrete path has a Doppler shift given by the corresponding element of the vector.

Initial phase(s) of line-of-sight component(s) (rad)

The initial phase of the line-of-sight component. It must be either a scalar (if **K-factor** is a scalar value) or a vector of the same size as **K-factor**.

Maximum diffuse Doppler shift (Hz)

A positive scalar value that indicates the maximum diffuse Doppler shift.

Doppler spectrum type

Specifies the Doppler spectrum of the Rician process.

This parameter defaults to `Jakes` Doppler spectrum. Alternately, you can choose any of the following types:

- Flat on page 2-290
- Gaussian on page 2-292

- Rounded on page 2-300
- Restricted Jakes on page 2-297
- Asymmetrical Jakes on page 2-282
- Bi-Gaussian on page 2-287
- Bell on page 2-285

For all Doppler spectrum types except `Jakes` and `Flat`, You can use one or more parameters to control the shape of the spectrum.

You can also select `Specify as dialog parameter` for the **Doppler spectrum type**. Specify the Doppler spectrum by entering an object in the **Doppler spectrum** field. See the `doppler` function reference for details on how to construct doppler objects, and for the meaning of the parameters associated with the various Doppler spectrum types.

Discrete delay vector(s)

A vector that specifies the propagation delay for each path.

Average path gain vector (dB)

A vector that specifies the gain for each path.

Initial seed

The scalar seed for the Gaussian noise generator.

Open channel visualization at start of simulation

Select this check box to open the channel visualization tool when a simulation begins. This block supports channel visualization for a column vector input signal.

Complex path gains port

Select this check box to create a port that outputs the values of the complex path gains for each path. In this N-by-M multichannel output, N represents the number of samples the input contains and M represents the number of discrete paths (number of delays).

Channel filter delay port

Select this check box to create a port that outputs the value of the delay (in samples) that results from the filtering operation of this block. This delay is zero if only one path is simulated, but can be greater than zero if more than one path is present. See “Methodology for Simulating Multipath Fading Channels” for a definition of this delay, where it is denoted as N_1.

Algorithm

This implementation is based on the direct form simulator described in Reference [1]. A detailed explanation of the implementation, including a review of the different Doppler spectra, can be found in [4].

Some wireless applications prefer to specify Doppler shifts in terms of the speed of the mobile. If the mobile moves at speed v making an angle of θ with the direction of wave motion, the Doppler shift is

$$f_d = (vf/c)\cos\theta$$

where f is the transmission carrier frequency and c is the speed of light. The Doppler frequency is the maximum Doppler shift arising from the motion of the mobile.

See Also

Rician Noise Generator, SISO Fading Channel, `doppler`

References

[1] Jeruchim, Michel C., Balaban, P., and Shanmugan, K. Sam, *Simulation of Communication Systems*, Second edition, New York, Kluwer Academic/Plenum, 2000.

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

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

[4] Iskander, Cyril-Daniel, *A MATLAB-based Object-Oriented Approach to Multipath Fading Channel Simulation*, a MATLAB Central submission available from www.mathworks.com.

Introduced in R2006a

Mu-Law Compressor

Implement µ-law compressor for source coding

Library

Source Coding

Description

The Mu-Law Compressor block implements a µ-law compressor for the input signal. The formula for the µ-law compressor is

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

where µ is the µ-law parameter of the compressor, V is the peak magnitude of x, log is the natural logarithm, and sgn is the signum function (`sign` in MATLAB).

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

mu value

The µ-law parameter of the compressor.

Peak signal magnitude

The peak value of the input signal. This is also the peak value of the output.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

Mu-Law Expander

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

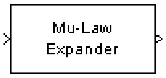
Blocks

A-Law Compressor | Mu-Law Expander

Introduced before R2006a

Mu-Law Expander

Implement µ-law expander for source coding



Library

Source Coding

Description

The Mu-Law Expander block recovers data that the Mu-Law Compressor block compressed. The formula for the µ-law expander, shown below, is the inverse of the compressor function.

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

The input can have any shape or frame status. This block processes each vector element independently.

Parameters

mu value

The µ-law parameter of the compressor.

Peak signal magnitude

The peak value of the input signal. This is also the peak value of the output.

Supported Data Type

Port	Supported Data Types
In	• double
Out	• double

Pair Block

Mu-Law Compressor

References

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

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

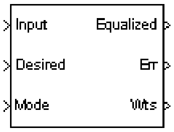
A-Law Expander | Mu-Law Compressor

Introduced before R2006a

Normalized LMS Decision Feedback Equalizer

(To be removed) Equalize using decision feedback equalizer that updates weights with normalized LMS algorithm

Note will be removed in a future release. Consider using Decision Feedback Equalizer instead.



Library

Equalizer Block

Description

The Normalized LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the normalized LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the normalized LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer. When you set the **Number of samples per symbol** parameter to a value greater than 1, , the weights are updated once every N^{th} sample, for a *T*/*N*-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Parameters

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps

The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of forward taps in the equalizer.

Step size

The step size of the normalized LMS algorithm.

Leakage factor

The leakage factor of the normalized LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Bias

The bias parameter of the normalized LMS algorithm, a nonnegative real number. This parameter is used to overcome difficulties when the algorithm's input signal is small.

Initial weights

A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

If you select this check box, the block has an input port that enables you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. The equalizer will train for the length of the Desired signal. If the mode input is not present, the equalizer will train at the beginning of every frame for the length of the Desired signal.

Output error

If you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Compatibility Considerations

Normalized LMS Decision Feedback Equalizer will be removed

Warns starting in R2020a

- Normalized LMS Decision Feedback Equalizer will be removed in a future release. Consider using Decision Feedback Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Decision Feedback Equalizer block is equivalent to the **Mode input port** parameter of the Normalized LMS Decision Feedback Equalizer block.
- The Decision Feedback Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Normalized LMS Decision Feedback Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

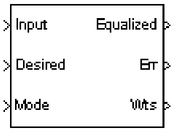
“Equalization”

Introduced before R2006a

Normalized LMS Linear Equalizer

(To be removed) Equalize using linear equalizer that updates weights with normalized LMS algorithm

Note will be removed in a future release. Consider using Linear Equalizer instead.



Library

Equalizers

Description

The Normalized LMS Linear Equalizer block uses a linear equalizer and the normalized LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the normalized LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, the block implements a symbol-spaced (i.e. T-spaced) equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the weights are updated once every N^{th} sample, for a *T*/*N*-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Parameters

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of taps in the equalizer.

Step size

The step size of the normalized LMS algorithm.

Leakage factor

The leakage factor of the normalized LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Bias

The bias parameter of the normalized LMS algorithm, a nonnegative real number. This parameter is used to overcome difficulties when the algorithm's input signal is small.

Initial weights

A vector that lists the initial weights for the taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

If you check this box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

If you check this box, the block outputs the current weights.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Compatibility Considerations

Normalized LMS Linear Equalizer will be removed

Warns starting in R2020a

- Normalized LMS Linear Equalizer will be removed in a future release. Consider using Linear Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Linear Equalizer block is equivalent to the **Mode input port** parameter of the Normalized LMS Linear Equalizer block.
- The Linear Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Normalized LMS Linear Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

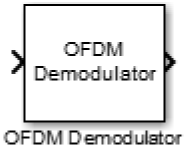
Topics

“Equalization”

Introduced before R2006a

OFDM Demodulator Baseband

Demodulate orthogonal frequency division modulated data



Library

OFDM, in Digital Baseband sublibrary of Modulation

Description

The Orthogonal Frequency Division Modulation (OFDM) Demodulator Baseband block demodulates an OFDM input signal. The block accepts a single input and has one or two output ports, depending on the status of **Pilot output port**.

Signal Dimensions

Pilot Output Port	Pilot Carrier Indices	Signal Input	Signal Output	Pilot Output
false	N/A	$N_{CPTotal}$ $+N_{FFT}×N_{sym}$-by-N_r	N_{data}-by-N_{sym}-by-N_r	N/A
true	2-D			N_{pilot}-by-N_{sym}-by-N_r
	3-D			N_{pilot}-by-N_{sym}-by-N_t-by-N_r

where

- N_{CP} represents the cyclic prefix length as determined by **Cyclic prefix length**.
- $N_{CPTotal}$ represents the cyclic prefix length over all the symbols. When N_{CP} is a scalar, $N_{CPTotal} = N_{CP} \times N_{sym}$. When N_{CP} is a row vector, $N_{CPTotal} = \sum N_{CP}$.
- N_{FFT} represents the number of subcarriers as determined by **FFT length**.
- N_{sym} represents the number of symbols as determined by **Number of OFDM symbols**.
- N_r represents the number of receive antennas as determined by **Number of receive antennas**.
- N_{data} represents the number of data subcarriers. For further information on how N_{data} is determined, see the `info` reference page.
- N_{pilot} represents the number of pilot symbols determined by the second dimension in the **Pilot subcarrier indices** array.
- N_t represents the number of transmit antennas. This parameter is derived from the third dimension of the **Pilot subcarrier indices** array.

Parameters

FFT Length

Specify the FFT length, which is equivalent to the number of subcarriers. The length of the FFT, N_{FFT}, must be greater than or equal to 8.

Number of guard bands

Assign the number of subcarriers to the left, N_{leftG}, and right, N_{rightG}, guard bands. The input is a 2-by-1 vector. The number of subcarriers must fall within $[0,N_{FFT}/2 - 1]$.

Remove DC carrier

Select to remove the DC subcarrier.

Pilot output port

Select to separate the data from the pilot signal and output the demodulated pilot signal.

Pilot subcarrier indices

Specify the pilot subcarrier indices. This field is available only when the **Pilot output port** check box is selected. You can assign the indices can be assigned to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the indices' array vary from 1 to 3. Valid pilot indices fall in the range

$$[N_{\mathrm{leftG}} + 1,\ N_{\mathrm{FFT}}/2] \cup \left[N_{\mathrm{FFT}}/2 + 2,\ N_{\mathrm{FFT}} - N_{\mathrm{rightG}}\right],$$

where the index value cannot exceed the number of subcarriers. If the number of transmit antennas is greater than one, ensure that the indices per symbol are mutually distinct across antennas to minimize interference.

Cyclic prefix length

Specify the length of the cyclic prefix. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same length through all antennas.

Number of OFDM symbols

Specify the number of OFDM symbols, N_{sym}, in the time-frequency grid.

Number of receive antennas

Specify the number of receive antennas, N_r, as a positive integer such that $N_r \le 64$.

Simulate using

Select the simulation type from these choices:

- `Code generation`
- `Interpreted execution`

Algorithms

This block implements the algorithm, inputs, and outputs described in the `OFDM Demodulator` System object reference page. The object properties correspond to the block parameters.

Supported Data Types

Port	Supported Data Types
Input	- Double-precision floating point
Pilot (optional)	- Double-precision floating point
Output	- Double-precision floating point

Pair Block

OFDM Modulator Baseband

References

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband*.London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX*.Upper Saddle River, NJ: Prentice Hall, 2007.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
OFDM Modulator Baseband | QPSK Demodulator Baseband | Rectangular QAM Demodulator Baseband

Objects
`comm.OFDMDemodulator`

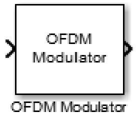
Topics
"IEEE 802.16-2009 WirelessMAN-OFDMA PHY Downlink PUSC"
"Digital Video Broadcasting - Terrestrial"

Introduced in R2014a

OFDM Modulator Baseband

Modulate using orthogonal frequency division modulation

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / OFDM



Description

The OFDM Modulator Baseband block applies Orthogonal Frequency Division Multiplexing modulation to an incoming data signal. The output is a baseband representation of the OFDM modulated signal.

Ports

Input

In — Input signal
3-D array

Input signal, specified as a 3D vector. The block accepts one or two inputs depending on the state of the **Pilot input port**. The input signal dimensions are :

Pilot Input Port	Signal Input	Pilot Input
off	N $_{data}$ -by-N $_{sym}$ -by-N $_{t}$	N/A
on		N $_{pilot}$ -by-N $_{sym}$ -by-N $_{t}$

where

- N_{data} represents the number of data subcarriers. For further information on how N_{data} is determined, see the `info` reference page.
- N_{sym} represents the number of symbols determined by **Number of OFDM symbols**.
- N_{t} represents the number of transmit antennas determined by **Number of transmit antennas**.
- N_{pilot} represents the number of pilot symbols determined by the first dimension size in the **Pilot subcarrier indices** array.
- N_{CP} represents the cyclic prefix length as determined by **Cyclic prefix length**.
- $N_{CPTotal}$ represents the cyclic prefix length over all the symbols. When N_{CP} is a scalar, $N_{CPTotal} = N_{CP} \times N_{sym}$. When N_{CP} is a row vector, $N_{CPTotal} = \sum N_{CP}$.
- N_{FFT} represents the number of subcarriers as determined by **FFT length**.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Baseband modulated signal
2-D array

Baseband modulated signal, returned as a 2D array. The datatype of the output follows the input datatype. The output signal has dimension (N $_{CP}$ +N $_{FFT}$) ×N $_{sym}$ -by-N $_{t}$.

Parameters

`FFT Length` — Number of DFT points
64 (default) | positive integer

Number of DFT points, specified as a positive integer. The length of the FFT, N_{FFT}, must be greater than or equal to 8 and is equivalent to the number of subcarriers.

`Number of guard bands` — Number of subcarriers to the left and right guard bands
`[6;5]` (default) | 2-by-1 integer-valued vector

Number of subcarriers allocated to the left and right guard bands, specified as a 2-by-1 integer-valued vector. The number of subcarriers must fall within $[0,\lfloor N_{FFT}/2\rfloor - 1]$ where you specify the left, N_{leftG}, and right, N_{rightG}, guard bands independently in a 2-by-1 column vector.

`Insert DC null` — Option to insert DC null
`off` (default) | `on`

Select this parameter to insert a null on the DC subcarrier.

`Pilot input port` — Option to specify pilot input port
`off` (default) | `on`

Select this parameter to allow the specifying of pilot input port.

`Pilot subcarrier indices` — Pilot subcarrier indices
`[12; 26; 40; 54]` (default) | column vector

Pilot subcarrier indices, specified as a column vector. This field is available only when the **Pilot input port** check box is selected. You can assign the indices to the same or different subcarriers for each symbol. Similarly, the pilot carrier indices can differ across multiple transmit antennas. Depending on the desired level of control for index assignments, the dimensions of the indices array vary. Valid pilot indices fall in the range

$$[N_{\mathrm{leftG}} + 1,\ N_{\mathrm{FFT}}/2] \cup \left[N_{\mathrm{FFT}}/2 + 2,\ N_{\mathrm{FFT}} - N_{\mathrm{rightG}}\right],$$

where the index value cannot exceed the number of subcarriers. When the pilot indices are the same for every symbol and transmit antenna, the property has dimensions N_{pilot}-by-1. When the pilot indices vary across symbols, the property has dimensions of N_{pilot}-by-N_{sym}. If there is only one symbol but multiple transmit antennas, the property has dimensions of N_{pilot}-by-1-by-N_t. If the indices vary across the number of symbols and transmit antennas, the property will have dimensions of N_{pilot}-by-N_{sym}-by-N_t. If the number of transmit antennas is greater than one, ensure that the indices per symbol are mutually distinct across antennas to minimize interference. The default value is `[12; 26; 40; 54]`.

`Cyclic prefix length` — Length of cyclic prefix
16 (default) | positive scalar | positive vector

Length of cyclic prefix, specified as a positive integer. If you specify a scalar, the prefix length is the same for all symbols through all antennas. If you specify a row vector of length N_{sym}, the prefix length can vary across symbols but remains the same through all antennas.

Apply raised cosine windowing between OFDM symbols — Option to apply raised cosine window between OFDM symbols

off (default) | on

Select this parameter to apply raised cosine windowing between OFDM symbols. Windowing is the process in which the OFDM symbol is multiplied by a raised cosine window before transmission to reduce the power of out-of-band subcarriers, which serves to reduce spectral regrowth.

Window length — Length of raised cosine window

1 (default) | positive scalar

Length of raised cosine window, specified as a positive scalar. This field is available only when **Apply raised cosine windowing between OFDM symbols** is selected. Use positive integers having a maximum value no greater than the minimum cyclic prefix length. For example, in a configuration in which there are four symbols with cyclic prefix lengths of [12 16 14 18], the window length cannot exceed 12.

Number of OFDM symbols — Number of OFDM symbols

1 (default) | positive scalar

Number of OFDM symbols in the time-frequency grid, specified as a positive scalar.

Number of transmit antennas — Number of transmit antennas

1 (default) | positive scalar

Number of transmit antennas, specified as a real positive scalar. Specify the number of transmit antennas, N_t, as a positive integer such that $N_t \leq 64$.

Simulate using — Type of simulation to run

Code generation (default) | Interpreted execution

Type of simulation to run, specified as Code generation or Interpreted execution.

- Code generation –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than Interpreted execution.
- Interpreted execution –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the Code generation method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

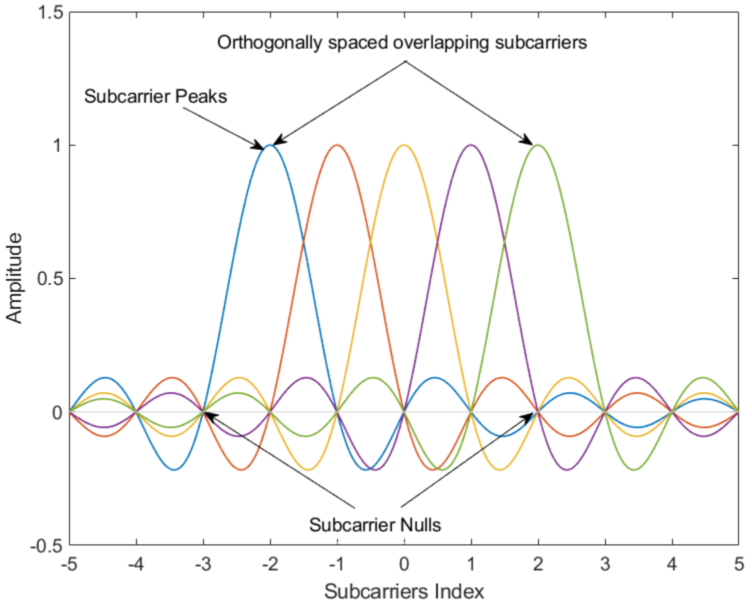
Data Types	double
Multidimensional Signals	yes
Variable-Size Signals	no

More About

Orthogonal Frequency Division Modulation

OFDM operation divides a high-rate data stream into lower data rate substreams by decomposing the transmission frequency band into N contiguous individually modulated subcarriers. Multiple parallel and orthogonal subcarriers carry the samples with almost the same bandwidth as a wideband channel. By using narrow orthogonal subcarriers, the OFDM signal gains robustness over a frequency-selective fading channel and eliminates adjacent subcarrier interference. Intersymbol interference (ISI) is reduced because the lower data rate substreams have symbol durations larger than the channel delay spread.

The Frequency domain representation of orthogonal subcarriers in an OFDM waveform looks as follows:



The transmitter applies inverse fast Fourier transform (IFFT) to N symbols at a time. The output of the IFFT is the sum of the N orthogonal sinusoids:

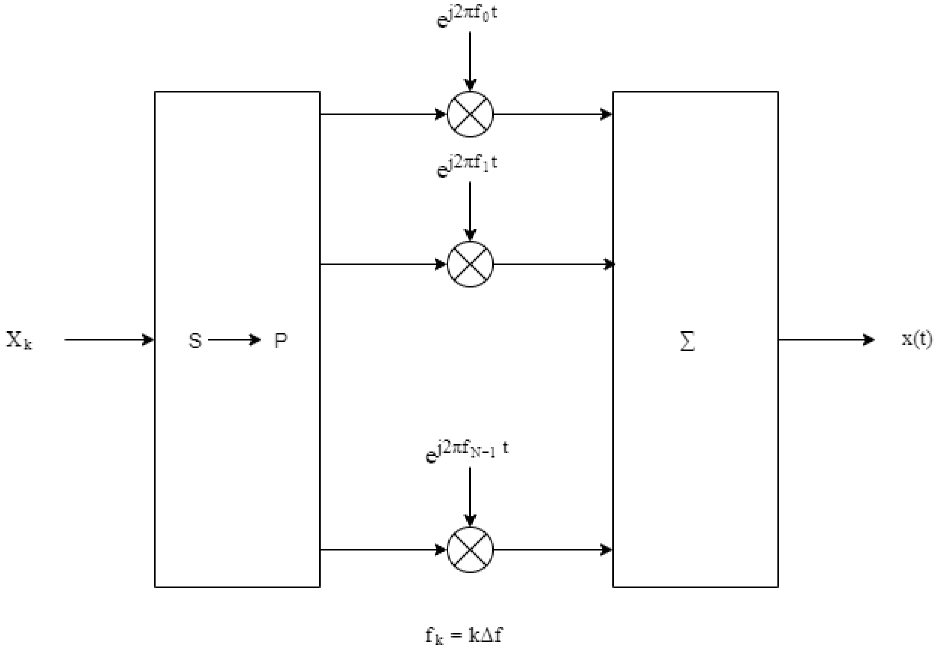
$$x(t) = \sum_{k=0}^{N-1} X_k e^{j2\pi k \Delta f t}, \quad 0 \le t \le T,$$

where $\{X_k\}$ are data symbols, and T is the OFDM symbol time. The data symbols X_k are typically complex and can be from any digital modulation alphabet (for example, QPSK, 16-QAM, 64-QAM).

The subcarrier spacing is $\Delta f = 1/T$; ensuring that the subcarriers are orthogonal over each symbol period, as shown below:

$$\frac{1}{T}\int_0^T \left(e^{j2\pi m \Delta f t}\right)^* \left(e^{j2\pi n \Delta f t}\right) dt = \frac{1}{T}\int_0^T e^{j2\pi(m-n)\Delta f t}\, dt = 0 \quad \text{for } m \ne n\,.$$

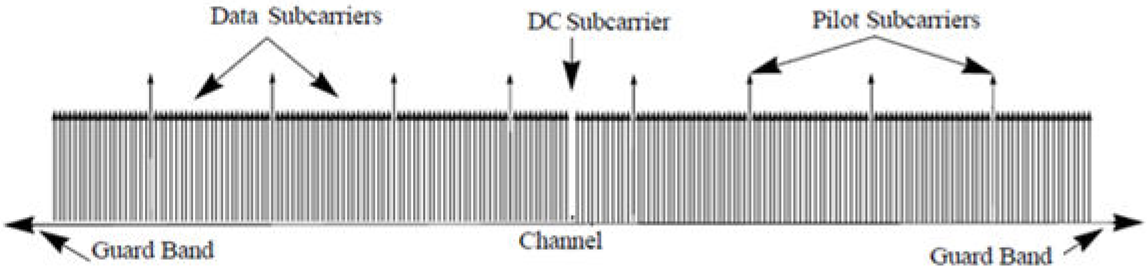
An OFDM modulator consists of a serial-to-parallel conversion followed by a bank of N complex modulators, individually corresponding to each OFDM subcarrier.



Subcarrier Allocation, Guard Bands and Guard Intervals

Individual OFDM subcarriers are allocated as data, pilot, or null subcarriers.

As shown here, subcarriers are designated as data, DC, pilot, or guard band subcarriers.

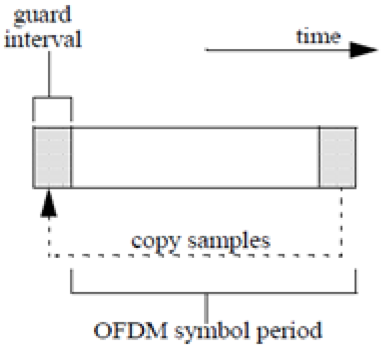


- Data subcarriers transmit user data.
- Pilot subcarriers are used for channel estimation.
- Null subcarriers transmit no data. Subcarriers with no data are used to provide a DC null and serve as buffers between OFDM resource blocks.
  - The null DC subcarrier is the center of the frequency band with an index value of (`nfft`/2 + 1) if `nfft` is even, or ((`nfft` + 1) / 2) if `nfft` is odd.
  - The guard bands provide buffers between consecutive OFDM symbols to protect the integrity of transmitted signals by reducing intersymbol interference.

Null subcarriers enable you to model guard bands and DC subcarrier locations for specific standards, such as the various 802.11 formats, LTE, WiMAX, or for custom allocations. You can allocate the location of nulls by assigning a vector of null subcarrier indices.

Similar to guard bands, guard intervals are used in OFDM to protect the integrity of transmitted signals by reducing intersymbol interference.

Assignment of guard intervals is analogous to the assignment of guard bands. You can model guard intervals to provide temporal separation between OFDM symbols. The guard intervals help preserve intersymbol orthogonality after the signal passes through time-dispersive channels. Guard intervals are created by using cyclic prefixes. Cyclic prefix insertion copies the last part of an OFDM symbol as the first part of the OFDM symbol.



As long as the span of the time dispersion does not exceed the duration of the cyclic prefix, the benefit of cyclic prefix insertion is maintained.

Inserting a cyclic prefix results in a fractional reduction of user data throughput because the cyclic prefix occupies bandwidth that could be used for data transmission.

Raised Cosine Windowing

While the cyclic prefix creates a guard period in time domain to preserve orthogonality, an OFDM symbol rarely begins with the same amplitude and phase exhibited at the end of the prior OFDM symbol causing spectral regrowth and therefore, spreading of signal bandwidth due to intermodulation distortion. To limit this spectral regrowth, it is desired to create a smooth transition between the last sample of a symbol and the first sample of the next symbol. This can be done by using a cyclic suffix and raised cosine windowing.

To create the cyclic suffix, the first N_{WIN} samples of a given symbol are appended to the end of that symbol. However, in order to comply with the 802.11g standard, for example, the length of a symbol cannot be arbitrarily lengthened. Instead, the cyclic suffix must overlap in time and is effectively summed with the cyclic prefix of the following symbol. This overlapped segment is where windowing is applied. Two windows are applied, one of which is the mathematical inverse of the other. The first raised cosine window is applied to the cyclic suffix of symbol k and decreases from 1 to 0 over its duration. The second raised cosine window is applied to the cyclic prefix of symbol k+1 and increases from 0 to 1 over its duration. This process provides a smooth transition from one symbol to the next.

The raised cosine window, $w(t)$, in the time domain can be expressed as:

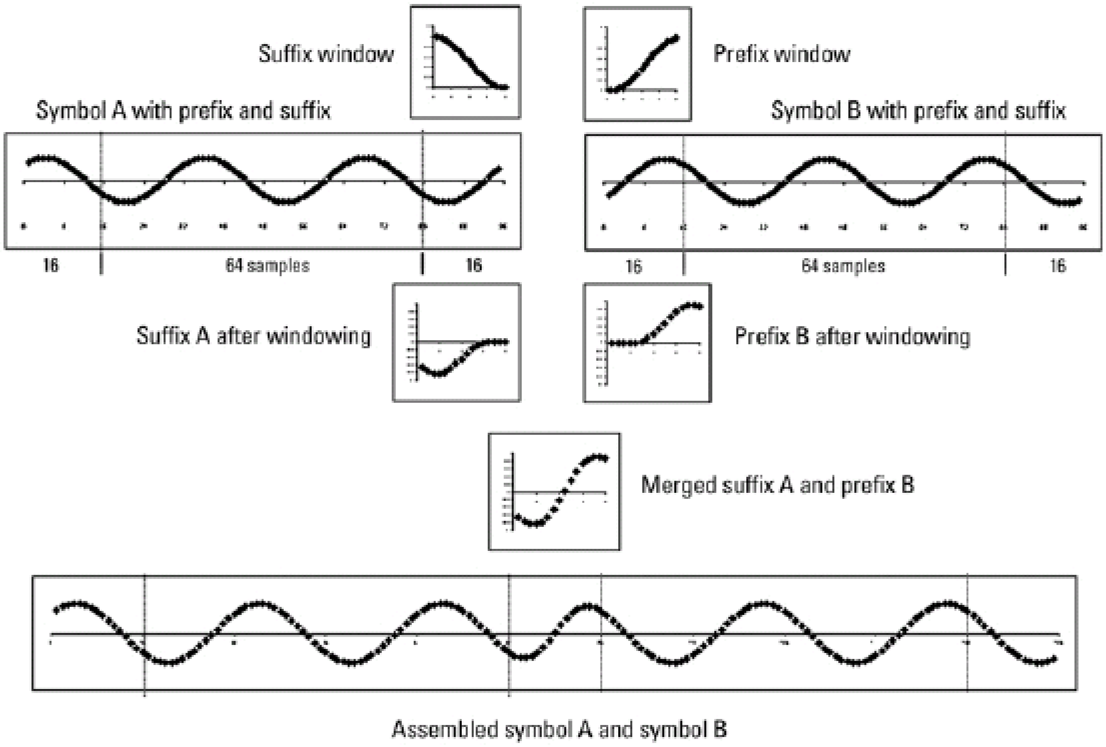
$$w(t) = \begin{cases} 1,\ 0 \le |t| < \dfrac{T - T_W}{2} \\ \dfrac{1}{2}\left\{1 + \cos\left[\dfrac{\pi}{T_W}\left(|t| - \dfrac{T - T_W}{2}\right)\right]\right\}, \dfrac{T - T_W}{2} \le |t| \le \dfrac{T + T_W}{2} \\ 0, \text{otherwise} \end{cases}$$

where:

- T is the OFDM symbol duration including the guard interval.
- T_W is the duration of the window.

Adjust the length of the cyclic suffix via the window length setting property, with suffix lengths set between 1 and the minimum cyclic prefix length. While windowing improves spectral regrowth, it does so at the expense of multipath fading immunity. This occurs because redundancy in the guard band is reduced because the guard band sample values are compromised by the smoothing.

The following figures display the application of raised cosine windowing.


Suffix window
Prefix window
Symbol A with prefix and suffix
Symbol B with prefix and suffix
16
64 samples
16
16
64 samples
16
Suffix A after windowing
Prefix B after windowing
Merged suffix A and prefix B
Assembled symbol A and symbol B

References

[1] Dahlman, E., S. Parkvall, and J. Skold. *4G LTE/LTE-Advanced for Mobile Broadband* .London: Elsevier Ltd., 2011.

[2] Andrews, J. G., A. Ghosh, and R. Muhamed. *Fundamentals of WiMAX* .Upper Saddle River, NJ: Prentice Hall, 2007.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

OFDM Demodulator Baseband | QPSK Modulator Baseband | Rectangular QAM Modulator Baseband

Objects

`comm.OFDMModulator`

Introduced in R2014a

OQPSK Demodulator Baseband

Demodulation using OQPSK method

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM

Description

The OQPSK Demodulator Baseband block applies pulse shape filtering to the input waveform and demodulates it using the offset quadrature phase shift keying (OQPSK) method. For more information, see “Pulse Shaping Filter” on page 5-621. The input is a baseband representation of the modulated signal.

For information about delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 5-620.

Ports

Input

`In` — Input baseband waveform
scalar | column vector

Input baseband waveform, specified as a discrete-time complex scalar or column vector.

The block processes the input signal based on the Output type setting.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Output data
integer column vector | bit column vector

Output data, returned as an integer or bit column vector.

Parameters

Modulation

`Output type` — Output type
`Integer` (default) | `Bit`

Output type, specified as `Integer` or `Bit`.

- When you set **Output type** to `Integer`, the block outputs a vector of integer symbols with values from 0 to 3, the length of which is the number of output symbols.
- When you set **Output type** to `Bit`, the block outputs a 2-bit binary representation of integers, in a binary-valued, even-length vector.

The input period for each integer or bit pair is the Samples per symbol times the output sample period.

`Phase offset (rad)` — Phase of zeroth point of signal constellation

`0` (default) | scalar

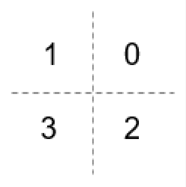
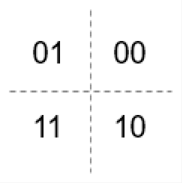
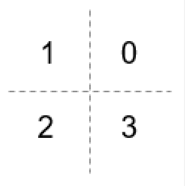
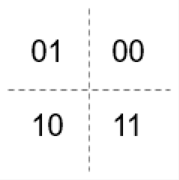
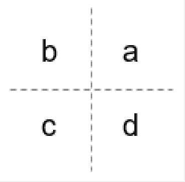
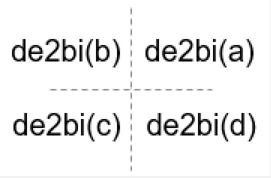
Phase offset from π/4, specified as a scalar in radians. The phase offset is applied to the zeroth point of the signal constellation before delay of quadrature component. After the OQPSK imaginary-component delay the signal is normalized with unity power.

Example: Setting **Phase offset (rad)** to `pi/4` aligns the zeroth point of the QPSK signal constellation point on the axes, {(1,0), (0,j), (-1,0), (0,-j)}.

`Symbol mapping` — Signal constellation bit mapping

`Gray` (default) | `Binary` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `Gray`, `Binary`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Gray`			The signal constellation mapping is Gray-encoded.
`Binary`			The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3			Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Filtering

`Pulse shape` — Filtering pulse shape

`Half sine` (default) | `Normal raised cosine` | `Root raised cosine` | `Custom`

Select the filtering pulse shape: `Half sine`, `Normal raised cosine`, `Root raised cosine`, or `Custom`.

`Rolloff factor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar from `0` to `1`.

Dependencies

This property is enabled when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter span (in symbols)` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to **Filter span (in symbols)** symbols.

Dependencies

This property is enabled when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter numerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This parameter is enabled when Pulse shape is `Custom`.

Data Types: `double`

`Samples per symbol` — Number of samples per symbol

`4` (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

Other Parameters

`Rate options` — Processing rate option

`Enforce single-rate processing` (default) | `Allow multirate processing`

- `Enforce single-rate processing` — Executes the model, ensuring that the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. For integer outputs, the output width equals 1/Samples per symbol times the input width.

  For more information, see Single-Rate Processing with OQPSK Demodulator Block.
- `Allow multirate processing` — Executes the model, allowing the input and output signals to have different port sample times. The output symbol time is Samples per symbol times the input sample time.

  For more information, see Multirate Processing with OQPSK Demodulator Block.

`Output data type` — Output data type

`double` (default) | `single` | `uint8`

Select the output data type: `double`, `single`, or `uint8`.

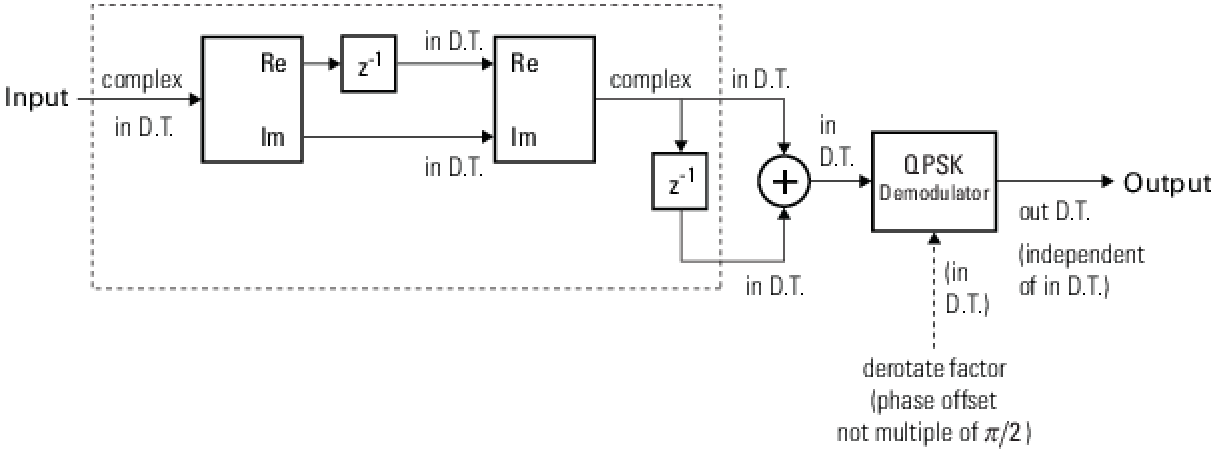
Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

OQPSK Signal Flow Diagram

Every Samples per symbol input samples produce one output symbol. In this figure, the dotted line represents the region comprising the input sample processing.



Modulation Delays

Digital modulation and demodulation blocks incur delays between their inputs and outputs that result in an offset in the arrival time of the received data. Data that enters a modulation or demodulation block at time *T* appears in the output at time *T*+delay. Take system delays into account when comparing transmitted data with received data, such as in overlaid plots or when computing error statistics. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter, input/output data setting, and simulation configuration.

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
`Half sine` or `Custom`	`Enforce single-rate operation`	N/A	`Integer`	1
			`Bit`	2
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + 1 + 1
			`Bit`	length(data) + 2 + 2
		`true` (multitasking)	`Integer`	length(data) + 1 + 2
			`Bit`	length(data) + 2 + 4
`Normal raised cosine` or `Root raised cosine`	`Enforce single-rate operation`	N/A	`Integer`	**Filter span (in symbols)**
			`Bit`	2***Filter span (in symbols)**
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + **Filter span (in symbols)** + 1
			`Bit`	length(data) + 2***Filter span (in symbols)** + 2
		`true` (multitasking)	`Integer`	2*length(data) + **Filter span (in symbols)** + 2
			`Bit`	2*length(data) + 2***Filter span (in symbols)** + 4
(*) The data type parameter is **Input type** for modulation and **Output type** for demodulation.				

Pulse Shaping Filter

The OQPSK modulation scheme requires oversampling of two or greater in order to delay (or offset) the quadrature channel by 90 degrees. This oversampling is achieved through interpolation filtering implemented by pulse shaping.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
OQPSK Modulator Baseband | QPSK Demodulator Baseband

Objects
`comm.OQPSKDemodulator`

Topics
Phase Modulation

Introduced before R2006a

OQPSK Modulator Baseband

Modulation using OQPSK method

Library: Communications Toolbox / Modulation / Digital Baseband Modulation / PM

Description

The OQPSK Modulator Baseband block modulates the input signal using the offset quadrature phase shift keying (OQPSK) method and applies pulse shape filtering to the waveform. For more information, see “Pulse Shaping Filter” on page 5-627. The output is a baseband representation of the modulated signal.

For information about delays incurred by modulator-demodulator pair processing, see “Modulation Delays” on page 5-626.

Ports

Input

`In` — Input data
integer column vector | bit column vector

Input data, specified as an integer or bit column vector.

The input signal is processed based on the setting selected for Input type.

Data Types: `double`

Output

`Out` — Output baseband waveform
column vector

Output baseband waveform, returned as a column vector of complex data.

Parameters

Modulation

`Input type` — Input type
`Integer` (default) | `Bit`

Input type, specified as `Integer` or `Bit`.

- When you set **Input type** to `Integer`, the input can be a scalar value or column vector, the length of which is the number of input symbols.

- When you set **Input type** to `Bit`, the input width must be an integer multiple of two.

The output sample period is the period of each integer or bit pair in the input divided by Samples per symbol.

`Phase offset (rad)` — Phase of zeroth point of signal constellation

`0` (default) | scalar

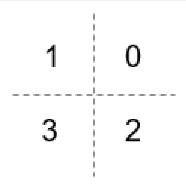
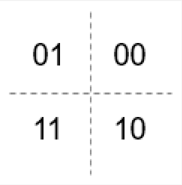
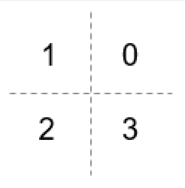
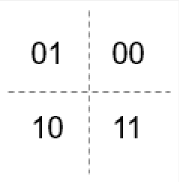
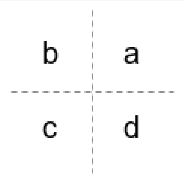
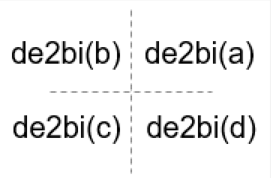
Phase offset from π/4, specified as a scalar in radians. The phase offset is applied to the zeroth point of the signal constellation before delay of quadrature component. After the OQPSK imaginary-component delay, the signal is normalized with unity power.

Example: Setting **Phase offset (rad)** to `pi/4` aligns the zeroth point of the QPSK signal constellation point on the axes, {(1,0), (0,j), (-1,0), (0,-j)}.

`Symbol mapping` — Signal constellation bit mapping

`Gray` (default) | `Binary` | custom 4-element numeric vector of integers with values from 0 to 3

Signal constellation bit mapping, specified as `Gray`, `Binary`, or a custom 4-element numeric vector of integers with values from 0 to 3.

Setting	Constellation Mapping for Integers	Constellation Mapping for Bits	Comment
`Gray`			The signal constellation mapping is Gray-encoded.
`Binary`			The signal constellation mapping for the input integer `m` (0 ≤ `m` ≤ 3) is the complex value $e^{(j*(PhaseOffset+\pi/4) + j*2*\pi*m/4)}$.
Custom 4-element numeric vector of integers with values from 0 to 3			Elements [a b c d] must be composed of the set of values [0, 1, 2, 3] in any order.

Filtering

`Pulse shape` — Filtering pulse shape

`Half sine` (default) | `Normal raised cosine` | `Root raised cosine` | `Custom`

Select the filtering pulse shape: `Half sine`, `Normal raised cosine`, `Root raised cosine`, or `Custom`.

`Rolloff factor` — Raised cosine filter rolloff factor

`0.2` (default) | scalar

Raised cosine filter rolloff factor, specified as a scalar in the range [0, 1].

Dependencies

This property is enabled when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter span (in symbols)` — Filter length

`10` (default) | scalar

Filter length in symbols, specified as a scalar. An ideal raised cosine filter has an infinite impulse response. However, to realize a practical implementation of this filter, the object truncates the impulse response to **Filter span (in symbols)** symbols.

Dependencies

This property is enabled when Pulse shape is `Normal raised cosine` or `Root raised cosine`.

Data Types: `double`

`Filter numerator` — Filter numerator

`[0.7071 0.7071]` (default) | row vector

Filter numerator, specified as a row vector.

Dependencies

This parameter is enabled when Pulse shape is `Custom`.

Data Types: `double`

`Samples per symbol` — Number of samples per symbol

`4` (default) | positive even integer

Number of samples per symbol, specified as a positive even integer.

Data Types: `double`

Other Parameters

`Rate options` — Processing rate option

`Enforce single-rate processing` (default) | `Allow multirate processing`

- `Enforce single-rate processing` — Executes the model, ensuring that the input and output signals have the same port sample time. The block implements the rate change by making a size change at the output when compared to the input. For integer inputs, the output width equals Samples per symbol times the number of symbols.

  For more information, see Single-Rate Processing with OQPSK Modulator Block.
- `Allow multirate processing` — Executes the model, allowing the input and output signals to have different port sample times. The output sample time equals the symbol period divided by Samples per symbol.

For more information, see Single-Rate Processing with OQPSK Modulator Block.

`Output data type` — Output data type

`double` (default) | `single`

Select the output data type: `double` or `single`.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Modulation Delays

Digital modulation and demodulation blocks incur delays between their inputs and outputs that result in an offset in the arrival time of the received data. Data that enters a modulation or demodulation block at time *T* appears in the output at time *T*+delay. Take system delays into account when comparing transmitted data with received data, such as in overlaid plots or when computing error statistics. As shown here, the OQPSK modulation-demodulation delay varies depending on the pulse shaping filter, input/output data setting, and simulation configuration.

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
`Half sine` or `Custom`	`Enforce single-rate operation`	N/A	`Integer`	1
			`Bit`	2
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + 1 + 1
			`Bit`	length(data) + 2 + 2
		`true` (multitasking)	`Integer`	length(data) + 1 + 2
			`Bit`	length(data) + 2 + 4
`Normal raised cosine` or `Root raised cosine`	`Enforce single-rate operation`	N/A	`Integer`	**Filter span (in symbols)**
			`Bit`	2* **Filter span (in symbols)**
	`Allow multirate processing`	`false` (single tasking)	`Integer`	length(data) + **Filter span (in symbols)** + 1

Pulse Shape	Rate Options	Treat Each Discrete Rate as a Separate Task?	Input/Output Data (*)	End-to-End Delay Incurred by an OQPSK Modulator-Demodulator Block Pair (in samples)
			`Bit`	length(data) + 2***Filter span (in symbols)** + 2
		`true` (multitasking)	`Integer`	2*length(data) + **Filter span (in symbols)** + 2
			`Bit`	2*length(data) + 2***Filter span (in symbols)** + 4

(*) The data type parameter is **Input type** for modulation and **Output type** for demodulation.

Pulse Shaping Filter

The OQPSK modulation scheme requires oversampling of two or greater in order to delay (or offset) the quadrature channel by 90 degrees. This oversampling is achieved through interpolation filtering implemented by pulse shaping.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

OQPSK Demodulator Baseband | QPSK Modulator Baseband

Objects

`comm.OQPSKModulator`

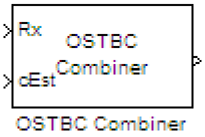
Topics

Phase Modulation

Introduced before R2006a

OSTBC Combiner

Combine inputs for received signals and channel estimate according to orthogonal space-time block code (OSTBC)



Library

MIMO

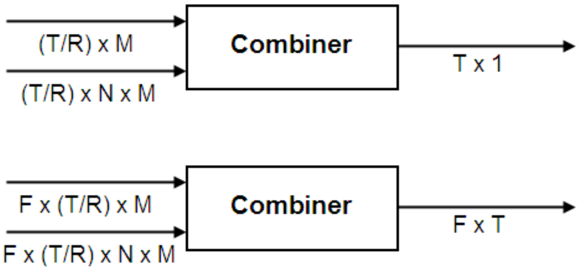
Description

The OSTBC Combiner block combines the input signal (from all of the receive antennas) and the channel estimate signal to extract the soft information of the symbols that were encoded using an OSTBC. The input channel estimate may not be constant during each codeword block transmission and the combining algorithm uses only the estimate for the first symbol period per codeword block. A symbol demodulator or decoder would follow the Combiner block in a MIMO communications system.

The block conducts the combining operation for each symbol independently. The combining algorithm depends on the structure of the OSTBC. For more information, see the OSTBC Combining Algorithms on page 5-630 section of this help page.

Dimension

Along with the time and spatial domains for OSTBC transmission, the block supports an optional dimension, over which the combining calculation is independent. This dimension can be thought of as the frequency domain for OFDM-based applications. The following illustration indicates the supported dimensions for inputs and output of the OSTBC Combiner block.



The following table describes each variable for the block.

Variable	Description
F	The additional dimension; typically the frequency dimension. The combining calculation is independent of this dimension.

Variable	Description
N	Number of transmit antennas.
M	Number of receive antennas.
T	Output symbol sequence length in time domain.
R	Symbol rate of the code.

Note On the two inputs, T/R is the symbol sequence length in the time domain.

F can be any positive integers. *M* can be 1 through 8, indicated by the **Number of receive antennas** parameter. *N* can be 2, 3 or 4, indicated by the **Number of transmit antennas** parameter. The time domain length T/R must be a multiple of the codeword block length (2 for Alamouti; 4 for all other OSTBC). For $N = 2$, T/R must be a multiple of 2. When $N > 2$, T/R must be a multiple of 4. *R* defaults to 1 for 2 antennas. *R* can be either $\frac{3}{4}$ or $\frac{1}{2}$ for more than 2 antennas.

The supported dimensions for the block depend upon the values of *F* and *M*. For one receive antenna ($M = 1$), the received signal input must be a column vector or a full 2–D matrix, depending on the value for *F*. The corresponding channel estimate input must be a full 2-D or 3-D matrix.

For more than one receive antenna ($M > 1$), the received signal input must be a full 2-D or 3-D matrix, depending on the value for *F*. Correspondingly, the channel estimate input must be a 3-D or 4-D matrix, depending on the value for *F*.

To understand the block's dimension propagation, refer to the following table.

	Input 1 (Received Signal)	Input 2 (Channel Estimate)	Output
F = 1 and M = 1	Column vector	2-D	Column vector
F = 1 and M > 1	2-D	3-D	Column vector
F > 1 and M = 1	2-D	3-D	2-D
F > 1 and M > 1	3-D	4-D	2-D

Data Type

For information about the data types each block port supports, see the “Supported Data Type” on page 5-633 table on this page. The output signal inherits the data type from the inputs. The block supports different fixed-point properties for the two inputs. For fixed-point signals, the output word length and fractional length depend on the block’s mask parameter settings. See Fixed-Point Signals for more information about fixed-point data propagation of this block.

Frames

The output inherits the frameness of the received signal input. For either column vector or full 2-D matrix input signal, the input can be either frame-based or sample-based. A 3–D or 4–D matrix input signal must have sample-based input.

OSTBC Combining Algorithms

The OSTBC Combiner block supports five different OSTBC combining computation algorithms. Depending on the selection for **Rate** and **Number of transmit antennas**, you can select one of the algorithms shown in the following table.

Transmit Antenna	Rate	Computational Algorithm per Codeword Block Length
2	1	$\begin{pmatrix} \widehat{s}_1 \\ \widehat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} \end{pmatrix}.$
3	1/2	$\begin{pmatrix} \widehat{s}_1 \\ \widehat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j} r_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
3	3/4	$\begin{pmatrix} \widehat{s}_1 \\ \widehat{s}_2 \\ \widehat{s}_3 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$
4	1/2	$\begin{pmatrix} \widehat{s}_1 \\ \widehat{s}_2 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} + h^*_{3,j3,j} r_{3,j} + h_{4,j} r^*_{4,j} \\ h^*_{2,j} r_{1,j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \end{pmatrix}.$
4	3/4	$\begin{pmatrix} \widehat{s}_1 \\ \widehat{s}_2 \\ \widehat{s}_3 \end{pmatrix} = \frac{1}{\lVert H \rVert^2} \sum_{j=1}^{M} \begin{pmatrix} h^*_{1,j} r_{1,j} + h_{2,j} r^*_{2,j} - h_{3,j} r^*_{3,j} - h^*_{4,j} r_{4j} \\ h^*_{2,j} r_{1j} - h_{1,j} r^*_{2,j} + h^*_{4,j} r_{3,j} - h_{3,j} r^*_{4,j} \\ h^*_{3,j} r_{1j} + h^*_{4,j} r_{2,j} + h_{1,j} r^*_{3,j} + h_{2,j} r^*_{4,j} \end{pmatrix}.$

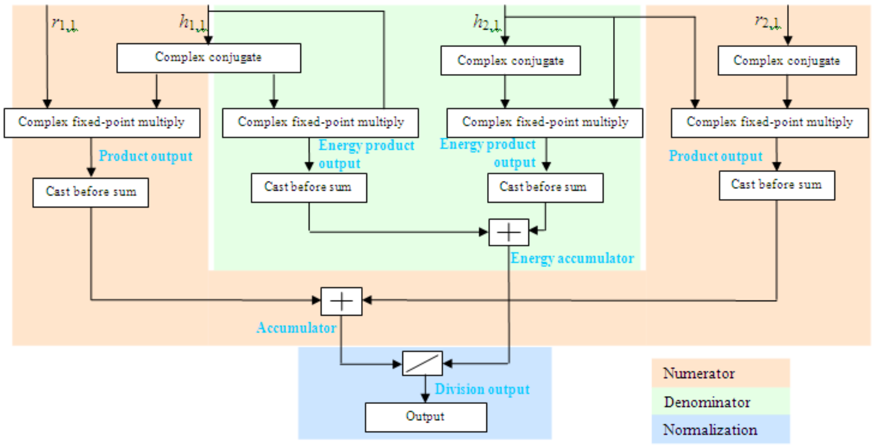
$\widehat{s}_k$ represents the estimated *k*th symbol in the OSTBC codeword matrix. h_{ij} represents the estimate for the channel from the *i*th transmit antenna and the *j*th receive antenna. The values of *i* and *j* can range from 1 to *N* (the number of transmit antennas) and to *M* (the number of receive antennas) respectively. r_{lj} represents the *l*th symbol at the *j*th receive antenna per codeword block. The value of *l* can range from 1 to the codeword block length. $\lVert H \rVert^2$ represents the summation of channel power per link, i.e., $\lVert H \rVert^2 = \sum_{i=1}^{N} \sum_{j=1}^{M} \lVert h_{ij} \rVert^2$

Fixed-Point Signals

Use the following formula for $\widehat{s}_1$ for Alamouti code with 1 receive antenna to highlight the data types used for fixed-point signals.

$$\widehat{s}_1 = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,} r^*_{2,1}}{\lVert H \rVert^2} = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,} r^*_{2,1}}{h_{1,1} h^*_{1,1,} + h_{2,1,} h^*_{2,1}}$$

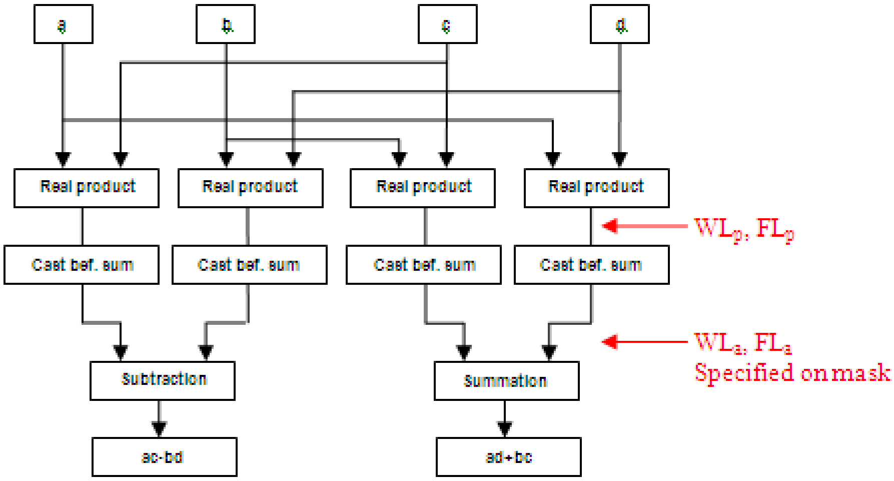
In this equation, the data types for **Product output** and **Accumulator** correspond to the product and summation in the numerator. Similarly, the types for **Energy product output** and **Energy accumulator** correspond to the product and summation in the denominator.



Signal Flow Diagram for s_1 Combining Calculation of Alamouti Code with One Receive Antenna

The following formula shows the data types used within the OSTBC Combiner block for fixed-point signals for more than one receive antenna for Alamouti code, where *M* represents the number of receive antennas.

$$\widehat{s}_1 = \frac{h^*_{1,1,}\ r_{1,1} + h_{2,1,} r^*_{2,1} + h^*_{1,2} r_{1,2} + h_{2,2} r^*_{2,2} + \ldots + h^*_{1,M,}\ r_{1,M} + h_{2,M} r^*_{2,M}}{h_{1,1} h^*_{1,1} +\ h_{2,1} h^*_{2,1} + h_{1,2} h^*_{1,2} +\ h_{2,2} h^*_{2,2} + \ldots + h_{1,M} h^*_{1,M} +\ h_{2,M,} h^*_{2,M}}$$



Signal Flow Diagram for Complex Multiply of *a* + *ib* and *c* + *id*

For Binary point scaling, you cannot specify WL_p and FL_p. Instead, the blocks determine these values implicitly from WL_a and FL_a

The Internal Rule for **Product output** and **Energy product output** are:

- When you select `Inherit via internal rule`, the internal rule determines WL_p and FL_p. Therefore, $WL_a = WL_p + 1$ and $FL_a = FL_p$
- For `Binary point scaling`, you specify WL_a and FL_a. Therefore, $WL_p = WL_a - 1$ and $FL_a = FL_p$.

For information on how the Internal Rule applies to the **Accumulator** and **Energy Accumulator**, see Inherit via Internal Rule.

Parameters

Number of transmit antennas

Sets the number of transmit antennas. The block supports 2, 3, or 4 transmit antennas. This value defaults to 2.

Rate

Sets the symbol rate of the code. You can specify either 3/4 or 1/2. This field only appears when you use more than 2 transmit antennas. This field defaults to $\frac{3}{4}$ for more than 2 transmit antennas. For 2 transmit antennas, there is no rate option and the implicit (default) rate defaults to 1.

Number of receive antennas

The number of antennas the block uses to receive signal streams. The block supports from 1 to 8 receive antennas. This value defaults to 1.

Rounding mode

Sets the rounding mode for fixed-point calculations. The block uses the rounding mode if a value cannot be represented exactly by the specified data type and scaling. When this occurs, the value is rounded to a representable number. For more information refer to Rounding (Fixed-Point Designer).

Saturate on integer overflow

Sets the overflow mode for fixed-point calculations. Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result. For more information refer to Precision and Range.

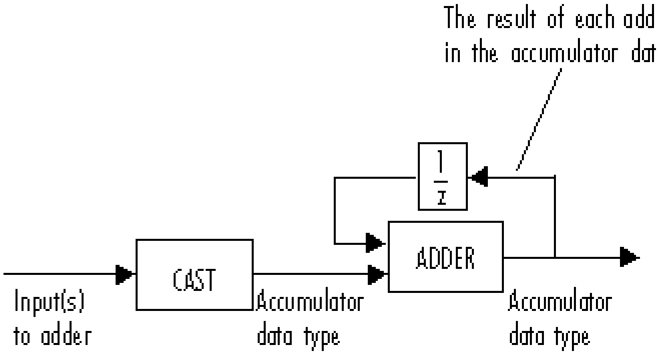
Product Output

Complex product in the numerator for the diversity combining. For more information refer to the Fixed-Point Signals section of this help page.

Accumulator

Summation in the numerator for the diversity combining.

Fixed-point Communications Toolbox blocks that must hold summation results for further calculation usually allow you to specify the data type and scaling of the accumulator. Most such blocks cast to the accumulator data type prior to summation:



Use the **Accumulator—Mode** parameter to specify how you would like to designate the accumulator word and fraction lengths:

- When you select `Inherit via internal rule`, the accumulator output word and fraction lengths are automatically calculated for you. Refer to Inherit via Internal Rule for more information.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Same as input`, these characteristics match those of the first input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. The bias of all signals in DSP System Toolbox software is zero.

Energy product output

Complex product in the denominator for calculating total energy in the MIMO channel .

Energy accumulator

Summation in the denominator for calculating total energy in the MIMO channel.

Division output

Normalized diversity combining by total energy in the MIMO channel.

Supported Data Type

Port	Supported Data Types
Rx	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
cEst	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed Fixed-point

Examples

For an example of this block in use, see OSTBC Over 3x2 Rayleigh Fading Channel. The model shows the use of a rate ¾ OSTBC for 3 transmit and 2 receive antennas with BPSK modulation using independent fading links and AWGN.

You can also see the block in the Concatenated OSTBC with TCM example by typing `commtcmostbc` at the MATLAB command line.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

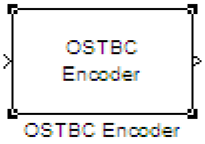
Blocks

OSTBC Encoder

Introduced in R2009a

OSTBC Encoder

Encode input message using orthogonal space-time block code (OSTBC)



Library

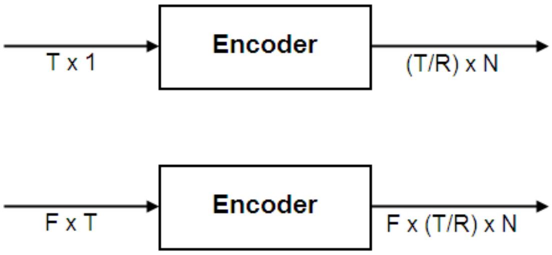
MIMO

Description

The OSTBC Encoder block encodes an input symbol sequence using orthogonal space-time block code (OSTBC). The block maps the input symbols block-wise and concatenates the output codeword matrices in the time domain. For more information, see the OSTBC Encoding Algorithms on page 5-636 section of this help page.

Dimension

The block supports time and spatial domains for OSTBC transmission. It also supports an optional dimension, over which the encoding calculation is independent. This dimension can be thought of as the frequency domain. The following illustration indicates the supported dimensions for the inputs and output of the OSTBC Encoder block.



The following table describes the variables.

Variable	Description
F	The additional dimension; typically the frequency domain. The encoding does not depend on this dimension.
T	Input symbol sequence length for the time domain.
R	Symbol rate of the code.
N	Number of transmit antennas.

Note On the output, T/R is the symbol sequence length in time domain.

F can be any positive integer. *N* can be 2, 3 or 4, indicated by **Number of transmit antennas**. For *N* = 2, *R* must be 1. For *N* = 3 or 4, *R* can be 3/4 or 1/2, indicated by **Rate**. The time domain length *T* must be a multiple of the number of symbols in each codeword matrix. Specifically, for *N* = 2 or *R* = 1/2, *T* must be a multiple of 2 and when *R* = 3/4, *T* must be a multiple of 3.

To understand the block's dimension propagation, refer to the following table.

Dimension	Input	Output
F = 1	Column vector	2-D
F > 1	2-D	3-D

Data Type

For information about the data types each block port supports, see the "Supported Data Type" on page 5-637 table on this page. The output signal inherits the data type from the input signal. For fixed-point signals, the complex conjugation may cause overflows which the fixed-point parameter **Saturate on integer overflow** must handle.

Frames

The output signal inherits frame type from the input signal. A column vector input requires either frame-based or sample-based input; otherwise, the input must be sample-based.

OSTBC Encoding Algorithms

The OSTBC Encoder block supports five different OSTBC encoding algorithms. Depending on the selection for **Rate** and **Number of transmit antennas**, the block implements one of the algorithms in the following table:

Transmit Antenna	Rate	OSTBC Codeword Matrix
2	1	$\begin{pmatrix} s_1 & s_2 \\ -s_2^* & s_1^* \end{pmatrix}$
3	1/2	$\begin{pmatrix} s_1 & s_2 & 0 \\ -s_2^* & s_1^* & 0 \\ 0 & 0 & s_1 \\ 0 & 0 & -s_2^* \end{pmatrix}$
3	3/4	$\begin{pmatrix} s_1 & s_2 & s_3 \\ -s_2^* & s_1^* & 0 \\ s_3^* & 0 & -s_1^* \\ 0 & s_3^* & -s_2^* \end{pmatrix}$

Transmit Antenna	Rate	OSTBC Codeword Matrix
4	1/2	$\begin{pmatrix} s_1 & s_2 & 0 & 0 \\ -s_2^* & s_1^* & 0 & 0 \\ 0 & 0 & s_1 & s_2 \\ 0 & 0 & -s_2^* & s_1^* \end{pmatrix}$
4	3/4	$\begin{pmatrix} s_1 & s_2 & s_3 & 0 \\ -s_2^* & s_1^* & 0 & s_3 \\ s_3^* & 0 & -s_1^* & s_2 \\ 0 & s_3^* & -s_2^* & -s_1 \end{pmatrix}$

In each matrix, its (l, i) entry indicates the symbol transmitted from the ith antenna in the lth time slot of the block. The value of i can range from 1 to N (the number of transmit antennas). The value of l can range from 1 to the codeword block length.

Parameters

Number of transmit antennas

Sets the number of antennas at the transmitter side. The block supports 2, 3, or 4 transmit antennas. The value defaults to 2.

Rate

Sets the symbol rate of the code. You can specify either 3/4 or 1/2. This field only appears when using more than 2 transmit antennas. This field defaults to $\frac{3}{4}$ for more than 2 transmit antennas. For 2 transmit antennas, there is no rate option and the rate defaults to 1.

Saturate on integer overflow

Sets the overflow mode for fixed-point calculations. Use this parameter to specify the method to be used if the magnitude of a fixed-point calculation result does not fit into the range of the data type and scaling that stores the result. For more information refer to “Precision and Range”.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed Fixed-point

Examples

For an example of this block in use, see “OSTBC Over 3x2 Rayleigh Fading Channel” . The model shows the use of a rate ¾ OSTBC for 3 transmit and 2 receive antennas with BPSK modulation using independent fading links and AWGN

You can also see the block in the Concatenated OSTBC with TCM example by typing `commtcmostbc` at the MATLAB command line.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

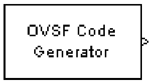
Blocks

OSTBC Combiner

Introduced in R2009a

OVSF Code Generator

Generate orthogonal variable spreading factor (OVSF) code from set of orthogonal codes



Library

Spreading Codes

Description

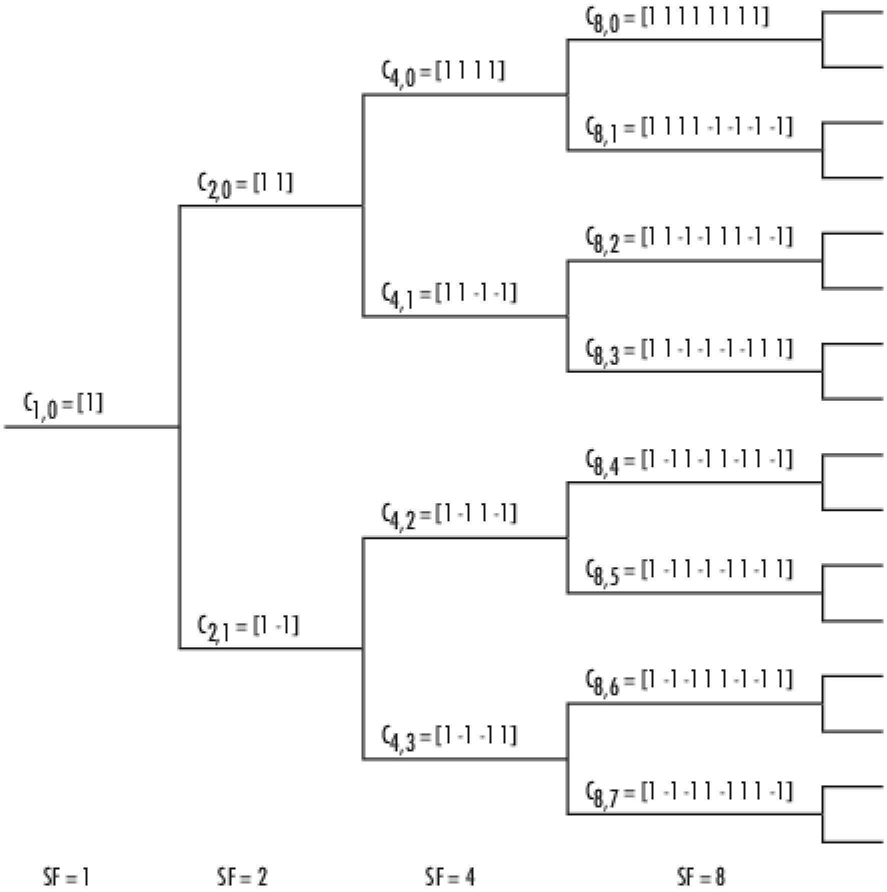
The OVSF Code Generator block generates an OVSF code from a set of orthogonal codes. OVSF codes were first introduced for 3G communication systems. OVSF codes are primarily used to preserve orthogonality between different channels in a communication system.

OVSF codes are defined as the rows of an N-by-N matrix, C_N, which is defined recursively as follows. First, define C_1 = [1]. Next, assume that C_N is defined and let $C_N(k)$ denote the kth row of C_N. Define C_{2N} by

$$C_{2N} = \begin{bmatrix} C_N(0) & C_N(0) \\ C_N(0) & -C_N(0) \\ C_N(1) & C_N(1) \\ C_N(1) & -C_N(1) \\ \ldots & \ldots \\ C_N(N-1) & C_N(N-1) \\ C_N(N-1) & -C_N(N-1) \end{bmatrix}$$

Note that C_N is only defined for N a power of 2. It follows by induction that the rows of C_N are orthogonal.

The OVSF codes can also be defined recursively by a tree structure, as shown in the following figure.


$C_{8,0}$ = [1 1 1 1 1 1 1 1]
$C_{4,0}$ = [1 1 1 1]
$C_{8,1}$ = [1 1 1 1 -1 -1 -1 -1]
$C_{2,0}$ = [1 1]
$C_{8,2}$ = [1 1 -1 -1 1 1 -1 -1]
$C_{4,1}$ = [1 1 -1 -1]
$C_{8,3}$ = [1 1 -1 -1 -1 -1 1 1]
$C_{1,0}$ = [1]
$C_{8,4}$ = [1 -1 1 -1 1 -1 1 -1]
$C_{4,2}$ = [1 -1 1 -1]
$C_{8,5}$ = [1 -1 1 -1 -1 1 -1 1]
$C_{2,1}$ = [1 -1]
$C_{8,6}$ = [1 -1 -1 1 1 -1 -1 1]
$C_{4,3}$ = [1 -1 -1 1]
$C_{8,7}$ = [1 -1 -1 1 -1 1 1 -1]
SF = 1
SF = 2
SF = 4
SF = 8

If [C] is a code length 2^r at depth r in the tree, where the root has depth 0, the two branches leading out of C are labeled by the sequences [C C] and [C -C], which have length 2^{r+1}. The codes at depth r in the tree are the rows of the matrix C_N, where N = 2^r.

Note that two OVSF codes are orthogonal if and only if neither code lies on the path from the other code to the root. Since codes assigned to different users in the same cell must be orthogonal, this restricts the number of available codes for a given cell. For example, if the code C_{41} in the tree is assigned to a user, the codes C_{10}, C_{20}, C_{82}, C_{83}, and so on, cannot be assigned to any other user in the same cell.

Block Parameters

You specify the code the OVSF Code Generator block outputs by two parameters in the block's dialog: the **Spreading factor**, which is the length of the code, and the **Code index**, which must be an integer in the range [0, 1, ... , N - 1], where N is the spreading factor. If the code appears at depth r in the preceding tree, the **Spreading factor** is 2^r. The **Code index** specifies how far down the column of the tree at depth r the code appears, counting from 0 to *N* - 1. For $C_{N,\ k}$ in the preceding diagram, *N* is the **Spreading factor** and *k* is the **Code index**.

You can recover the code from the **Spreading factor** and the **Code index** as follows. Convert the **Code index** to the corresponding binary number, and then add 0s to the left, if necessary, so that the resulting binary sequence $x_1\ x_2 \ldots x_r$ has length *r*, where *r* is the logarithm base 2 of the **Spreading factor**. This sequence describes the path from the root to the code. The path takes the upper branch from the code at depth *i* if $x_i = 0$, and the lower branch if $x_i = 1$.

To reconstruct the code, recursively define a sequence of codes C_i for as follows. Let C_0 be the root [1]. Assuming that C_i has been defined, for *i* < *r*, define C_{i+1} by

$$C_{i+1} = \begin{cases} C_i C_i & \text{if } x_i = 0 \\ C_i(-C_i) & \text{if } x_i = 1 \end{cases}$$

The code C_N has the specified **Spreading factor** and **Code index**.

For example, to find the code with **Spreading factor** 16 and **Code index** 6, do the following:

1. Convert 6 to the binary number 110.
2. Add one 0 to the left to obtain 0110, which has length 4 = $\log_2$ 16.
3. Construct the sequences C_i according to the following table.

i	x_i	C_i
0		C_0 = [1]
1	0	$C_1 = C_0\ C_0$ = [1] [1]
2	1	$C_2 = C_1\ {-C_1}$ = [1 1] [-1 -1]
3	1	$C_3 = C_2\ {-C_2}$ = [1 1 -1 -1] [-1 -1 1 1]
4	0	$C_4 = C_3\ C_3$ = [1 1 -1 -1 -1 -1 1 1] [1 1 -1 -1 -1 -1 1 1]

The code C_4 has **Spreading factor** 16 and **Code index** 6.

Parameters

Spreading factor

Positive integer that is a power of 2, specifying the length of the code.

Code index

Integer in the range [0, 1, ... , N - 1] specifying the code, where N is the **Spreading factor**.

Sample time

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-642.

Samples per frame

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-642.

Output data type

The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the OVSF Code Generator block version available before R2015b.

Existing models automatically update to load the OVSF Code Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Does not support integer only code generation.

See Also

Blocks

Hadamard Code Generator | Walsh Code Generator

Introduced before R2006a

Phase/Frequency Offset

Apply phase and frequency offsets to complex baseband signal

Library: Communications Toolbox / RF Impairments Correction
Communications Toolbox / RF Impairments

Description

The Phase/Frequency Offset block applies phase and frequency offsets to a complex signal.

This icon shows the block with all ports enabled.

In
Phase/
Frequency
Frq
Offset

Ports

Input

`In` — Complex signal

scalar | vector | matrix

Complex signal, specified as a scalar, vector, or matrix. The port is unnamed until you enable the **Frequency offset from port** parameter.

Data Types: `double` | `single`
Complex Number Support: Yes

`Frq` — Frequency offset

scalar | vector | matrix

Frequency offset, specified as a scalar, a vector with the same number of rows or columns as the input signal, or a matrix with the same dimensions as the input signal. For more information, see "Interdependent Parameter-Port Dimensions" on page 5-644.

Dependencies

To enable this port, select the **Frequency offset from port** parameter.

Data Types: `double` | `single`

Output

`Out1` — Output signal

scalar | vector | matrix

Output signal, returned as a scalar, vector, or matrix. This output is the same dimension and data type as the input signal.

Parameters

`Phase offset (deg)` — Phase offset

`0` (default) | scalar | vector | matrix

Phase offset in degrees, specified as a scalar, vector, or matrix.

If **Phase offset (deg)** and **Frequency offset (Hz)** are both nonscalar, they must be the same size.

Tunable: Yes

`Frequency offset from port` — Option to add port to set frequency offset

`off` (default) | on

Select this parameter to add the **Frq** port.

- When you select this parameter, the **Frq** port specifies the frequency offset.
- When you clear this parameter, the **Frequency offset (Hz)** parameter specifies the frequency offset.

`Frequency offset (Hz)` — Frequency offset

`0` (default) | scalar | vector | matrix

Frequency offset in hertz, specified as a scalar, a vector with the same number of rows or columns as the input signal, or a matrix with the same dimensions as the input signal. For more information, see “Interdependent Parameter-Port Dimensions” on page 5-644.

If **Phase offset (deg)** and **Frequency offset (Hz)** are both nonscalar, they must be the same size.

Tunable: Yes

Dependencies

To enable this port, clear the **Frequency offset from port** parameter.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Interdependent Parameter-Port Dimensions

This table outlines the interdependency of parameter-to-port dimensions.

Number of Dimensions	Data I/O Dimension	Frame Size	Number of Channels	Frequency/Phase Offset Parameter Dimension	Frequency Offset Input Port Dimension
Any	Scalar	1	1	Scalar	Scalar

Number of Dimensions	Data I/O Dimension	Frame Size	Number of Channels	Frequency/Phase Offset Parameter Dimension	Frequency Offset Input Port Dimension
2	*M*-by-1	*M*	1	*M*-by-1 1-by-*M* 1-by-1	*M* *M*-by-1 1 1-by-1
2	1-by-*N*	1	*N*	*N*-by-1 1-by-*N* 1-by-1	*N* 1-by-*N* 1 1-by-1
2	*M*-by-*N*	*M*	*N*	*M*-by-*N* *N*-by-1 1-by-*N* *M*-by-1 1-by-*M* 1-by-1	*M*-by-*N* *N* 1-by-*N* 1 1-by-1 *M* *M*-by-1

For example:

- When you specify a scalar offset parameter, the block applies the same offset to all elements of the input signal
- When you specify a 2-by-1 offset parameter for a 2-by-3 input signal (one offset value per sample), the block applies the same sample offset across the three channels.
- When you specify a 1-by-3 offset parameter for a 2-by-3 input signal (one offset value per channel), the same channel offset is applied across the two samples of a channel.
- When you specify a 2-by-3 offset parameter for a 2-by-3 input signal (one offset value per sample for each channel), the offsets are applied element-wise to the input signal.

Algorithms

If the input signal is $u(t)$, then the output signal is

$$y(t) = u(t)\cdot\left(\cos\left(2\pi\int_0^t f(\tau)d\tau + \varphi(t)\right) + j\sin\left(2\pi\int_0^t f(\tau)d\tau + \varphi(t)\right)\right),$$

where $f(t)$ is the frequency offset, and $\varphi(t)$ is the phase offset.

The discrete-time output is given by

$y(0) = u(0)(\cos(\varphi(0)) + j\sin(\varphi(0)))$ and

$$y(i) = u(i)\left(\cos\left(2\pi\sum_{n=0}^{i-1} f(n)\Delta t + \varphi(i)\right) + j\sin\left(2\pi\sum_{n=0}^{i-1} f(n)\Delta t + \varphi(i)\right)\right),$$

where $i > 0$, and Δt is the sample time.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The **Frequency offset (Hz)** and **Phase offset (deg)** parameters are tunable in Normal mode, Accelerator mode, and Rapid Accelerator mode. If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune these parameters without recompiling the model. For more information, see Tunable Parameters (Simulink).

See Also

Blocks

Free Space Path Loss | I/Q Imbalance | Memoryless Nonlinearity | Phase Noise | Receiver Thermal Noise

Introduced before R2006a

Phase-Locked Loop

Implement phase-locked loop to recover phase of input signal

Filt
PLL
PD
VCO

Library

Components sublibrary of Synchronization

Description

The Phase-Locked Loop (PLL) block is a feedback control system that automatically adjusts the phase of a locally generated signal to match the phase of an input signal. This block is most appropriate when the input is a narrowband signal.

This PLL has these three components:

- A multiplier used as a phase detector.
- A filter. You specify the filter transfer function using the **Lowpass filter numerator** and **Lowpass filter denominator** parameters. Each is a vector that gives the respective polynomial's coefficients in order of descending powers of *s*.

  To design a filter, you can use functions such as `butter`, `cheby1`, and `cheby2` in Signal Processing Toolbox software. The default filter is a Chebyshev type II filter whose transfer function arises from the command below.

  ```
  [num, den] = cheby2(3,40,100,'s')
  ```
- A voltage-controlled oscillator (VCO). You specify characteristics of the VCO using the **VCO quiescent frequency**, **VCO initial phase**, and **VCO output amplitude** parameters.

This block accepts a sample-based scalar input signal. The input signal represents the received signal. The three output ports produce:

- The output of the filter
- The output of the phase detector
- The output of the VCO

Parameters

Lowpass filter numerator

The numerator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

Lowpass filter denominator

The denominator of the lowpass filter transfer function, represented as a vector that lists the coefficients in order of descending powers of *s*.

VCO input sensitivity (Hz/V)

This value scales the input to the VCO and, consequently, the shift from the **VCO quiescent frequency** value. The units of **VCO input sensitivity** are Hertz per volt.

VCO quiescent frequency (Hz)

The frequency of the VCO signal when the voltage applied to it is zero. This should match the carrier frequency of the input signal.

VCO initial phase (rad)

The initial phase of the VCO signal.

VCO output amplitude

The amplitude of the VCO signal.

References

For more information about phase-locked loops, see the works listed in “Selected Bibliography for Synchronization”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Baseband PLL | Charge Pump PLL | Linearized Baseband PLL

Introduced before R2006a

Phase Noise

Apply receiver phase noise to complex baseband signal
Library: Communications Toolbox / RF Impairments

Description

The Phase Noise block adds phase noise to a complex signal. This block emulates impairments introduced by the local oscillator of a wireless communication transmitter or receiver. The block generates filtered phase noise according to the specified spectral mask and adds it to the input signal. For a description of the phase noise modeling, see “Algorithms” on page 5-651.

Ports

Input

`In` — Input signal
complex column vector

Input signal, specified as an N_S-by-1 vector of complex values. N_S represents the number of samples in the input signal.

Data Types: `double`
Complex Number Support: Yes

Output

`Out` — Output signal
complex column vector

Output signal, returned as an N_S-by-1 vector of complex values. N_S equals the number of samples in the input signal.

Data Types: `double`
Complex Number Support: Yes

Parameters

`Phase noise level (dBc/Hz)` — Phase noise level
`[-60 -80]` (default) | vector of negative scalars

Phase noise level in decibels relative to carrier per hertz (dBc/Hz), specified as a vector of negative scalars. The **Phase noise level (dBc/Hz)** and **Frequency offset (Hz)** parameters must have the same length.

Tunable: Yes

`Frequency offset (Hz)` — Frequency offset

`[20 200]` (default) | vector of positive increasing values

Frequency offset in Hz, specified as a vector of positive increasing values. The **Phase noise level (dBc/Hz)** and **Frequency offset (Hz)** parameters must have the same length.

Tunable: Yes

Data Types: `double`

`Sample rate (Hz)` — Sample rate

`1024` (default) | positive scalar

Sample rate in samples per second, specified as a positive scalar. To avoid aliasing, the sample rate must be greater than twice the largest value specified by **Frequency offset (Hz)**.

Tunable: Yes

Data Types: `double`

`Initial seed` — Initial seed of noise generator

`2137` (default) | positive scalar

Initial seed of noise generator, specified as a positive scalar.

This block uses the Random Source block to generate noise. The block generates random numbers using the Ziggurat method (V5 RANDN algorithm). Every time you rerun the simulation, the block reuses the same initial seed. That way, the block outputs the same signal each time you run a simulation.

Tunable: Yes

Data Types: `double`

`View Filter Response` — Display magnitude response of filter

button

Display magnitude response of filter defined by the Phase Noise block. The block uses the `fvtool` function to display the magnitude response.

`Simulate using` — Specify type of simulation to run

`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` -– Simulate model using the MATLAB interpreter. This option shortens startup time speed, but the speed of the subsequent simulations is slower than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` -– Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.

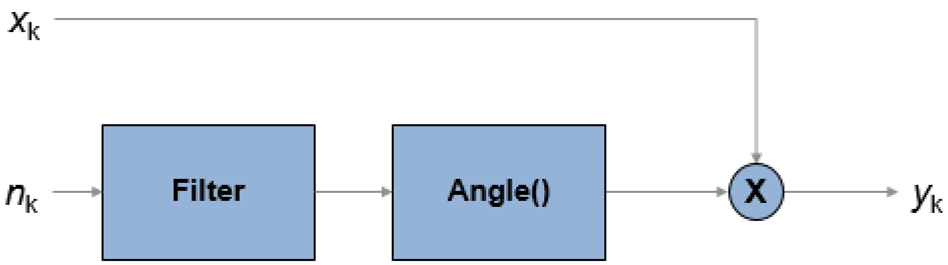
Block Characteristics

Data Types	`double` | `single`

Multidimensional Signals	no
Variable-Size Signals	no

Algorithms

The output signal, y_k, is related to input sequence x_k by $y_k = x_k e^{j\varphi_k}$, where φ_k is the phase noise. The phase noise is filtered Gaussian noise such that $\varphi_k = f(n_k)$, where n_k is the noise sequence and f represents a filtering operation.



To model the phase noise, define the power spectrum density (PSD) mask characteristic by specifying scalar or vector values for the frequency offset and phase noise level.

- For a scalar frequency offset and phase noise level specification, an IIR digital filter computes the spectrum mask. The spectrum mask has a 1/*f* characteristic that passes through the specified point.
- For a vector frequency offset and phase noise level specification, an FIR filter computes the spectrum mask. The spectrum mask is interpolated across log10(*f*). It is flat from DC to the lowest frequency offset, and from the highest frequency offset to half the sample rate.

IIR Digital Filter

For the IIR digital filter, the numerator coefficient is

$$\lambda = \sqrt{2\pi f_{offset} 10^{L/10}},$$

where f_{offset} is the frequency offset in Hz and L is the phase noise level in dBc/Hz. The denominator coefficients, γ_i, are recursively determined as

$$\gamma_i = (i - 2.5)\frac{\gamma_{i-1}}{i-1},$$

where $\gamma_1 = 1$, $i = \{1, 2,..., N_t\}$, and N_t is the number of filter coefficients. N_t is a power of 2, from 2^7 to 2^{19}. The value of N_t grows as the phase noise offset decreases towards 0 Hz.

FIR Filter

For the FIR filter, the phase noise level is determined through log10(*f*) interpolation for frequency offsets over the range [*df*, f_s / 2], where *df* is the frequency resolution and f_s is the sample rate. The phase noise is flat from 0 Hz to the smallest frequency offset, and from the largest frequency offset to f_s / 2. The frequency resolution is equal to $\frac{f_s}{2}\left(\frac{1}{N_t}\right)$, where N_t is the number of coefficients, and is a power of 2 less than or equal to 2^{16}. If $N_t < 2^8$, a time domain FIR filter is used. Otherwise, a frequency domain FIR filter is used.

The algorithm increases N_t until these conditions are met:

- The frequency resolution is less than the minimum value of the frequency offset vector.
- The frequency resolution is less than the minimum difference between two consecutive frequencies in the frequency offset vector.
- The maximum number of FIR filter taps is 2^{16}.

References

[1] Kasdin, N. J., "Discrete Simulation of Colored Noise and Stochastic Processes and 1/(f^alpha); Power Law Noise Generation." *The Proceedings of the IEEE*. Vol. 83, No. 5, May, 1995, pp 802–827.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Phase/Frequency Offset

Functions

`plotPhaseNoiseFilter`

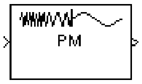
Objects

`comm.PhaseNoise`

Introduced before R2006a

PM Demodulator Passband

Demodulate PM-modulated data



Library

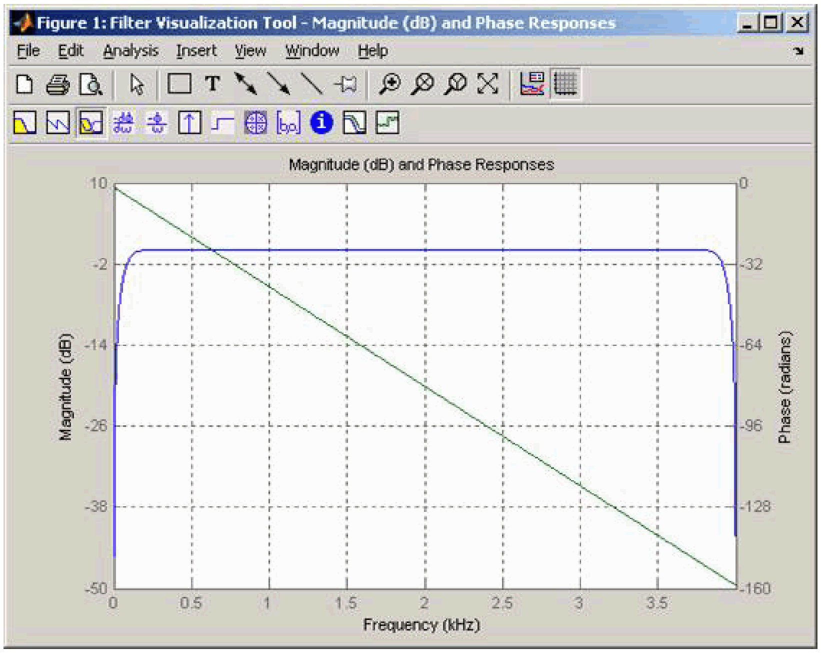
Analog Passband Modulation, in Modulation

Description

The PM Demodulator Passband block demodulates a signal that was modulated using phase modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

For best results, use a carrier frequency which is estimated to be larger than 10% of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.



Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the input signal's sample rate (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier in radians.

Phase deviation (rad)

The phase deviation of the carrier frequency in radians. Sometimes it is referred to as the "variation" in the phase.

Hilbert transform filter order

The length of the FIR filter used to compute the Hilbert transform.

Pair Block

PM Modulator Passband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

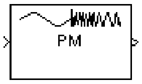
Blocks

PM Modulator Passband

Introduced before R2006a

PM Modulator Passband

Modulate using phase modulation



Library

Analog Passband Modulation, in Modulation

Description

The PM Modulator Passband block modulates using phase modulation. The output is a passband representation of the modulated signal. The output signal's frequency varies with the input signal's amplitude. Both the input and output signals are real scalar signals.

If the input is $u(t)$ as a function of time t, then the output is

$$\cos(2\pi f_c t + K_c u(t) + \theta)$$

where

- f_c represents the **Carrier frequency** parameter
- θ represents the **Initial phase** parameter
- K_c represents the **Phase deviation** parameter

An appropriate **Carrier frequency** value is generally much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The initial phase of the carrier in radians.

Phase deviation (rad)

The phase deviation of the carrier frequency in radians. This is sometimes referred to as the "variation" in the phase.

Pair Block

PM Demodulator Passband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

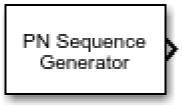
PM Demodulator Passband

Introduced before R2006a

PN Sequence Generator

Generate pseudonoise sequence

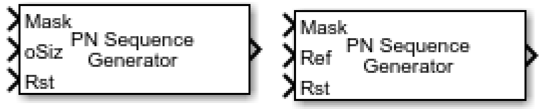
Library: Communications Toolbox / Comm Sources / Sequence Generators
Communications Toolbox HDL Support / Comm Sources



Description

The PN Sequence Generator block generates a sequence of pseudorandom binary numbers using a linear-feedback shift register (LFSR). Pseudonoise sequences are typically used for pseudorandom scrambling, and in direct-sequence spread-spectrum systems. For more information, see “More About” on page 5-662.

These icons shows the block with all ports enabled.



Ports

Input

`Mask` — Output mask
binary vector

Output mask to delay the PN sequence from initial time, specified as a binary vector with *N* elements. *N* is the degree of the generator polynomial.

Dependencies

To enable this port, set **Output mask source** to `Input port`.

Data Types: `double` | `uint8` | `ufix1`

`oSiz` — Output size
integer

Output size for variable-size output signals, specified as an integer. For information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Dependencies

To enable this port, select **Output variable-size signals** and set **Maximum output size source** to `Dialog parameter`.

Data Types: `double`

`Ref` — Reference input

column vector

Reference input, specified as a column vector that determines the maximum and current output sequence length. The **Ref** input must be a variable-size signal. For information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Dependencies

To enable this port, select **Output variable-size signals** and set **Maximum output size source** to `Inherit from reference input`.

Data Types: `double`

`Rst` — Reset sequence generator

0 | 1

Reset sequence generator, specified as `0` or `1`. For more information, see “Reset Behavior” on page 5-663.

Dependencies

To enable this port, select **Reset on nonzero input**.

Data Types: `Boolean`

Output

`Out` — Pseudorandom noise sequence

binary vector

PN sequence, returned as a binary vector.

Parameters

`Generator polynomial` — Generator polynomial

`'z^6 + z + 1'` (default) | polynomial character vector | binary row vector

Generator polynomial, specified as one of the following:

- A polynomial character vector that includes the number `1`.
- A binary row vector that represents the coefficients of the generator polynomial in order of descending power. The first and last entries must be `1`. The length of this vector is (N+1), where N is the degree of the generator polynomial.
- A vector containing the exponents of z for the nonzero terms of the polynomial in descending order of powers. The last entry must be `0`.

For more information, see “Character Representation of Polynomials”.

Example: `'z^8 + z^2 + 1'`, `[1 0 0 0 0 0 1 0 1]`, and `[8 2 0]` represent the same polynomial, $p(z) = z^8 + z^2 + 1$.

Data Types: `double` | `char`

`Initial states` — Initial shift register states

`[0 0 0 0 0 1]` (default) | binary row vector

Initial shift register states, specified as a binary row vector of length *N*, where *N* is the degree of the generator polynomial.

Note For the block to generate a nonzero sequence, the **Initial states** vector must contain at least one nonzero element.

Data Types: `double`

`Output mask source` — Output mask source

`Dialog parameter` (default) | `Input port`

Output mask source that indicates how the output mask information is given to the block, specified as:

- `Dialog parameter` to use the `Output mask vector (or scalar shift value)` parameter setting.
- `Input port` to add and use the `Mask` input port.

`Output mask vector (or scalar shift value)` — Output mask vector or scalar shift value

0 (default) | integer scalar | binary vector

Output mask vector or scalar shift value, specified as an integer scalar or binary row vector of length *N*, where *N* is the degree of the generator polynomial. This parameter determines the delay of the PN sequence from the initial time. For more information, see “Shifting PN Sequence Starting Point” on page 5-663.

Dependencies

To enable this parameter, set **Output mask source** to `Dialog parameter`.

Data Types: `double`

`Output variable-size signals` — Output variable-size signals

off (default) | on

Select this parameter to permit variable length output sequences during simulation. When set to `off`, fixed-length sequences are output. When set to `on`, variable-length sequences can be output. For information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

`Maximum output size source` — Maximum output size source

`Dialog parameter` (default) | `Inherit from reference port`

Maximum output size source, which indicates how the maximum sequence output size is specified.

- `Dialog parameter` configures the block to use the **Maximum output size** parameter setting as the maximum permitted output sequence length. When you make this selection, the **oSiz** input port specifies the current size of the output signal and the block output inherits sample time from the input signal. The input value of **oSiz** must be less than or equal to the **Maximum output size** parameter.
- `Inherit from reference port` adds the `Ref` input port and configures the block to inherit the sample time, maximum size, and the current output size from the variable-sized signal at the `Ref` input port to set the maximum permitted output sequence length.

Dependencies

To enable this parameter, select **Output variable-size signals**.

`Maximum output size` — Maximum output size

`[10 1]` (default) | two-element row vector

Maximum output size, specified as a two-element row vector that denotes the maximum output size for the block. The second element of the vector must be `1`.

Example: `[10 1]` gives a 10-by-1 maximum sized output signal.

Dependencies

To enable this parameter select **Output variable-size signals** and set **Maximum output size source** to `Dialog parameter`.

Data Types: `double`

`Sample time` — Output sample time

`1` (default) | -1 | positive scalar

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-665.

Example: `1` specifies a sample time of 1 second.

Dependencies

To enable this parameter, clear **Output variable-size signals**.

Data Types: `double`

`Samples per frame` — Samples per frame

`1` (default) | positive integer

Samples per frame in one channel of the output signal, specified as a positive integer. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-665.

Dependencies

To enable this parameter, clear **Output variable-size signals**.

Data Types: `double`

`Reset on nonzero input` — Reset on nonzero input

off (default) | on

Select this parameter to add the `Rst` input port. For more information, see “Reset Behavior” on page 5-663.

`Enable bit-packed outputs` — Enable bit-packed outputs

off (default) | on

Select this parameter to make the **Number of packed bits** and **Interpret bit-packed values as signed** parameters available.

When this parameter is selected, the object outputs a column vector of length *M*, which contains most-significant-bit (MSB) first integer representations of bit words of length *P*. *M* is the number of samples per frame specified in the **Samples per frame** parameter. *P* is the size of the bit-packed words specified in the **Number of packed bits** parameter.

Note The first bit from the left in the bit-packed word contains the most significant bit for the integer representation.

`Number of packed bits` — Number of packed bits

8 (default) | integer in the range [1, 32]

Number of packed bits, specified as an integer in the range [1, 32].

Dependencies

To enable this parameter, select **Enable bit-packed outputs**.

Data Types: `double`

`Interpret bit-packed values as signed` — Interpret bit-packed values as signed

off (default) | on

Interpret bit-packed values as signed integer data values when selected or unsigned integer data values when cleared. When selected, a 1 in the most significant bit (sign bit) indicates a negative value.

Dependencies

To enable this parameter, select **Enable bit-packed outputs**.

`Output data type` — Output data type

`double` (default) | `boolean` | `Smallest unsigned integer`

Output data type, specified as `double`, `boolean`, or `Smallest unsigned integer`.

- When **Enable bit-packed outputs** is cleared, the output data type can be specified as a `double`, `boolean`, or `Smallest unsigned integer`. When the **Output data type** parameter is set to `Smallest unsigned integer`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type `ufix(1)` = ideal minimum one-bit size. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (for example, `uint8`).
- When **Enable bit-packed outputs** is selected, the output data type can be specified as `double` or `Smallest unsigned integer`. When the **Output data type** parameter is set to `Smallest unsigned integer`, the output data type is selected based on the **Interpret bit-packed values as signed** and **Number of packed bits** parameters, and the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum *n*-bit size, such as `sfix(n)` or `ufix(n)`, based on the **Interpret bit-packed values as signed** parameter. For all other selections, it is a signed or unsigned integer with the smallest available word length large enough to fit *n* bits.

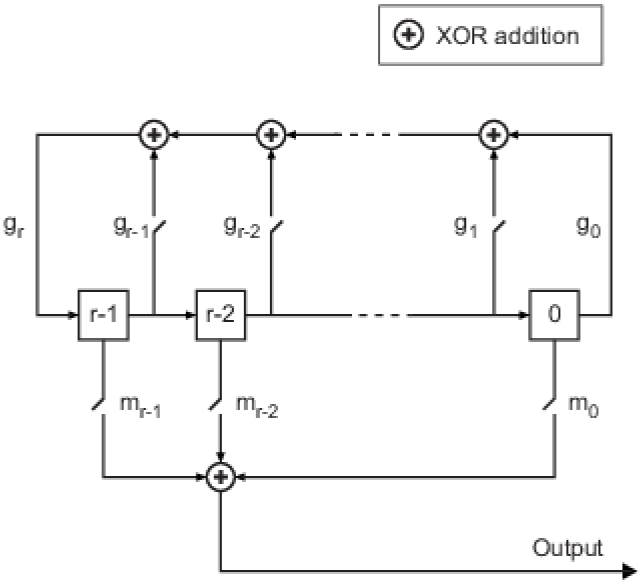
Block Characteristics

Data Types	`Boolean` | `double` | `fixed point`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

Simple Shift Register Generator

A linear-feedback shift register (LFSR), implemented as a simple shift register generator (SSRG), is used to generate PN sequences. This type of shift register is also known as a Fibonacci implementation. For an example, see “Model PN Sequence Generation With Linear Feedback Shift Register”.



The **Generator Polynomial** parameter determines the feedback connections of the shift register. It is a primitive binary polynomial in z, $g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \ldots + g_0$. For the coefficient, $g_{k=0 \text{ to } r}$, the coefficient g_k is 1 if there is a connection from the kth register to the adder. The leading term, g_r, and the constant term, g_0, of the **Generator Polynomial** parameter must be 1 because the polynomial must be primitive. The **Initial states** parameter specifies the initial values of the registers. For example, the following table indicates two sets of parameter values that correspond to a generator polynomial of $p(z) = z^8 + z^2 + 1$.

Quantity	Example 1	Example 2
Generator polynomial	`g1 = [1 0 0 0 0 0 1 0 1]`	`g2 = [8 2 0]`
Degree of generator polynomial	8, which is `length(g1)-1`	8
Initial states	`[1 0 0 0 0 0 1 0]`	`[1 0 0 0 0 0 1 0]`

At each time step, all *r* registers in the generator update their values according to the value of the incoming arrow to the shift register. The adders perform addition modulo 2. The output of the LFSR reflects the sum of all connections in the *m* mask vector.

The **Output mask vector (or scalar shift value)** parameter, *m*, determines the shift of the PN sequence starting point. For more information, see “Shifting PN Sequence Starting Point” on page 5-663.

Shifting PN Sequence Starting Point

To shift the starting point of the PN sequence, specify the **Output mask vector (or scalar shift value)** parameter as:

- An integer representing the length of the shift.

  The default **Output mask vector (or scalar shift value)** setting of 0 corresponds to no shift. As illustrated in the LFSR shift register diagram in “Simple Shift Register Generator” on page 5-662, there is no shift when the only connection is along the arrow labeled m_0.

  This table shows the shift that occurs when you set **Output mask vector (or scalar shift value)** to 0 versus a positive integer *d*.

	T = 0	**T = 1**	**T = 2**	**…**	**T = *d***	**T = *d*+1**
Shift = 0	x_0	x_1	x_2	…	x_d	x_{d+1}
Shift = *d*	x_d	x_{d+1}	x_{d+2}	…	x_{2d}	x_{2d+1}

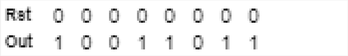
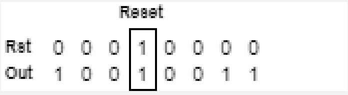
- A binary vector whose length is equal to the degree of the generator polynomial. The LFSR shift register diagram in “Simple Shift Register Generator” on page 5-662 shows **Output mask vector (or scalar shift value)** specified as a mask vector, *m*. The binary vector must have *N* elements, where *N* is the degree of the generator polynomial. To calculate the mask vector, use the `shift2mask` function.

  The binary vector corresponds to a polynomial in *z*, $m_{r-1}z^{r-1} + m_{r-2}z^{r-2} + \dots + m_1z + m_0$, of degree at most r–1. The mask vector that correspond to a shift of *d* is the vector that represents $m(z) = z^d$ modulo $g(z)$, where $g(z)$ is the generator polynomial.

  For example, if the degree of the generator polynomial is 4, then the mask vector that corresponds to $d = 2$ is `[0 1 0 0]`, which represents the polynomial $m(z) = z^2$.

Reset Behavior

To reset the generator sequence, you must first select **Reset on nonzero input** to add the `Rst` input. Suppose that the PN Sequence Generator block outputs `[1 0 0 1 1 0 1 1]` when there is no reset. The following table shows the effect on the PN Sequence Generator block output for the property values indicated.

	Reset Signal Properties	PN Sequence Generator block	Reset Signal Output Signal
No reset	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 0 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1 **Out** = `[1 0 0 1 1 0 1 1]`	 Rst 0 0 0 0 0 0 0 0 Out 1 0 0 1 1 0 1 1
Scalar reset signal	**Sample time** = 1 **Samples per frame** = 1 **Rst** = `[0 0 0 1 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 1	 Reset Rst 0 0 0 1 0 0 0 0 Out 1 0 0 1 0 0 1 1
Vector reset signal	**Sample time** = 1 **Samples per frame** = 8 **Rst** = `[0 0 0 1 0 0 0 0]`	**Sample time** = 1 **Samples per frame** = 8	

For the no reset case, the sequence is output without being reset. For the scalar and vector reset signal cases, the reset signal `[0 0 0 1 0 0 0 0]` is input to the `Rst` port. The sequence output is reset at the fourth bit, because the fourth bit of the reset signal is a 1 and the **Sample time** is 1.

For variable-sized outputs, the block only supports scalar reset signal inputs.

Sequences of Maximum Length

To generate a maximum length sequence for a generator polynomial that has the degree *r*, set **Generator polynomial** to a value from the following table. The maximum sequence length is $2^r - 1$.

r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial
2	`[2 1 0]`	15	`[15 14 0]`	28	`[28 25 0]`	41	`[41 3 0]`
3	`[3 2 0]`	16	`[16 15 13 4 0]`	29	`[29 27 0]`	42	`[42 23 22 1 0]`
4	`[4 3 0]`	17	`[17 14 0]`	30	`[30 29 28 7 0]`	43	`[43 6 4 3 0]`
5	`[5 3 0]`	18	`[18 11 0]`	31	`[31 28 0]`	44	`[44 6 5 2 0]`
6	`[6 5 0]`	19	`[19 18 17 14 0]`	32	`[32 31 30 10 0]`	45	`[45 4 3 1 0]`
7	`[7 6 0]`	20	`[20 17 0]`	33	`[33 20 0]`	46	`[46 21 10 1 0]`
8	`[8 6 5 4 0]`	21	`[21 19 0]`	34	`[34 15 14 1 0]`	47	[47 14 0]
9	`[9 5 0]`	22	`[22 21 0]`	35	`[35 2 0]`	48	[48 28 27 1 0]
10	`[10 7 0]`	23	`[23 18 0]`	36	`[36 11 0]`	49	[49 9 0]

r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial	r	Generator Polynomial
11	[11 9 0]	24	[24 23 22 17 0]	37	[37 12 10 2 0]	50	[50 4 3 2 0]
12	[12 11 8 6 0]	25	[25 22 0]	38	[38 6 5 1 0]	51	[51 6 3 1 0]
13	[13 12 10 9 0]	26	[26 25 24 20 0]	39	[39 8 0]	52	[52 3 0]
14	[14 13 8 4 0]	27	[27 26 25 22 0]	40	[40 5 4 3 0]	53	[53 6 2 1 0]

For more information about the shift-register configurations that these polynomials represent, see *Digital Communications* by John Proakis.[1].

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the PN Sequence Generator block version available before R2015b.

Existing models automatically update to load the PN Sequence Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

References

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

[2] Lee, J. S., and L. E. Miller. *CDMA Systems Engineering Handbook*. Boston and London. Artech House, 1998.

[3] Golomb, S.W. *Shift Register Sequences*. Laguna Hills. Aegean Park Press, 1967.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- You can select `Input port` as the **Output mask source** on the block. In this case, the `Mask` input signal must be a vector of data type `ufix1`.
- If you select **Reset on nonzero input**, the input to the `Rst` port must have data type `Boolean`.
- Outputs of type `double` are not supported for HDL code generation. All other output types (including bit-packed outputs) are supported.
- You cannot generate HDL for this block inside a Resettable Synchronous Subsystem.
- You cannot generate HDL for this block inside a Triggered Subsystem if the **Use trigger signal as clock** option is selected. See “Using Triggered Subsystems for HDL Code Generation” (HDL Coder).

See Also

Blocks
Gold Sequence Generator | Hadamard Code Generator | Kasami Sequence Generator | Scrambler

Objects
`comm.PNSequence`

Topics
“Spreading Sequences”

Introduced before R2006a

Poisson Integer Generator

Generate Poisson-distributed random integers

Library

Random Data Sources sublibrary of Comm Sources

Description

The Poisson Integer Generator block generates random integers using a Poisson distribution. The probability of generating a nonnegative integer *k* is

$$\lambda^{k}\exp(-\lambda)/(k!)$$

where λ is a positive number known as the Poisson parameter.

You can use the Poisson Integer Generator to generate noise in a binary transmission channel. In this case, the Poisson parameter **Lambda** should be less than 1, usually much less.

Attributes of Output Signal

The output signal can be a column or row vector, a two-dimensional matrix, or a scalar. The number of rows in the output signal corresponds to the number of samples in one frame and is determined by the **Samples per frame** parameter. The number of columns in the output signal corresponds to the number of channels and is determined by the number of elements in the **Lambda** parameter. See "Sources and Sinks" in *Communications Toolbox User's Guide* for more details.

Parameters

Lambda

The Poisson parameter λ. Specify λ as a scalar or row vector whose elements are real numbers. If **Lambda** is a scalar, then every element in the output vector shares the same Poisson parameter. If **Lambda** is a row vector, then the number of elements correspond to the number of independent channels output from the block.

Source of initial seed

The source of the initial seed for the random number generator. Specify the source as either `Auto` or `Parameter`. When set to `Auto`, the block uses the global random number stream.

> **Note** When **Source of initial seed** is `Auto` in `Code generation` mode, the random number generator uses an initial seed of zero. Therefore, the block generates the same random numbers each time it is started. Use `Interpreted execution` to ensure that the model uses different initial seeds. If `Interpreted execution` is run in `Rapid accelerator` mode, then it behaves the same as `Code generation` mode.

Initial seed

The initial seed value for the random number generator. Specify the seed as a nonnegative integer scalar. **Initial seed** is available when the **Source of initial seed** parameter is set to `Parameter`.

Sample time

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-668.

Samples per frame

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-668.

Output data type

The output type of the block can be specified as a `boolean`, `uint8`, `uint16`, `uint32`, `single`, or `double`. The default is `double`.

Simulate using

Select the simulation mode.

`Code generation`

On the first model run, simulate and generate code. If the structure of the block does not change, subsequent model runs do not regenerate the code.

If the simulation mode is `Code generation`, System objects corresponding to the blocks accept a maximum of nine inputs.

`Interpreted execution`

Simulate model without generating code. This option results in faster start times but can slow subsequent simulation performance.

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Poisson Integer Generator block version available before R2015b.

- Existing models automatically update to load the Poisson Integer Generator block version announced in R2015b. For more information on block forwarding, see “Forwarding Tables” (Simulink).

- Behavior of the random number generator is changed. The statistics are improved. For more information, see “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Random Integer Generator

Functions

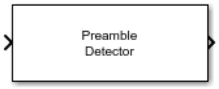
`poissrnd`

Introduced before R2006a

Preamble Detector

Detect preamble in data packet

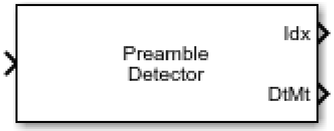
Library: Communications Toolbox / Synchronization



Description

The Preamble Detector block detects the end of preambles in data packets. A preamble is a set of symbols or bits used in packet-based communications systems to indicate the start of a packet. Packets consist of preamble data and user data. The length of the user data portion of the packet can vary during a simulation run.

This icon shows the block with all ports enabled:



Input/Output Ports

Input

In — Input data
scalar | column vector

Input data of symbols or bits, specified as a scalar or column vector. The input data can contain multiple packets. This port is unnamed on the block.

Data Types: `single` | `double` | `Boolean` | `int8` | `uint8`

Output

`Idx` — Index of last preamble symbol
scalar | column vector

Index of the last preamble symbol, returned as a scalar or column vector of the same size and data type as the input data.

- When the Detections parameter is set to `All`, `Idx` outputs the index corresponding to the last element of each detected preamble.
- When the Detections parameter is set to `First`, `Idx` outputs the index corresponding to the last element of the first detected preamble.

This port is unnamed until the `DtMt` port is enabled.

`DtMt` — Detection metric
scalar | column vector

Detection metric, returned as a scalar or column vector of the same size and data type as the input data packet.

- If either the preamble or input data is complex, the detection metric is the absolute value of the cross-correlation of the preamble and the input data.
- If both the preamble and input data are real, the detection metric is the cross-correlation of the preamble and the input data.

Dependencies

To enable this port, set the Input parameter to `Symbol`, and select the Output detection metric parameter.

Parameters

`Input` — Input type

`Symbol` (default) | `Bit`

Input type, specified as `Symbol` or `Bit`.

- For binary inputs, set this parameter to `Bit`.
- For all other inputs, set this parameter to `Symbol`.

For information on execution speed, see “Tips” on page 5-672.

`Preamble` — Preamble sequence

`[1 + 1i; 1 - 1i]` (default) | column vector

Preamble sequence, specified as a column vector.

- If the **Input** parameter is set to `Bit`, the preamble must be binary.
- If the **Input** parameter is set to `Symbol`, the preamble can be any real or complex sequence.

`Detection threshold` — Detection threshold

3 (default) | nonnegative scalar

Detection threshold, specified as a nonnegative scalar. When the detection metric is greater than or equal to the threshold, the block detects the preamble and updates `Idx`.

Tunable: Yes

Dependencies

To enable this parameter, set the Input parameter to `Symbol`.

`Output detection metric` — Option to output detection metric

`off` (default) | `on`

Select this parameter to output the detection metric and enable the `DtMt` output port.

Dependencies

To enable this parameter, set the Input parameter to `Symbol`.

`Detections` — Detections returned

`All` (default) | `First`

Detections returned, specified as `All` or `First`. Specifying `All` returns all detected preambles. Specifying `First` returns only the first detected preamble.

Tunable: Yes

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as:

- `Code generation` -- Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` -- Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

For information on execution speed, see “Tips” on page 5-672.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

Tips

- For faster execution of the Preamble Detector block, set the Simulate using parameter to:
  - `Code generation` when the Input parameter is set to `Symbol`
  - `Interpreted execution` when the Input parameter is set to `Bit`

Algorithms

Bit Inputs

When the input data is composed of bits, the preamble detector uses an exact pattern match.

Symbol Inputs

When the input data is composed of symbols, the preamble detector uses a cross-correlation algorithm. A finite impulse response (FIR) filter, in which the coefficients are specified from the preamble, computes the cross-correlation between the input data and the preamble. When a sequence of input samples match the preamble, the filter output reaches its peak. The index of the peak corresponds to the end of the preamble sequence in the input data. See Discrete FIR Filter for further information on the FIR filter algorithm.

The cross-correlation values that are greater than or equal to the specified threshold are reported as peaks.

- If the detection threshold is too low, the algorithm will detect false peaks, or, in the extreme case, detect as many detected peaks as there are input samples.
- If the detection threshold is too high, the algorithm will miss detecting peaks, or, in the extreme case, detect no peaks.

Consequently, the selection of the detection threshold is critical.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Carrier Synchronizer | Coarse Frequency Compensator | Symbol Synchronizer

Objects

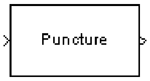
`comm.PreambleDetector`

Introduced in R2016b

Puncture

Output elements that correspond to 1s in binary puncture vector

Library: Communications Toolbox / Sequence Operations



Description

The Puncture block creates an output vector by removing selected elements of the input vector and preserving others. The block determines which elements to remove and preserve by using the binary **Puncture vector** parameter. The block repeats the puncturing pattern, as necessary, to include all input elements. The preserved elements appear in the output vector in the same order in which they appear in the input vector.

Ports

Input

`In` — Input signal

column vector

Input signal, specified as a column vector. The input length must be an integer multiple of the **Puncture vector** parameter length.

Data Types: `double` | `single`

Output

`Out` — Output signal

column vector

Output signal, returned as a column vector. The length of the output vector is an integer multiple of the number of 1s in the **Puncture vector** parameter. The output signal contains only elements from the input signal that align with integer multiples of the element location of 1s in the **Puncture vector**.

Parameters

`Puncture vector` — Puncture pattern

`[1 1 0 1 0 1]'` (default) | column vector of binary values

Puncture pattern, specified as a column vector of binary values. The input signal length must be an integer multiple of the **Puncture vector** parameter length. The block repeats the puncturing pattern, as necessary, to include all input elements.

- The element locations of 0s in **Puncture vector** indicate which elements are removed from the input signal to construct the output signal.

- The element locations of 1s in **Puncture vector** indicate which elements are preserved from the input signal to construct the output signal.

Block Characteristics

Data Types	`Boolean` | `double` | `enumerated` | `fixed point` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BCH Encoder | Binary-Input RS Encoder | Convolutional Encoder | Integer-Input RS Encoder

Introduced before R2006a

QPSK Demodulator Baseband

Demodulate QPSK-modulated data

Library

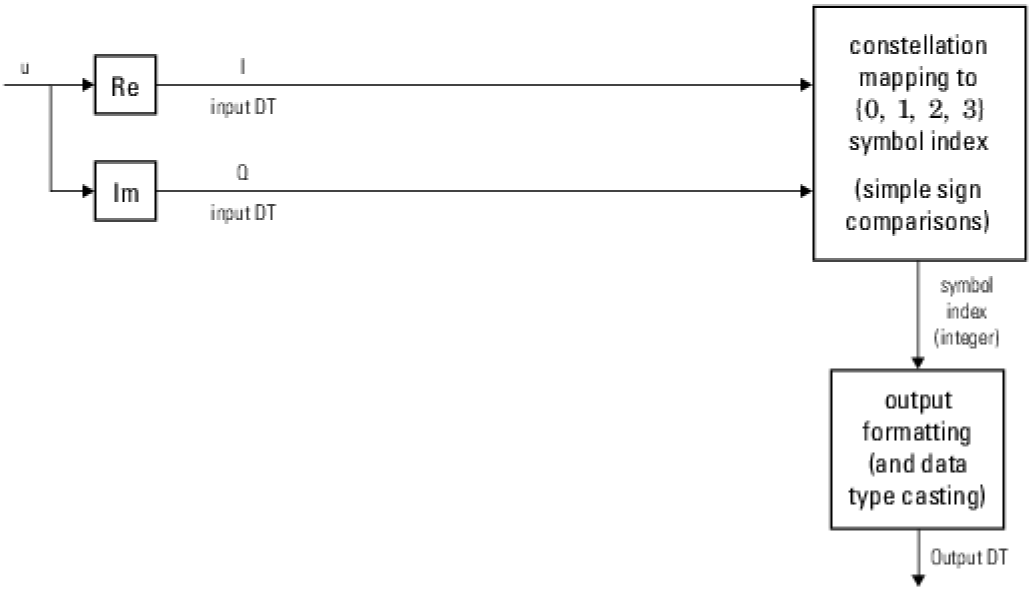
PM, in Digital Baseband sublibrary of Modulation

Description

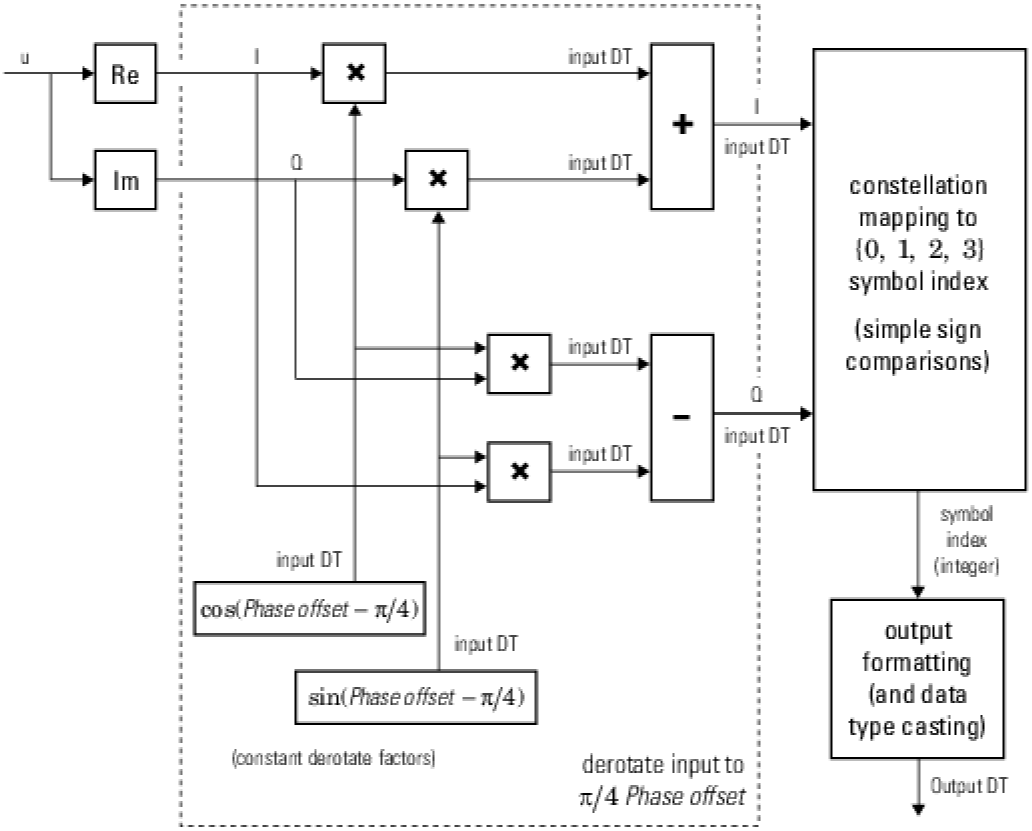
The QPSK Demodulator Baseband block demodulates a signal that was modulated using the quadrature phase shift keying method. The input is a baseband representation of the modulated signal.

The input must be a complex signal. This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-682.

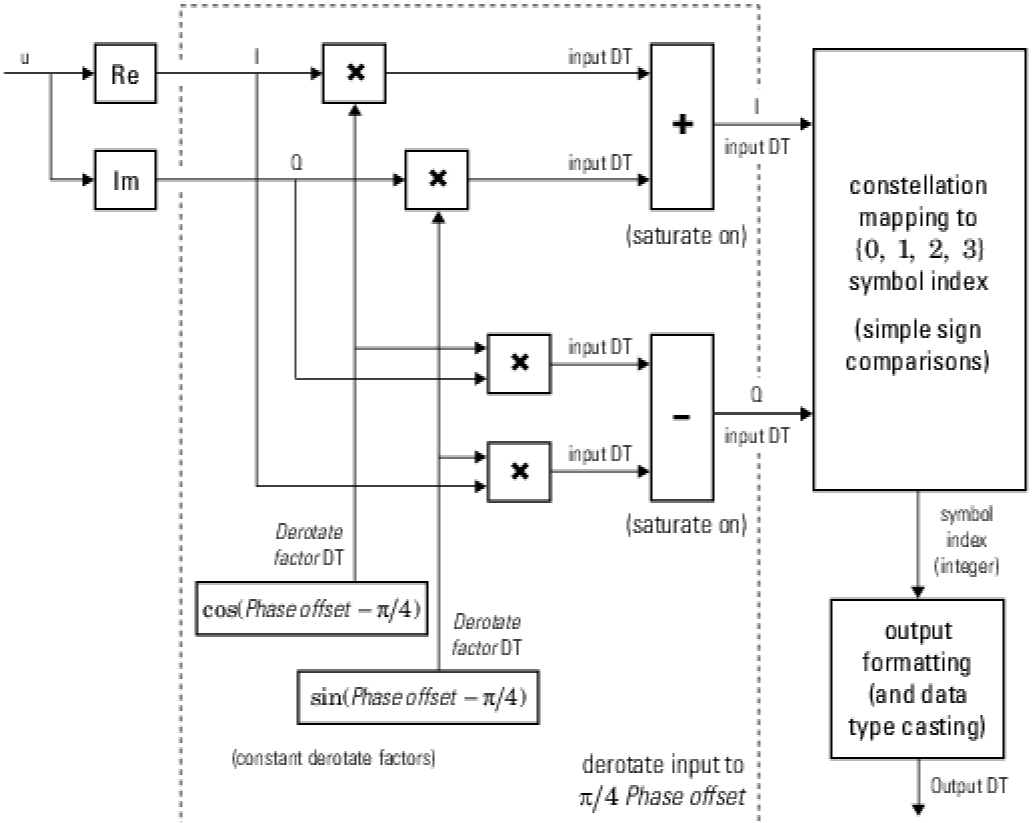
Algorithm


u
Re
Im
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT

Hard-Decision QPSK Demodulator Signal Diagram for Trivial Phase Offset (odd multiple of π/4)


u
Re
Im
I
Q
×
input DT
+
−
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
input DT
cos(Phase offset − π/4)
input DT
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset

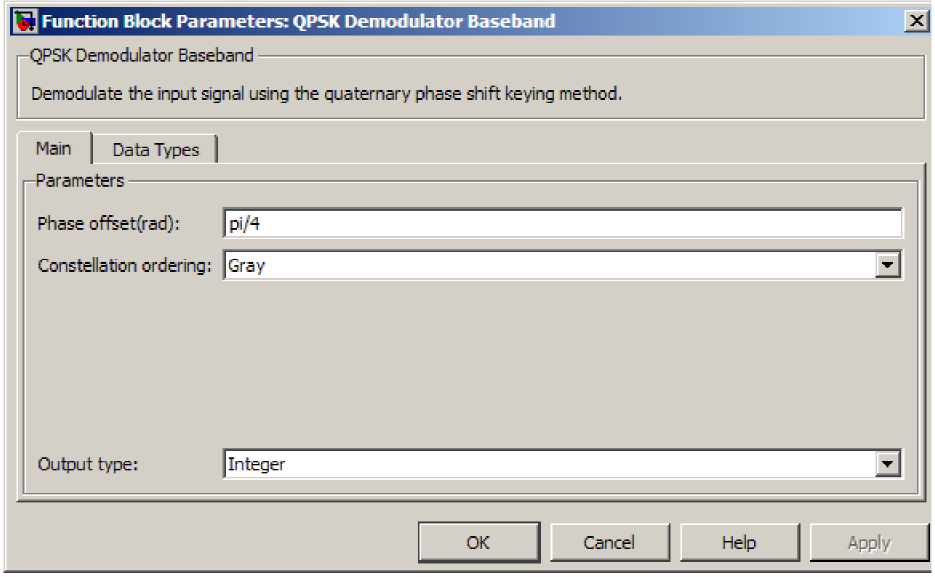
Hard-Decision QPSK Demodulator Floating-Point Signal Diagram for Nontrivial Phase Offset


u
Re
Im
I
Q
×
input DT
+
(saturate on)
−
I
input DT
Q
input DT
constellation mapping to {0, 1, 2, 3} symbol index
(simple sign comparisons)
symbol index (integer)
output formatting (and data type casting)
Output DT
Derotate factor DT
cos(Phase offset − π/4)
sin(Phase offset − π/4)
(constant derotate factors)
derotate input to π/4 Phase offset

Hard-Decision QPSK Demodulator Fixed-Point Signal Diagram for Nontrivial Phase Offset

The exact LLR and approximate LLR cases (soft-decision) are described in “Exact LLR Algorithm” and “Approximate LLR Algorithm”.

Dialog Box



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Main | Data Types

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Output type: Integer

Phase offset (rad)

The phase of the zeroth point of the signal constellation.

Constellation ordering

Determines how the block maps each integer to a pair of output bits.

Output type

Determines whether the output consists of integers or bits.

If the **Output type** parameter is set to `Integer` and **Constellation ordering** is set to `Binary`, then the block maps the point

$\exp(j\theta + j\pi m/2)$

to m, where θ is the **Phase offset** parameter and m is 0, 1, 2, or 3.

The reference page for the QPSK Modulator Baseband block shows the signal constellations for the cases when **Constellation ordering** is set to either `Binary` or `Gray`.

If the **Output type** is set to `Bit`, then the output contains pairs of binary values if **Decision type** is set to `Hard decision`. The most significant bit (i.e. the left-most bit in the vector), is the first bit the block outputs.

If the **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, then the output contains bitwise LLR or approximate LLR values, respectively.

Decision type

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. This parameter appears when you select `Bit` from the **Output type** drop-down list. The output values for Log-likelihood ratio and Approximate log-likelihood ratio decision types are of the same data type as the input values. For integer output, the block always performs Hard decision demodulation.

See “Exact LLR Algorithm” and “Approximate LLR Algorithm” for algorithm details.

Noise variance source

This field appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

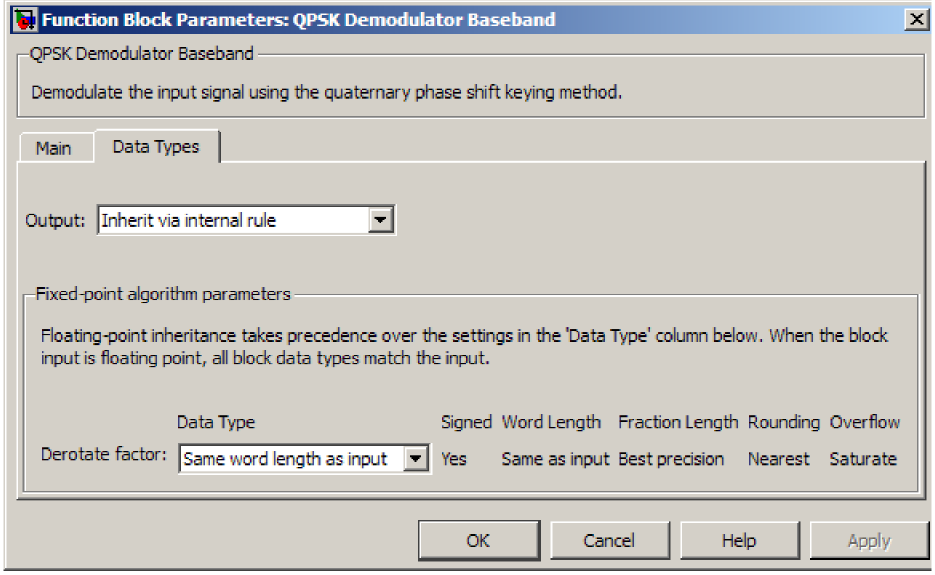
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Output: Inherit via internal rule

Fixed-point algorithm parameters

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Derotate factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate

Data Types Pane for Hard-Decision

Output

For bit outputs, when **Decision type** is set to `Hard decision`, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or `boolean`.

For integer outputs, the output data type can be set to `'Inherit via internal rule'`, `'Smallest unsigned integer'`, `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, or `uint32`.

When this parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is a floating-point type (`single` or `double`). If the input data type is fixed-point, the output data type will work as if this parameter is set to `'Smallest unsigned integer'`.

When this parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model.

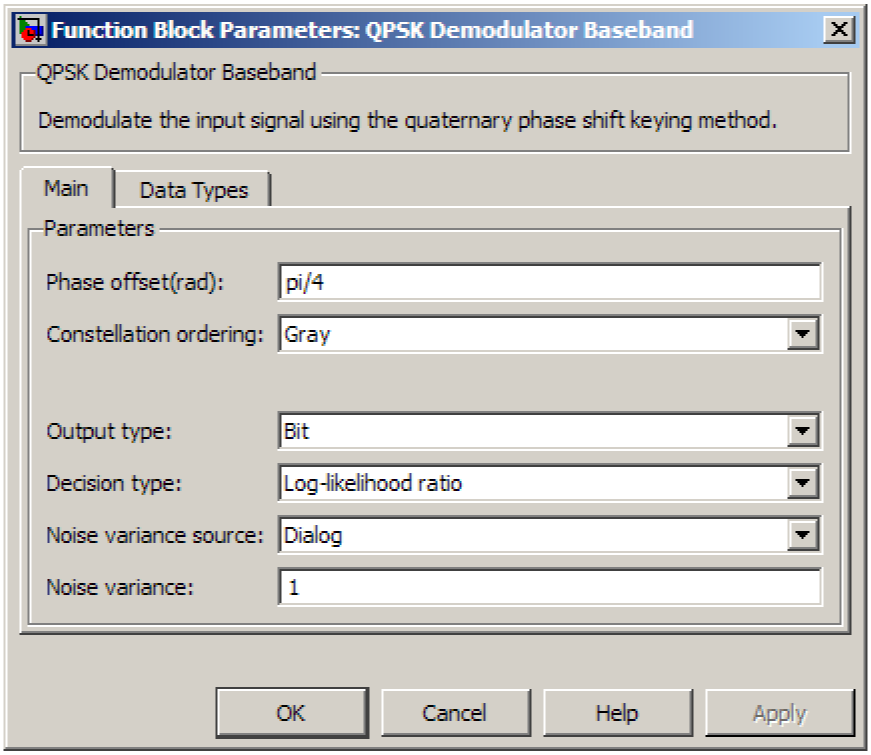
If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, and **Output type** is `Bit`, the output data type is the ideal minimum one-bit size, i.e., `ufix(1)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit one bit, usually corresponding to the size of a char (e.g., `uint8`).

If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, and **Output type** is `Integer`, the output data type is the ideal minimum two-bit size, i.e., `ufix(2)`. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit two bits, usually corresponding to the size of a char (e.g., `uint8`).

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not an even multiple of π/4.

You can select `Same word length as input` or `Specify word length`, in which case you define the word length using an input field.



Function Block Parameters: QPSK Demodulator Baseband

QPSK Demodulator Baseband

Demodulate the input signal using the quaternary phase shift keying method.

Main | Data Types

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Output type: Bit

Decision type: Log-likelihood ratio

Noise variance source: Dialog

Noise variance: 1

Data Types Pane for Soft-Decision

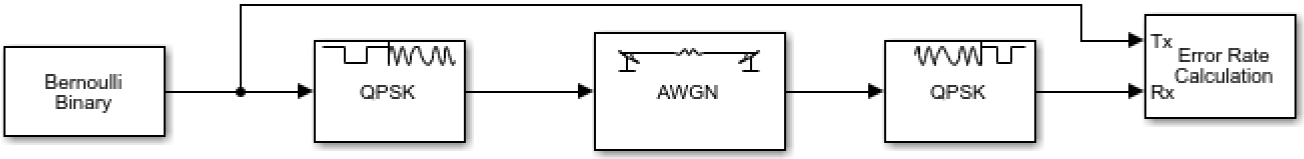
For bit outputs, when **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the output data type is inherited from the input (e.g., if the input is of data type `double`, the output is also of data type `double`).

Examples

Demodulate Noisy QPSK Signal

Modulate and demodulate a noisy QPSK signal.

Open the QPSK demodulation model.



Run the simulation. The results are saved to the base workspace, where the variable `ErrorVec` is a 1-by-3 row vector. The BER is found in the first element.

Display the error statistics. For the Eb/No provided, 4.3 dB, the resultant BER is approximately 0.01. Your results may vary slightly.

```
ans =

    0.0112
```

Increase the Eb/No to 7 dB. Rerun the simulation, and observe that the BER has decreased.

```
ans =

   1.0000e-03
```

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed-point when: • **Output type** is `Integer` • **Output type** is `Bit` and **Decision type** is `Hard-decision`
Var	• Double-precision floating point • Single-precision floating point

Port	Supported Data Types
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` and **Decision type** is `Hard-decision` • 8-, 16-, 32- bit signed integers • 8-, 16-, 32- bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • ufix(2) in ASIC/FPGA when **Output type** is `Integer`

Pair Block

QPSK Modulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

BPSK Demodulator Baseband | DQPSK Demodulator Baseband | M-PSK Demodulator Baseband | QPSK Modulator Baseband

Introduced before R2006a

QPSK Modulator Baseband

Modulate using quadrature phase shift keying method

Library

PM in Digital Baseband sublibrary of Modulation

Description

The QPSK Modulator Baseband block modulates using the quadrature phase shift keying method. The output is a baseband representation of the modulated signal.

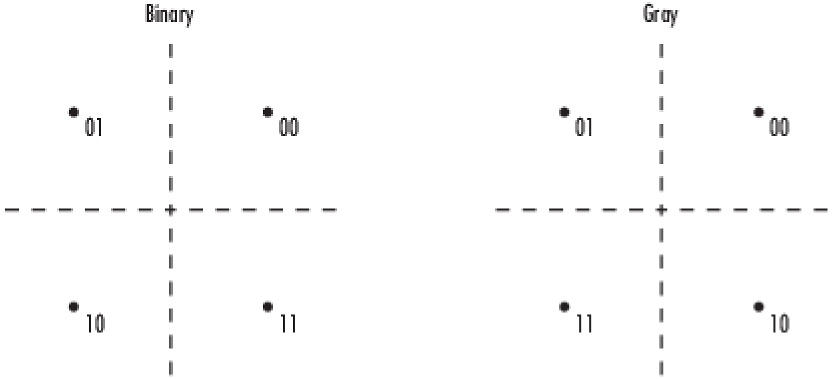
Integer-Valued Signals and Binary-Valued Signals

If you set the **Input type** parameter to `Integer`, then valid input values are 0, 1, 2, and 3. When you set **Constellation ordering** to `Binary` for input *m* the output symbol is

$\exp(j\theta + j\pi m/2)$

where θ represents the **Phase offset** parameter (see the following figure for Gray constellation ordering). In this case, the block accepts a scalar or column vector signal.

If you set the **Input type** parameter to `Bit`, then the input contains pairs of binary values. For this configuration, the block accepts column vectors with even lengths. When you set the **Phase offset** parameter to $\frac{\Pi}{4}$, then the block uses one of the signal constellations in the following figure, depending on whether you set the **Constellation ordering** parameter to `Binary` or `Gray`.

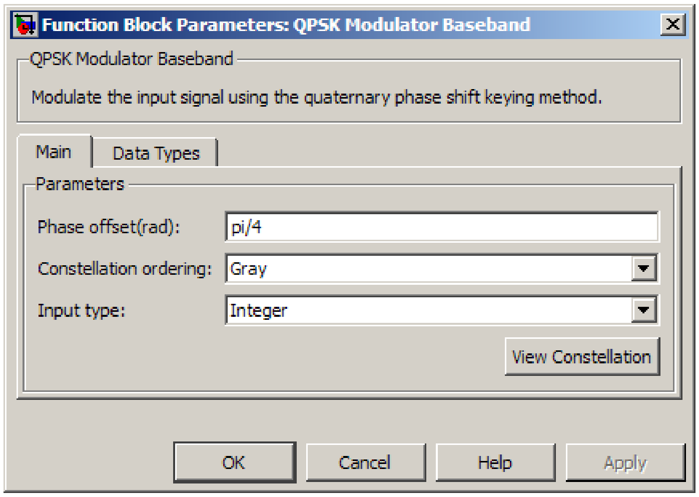


In the previous figure, the most significant bit (i.e. the left-most bit), is the first bit input to the block. For additional information about Gray mapping, see the M-PSK Modulator Baseband help page.

Constellation Visualization

The QPSK Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications Toolbox User's Guide*.

Dialog Box



Function Block Parameters: QPSK Modulator Baseband

QPSK Modulator Baseband

Modulate the input signal using the quaternary phase shift keying method.

Parameters

Phase offset(rad): pi/4

Constellation ordering: Gray

Input type: Integer

Phase offset (rad)

The phase of the zeroth point of the signal constellation.

Constellation ordering

Determines how the block maps each pair of input bits or input integers to constellation symbols.

Input type

Indicates whether the input consists of integers or pairs of bits.

Output data type

The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Output fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter or when you select `User-defined` and the specified output data type is a fixed-point data type.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Output fraction length

For fixed-point output data types, specify the number of fractional bits or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) when **Input type** is `Bit` • ufix(2) when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed point

Pair Block

QPSK Demodulator Baseband

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

See Also

Blocks

BPSK Modulator Baseband | DQPSK Modulator Baseband | M-PSK Modulator Baseband | QPSK Demodulator Baseband

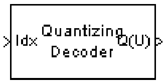
Topics

“Plot Noisy QPSK Constellation”
“Compare Filtered QPSK and MSK Signals in Simulink”

Introduced before R2006a

Quantizing Decoder

Decode quantization index according to codebook



Library

Source Coding

Description

The Quantizing Decoder block converts quantization indices to the corresponding codebook values. The **Quantization codebook** parameter, a vector of length N, prescribes the possible output values. If the input is an integer k between 0 and N-1, then the output is the (k+1)st element of **Quantization codebook**.

The input must be a discrete-time signal. This block processes each vector element independently. For information about the data types each block port supports, see the “Supported Data Type” on page 5-689 table on this page.

Note The Quantizing Encoder block also uses a **Quantization codebook** parameter. The first output of that block corresponds to the input of Quantizing Decoder, while the second output of that block corresponds to the output of Quantizing Decoder.

Parameters

Quantization codebook

A real vector that prescribes the output value corresponding to each nonnegative integer of the input.

Quantized output data type

Select the output data type.

Supported Data Type

Port	Supported Data Types
Idx	• Double-precision floating point • Single-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers

Port	Supported Data Types
Q(U)	• Double-precision floating point • Single-precision floating point

Pair Block

Quantizing Encoder

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

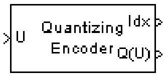
Blocks

Quantizing Encoder | Scalar Quantizer Decoder

Introduced before R2006a

Quantizing Encoder

Quantize signal using partition and codebook



Library

Source Coding

Description

The Quantizing Encoder block quantizes the input signal according to the **Partition** vector and encodes the input signal according to the **Codebook** vector. This block processes each vector element independently. The input must be a discrete-time signal. This block processes each vector element independently. For information about the data types each block port supports, see the “Supported Data Type” on page 5-692 table on this page.

The first output is the quantization index. The second output is the quantized signal. The values for the quantized signal are taken from the **Codebook** vector.

The **Quantization partition** parameter, *P*, is a real vector of length n whose entries are in strictly ascending order. The quantization index (second output signal value) corresponding to an input value of *x* is

- 0 if $x \leq P(1)$
- m if $P(\mathrm{m}) < x \leq P(\mathrm{m}+1)$
- n if $P(\mathrm{n}) < x$

The **Quantization codebook** parameter, whose length is n+1, prescribes a value for each partition in the quantization. The first element of **Quantization codebook** is the value for the interval between negative infinity and the first element of *P*. The second output signal from this block contains the quantization of the input signal based on the quantization indices and prescribed values.

Use the `lloyds` function with a representative sample of your data as training data, to obtain appropriate partition and codebook parameters.

Parameters

Quantization partition

The vector of endpoints of the partition intervals.

Quantization codebook

The vector of output values assigned to each partition.

Index output data type

Select the output data type.

Supported Data Type

Port	Supported Data Types
U	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Idx	• Double-precision floating point • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers
Q(U)	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Quantizing Decoder

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

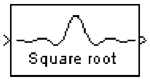
Blocks
Quantizing Decoder | Scalar Quantizer Encoder

Functions
`lloyds`

Introduced before R2006a

Raised Cosine Receive Filter

Apply pulse shaping by downsampling signal using raised cosine FIR filter



Library

Comm Filters

Description

The Raised Cosine Receive Filter block filters the input signal using a normal raised cosine FIR filter or a square root raised cosine FIR filter. It also downsamples the filtered signal if you set the **Output mode** parameter to `Downsampling`. The FIR Decimation block implements this functionality. The Raised Cosine Receive Filter block's icon shows the filter's impulse response.

Characteristics of the Filter

Characteristics of the raised cosine filter are the same as in the Raised Cosine Transmit Filter block, except that the length of the filter's input response has a slightly different expression: *L* * **Filter span in symbols** + 1, where *L* is the value of the **Input samples per symbol** parameter (not the **Output samples per symbol** parameter, as in the case of the Raised Cosine Transmit Filter block).

The block normalizes the filter coefficients to unit energy. If you specify a **Linear amplitude filter gain** other than `1`, then the block scales the normalized filter coefficients using the gain value you specify.

Decimating the Filtered Signal

To have the block decimate the filtered signal, set the **Decimation factor** parameter to a value greater than `1`.

If `K` represents the **Decimation factor** parameter value, then the block retains 1/*K* of the samples, choosing them as follows:

- If the **Decimation offset** parameter is zero, then the block selects the samples of the filtered signal indexed by 1, *K*+1, 2**K*+1, 3**K*+1, etc.
- If the **Decimation offset** parameter is a positive integer less than M, then the block initially discards that number of samples from the filtered signal and downsamples the remaining data as in the previous case.

To preserve the entire filtered signal and avoid decimation, set **Decimation factor** to `1`. This setting is appropriate, for example, when the output from the filter block forms the input to a timing phase recovery block such as Symbol Synchronizer. The timing phase recovery block performs the downsampling in that case.

Input Signals and Output Signals

This block accepts a column vector or matrix input signal. For information about the data types each block port supports, see the “Supported Data Type” on page 5-697 table on this page.

If you set **Decimation factor** to `1`, then the input and output signals share the same sampling mode, sample time, and vector length.

If you set **Decimation factor** to *K*, which is greater than `1`, then *K* and the input sampling mode determine characteristics of the output signal:

Single-Rate Processing

When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To generate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is $1/K$ times that of the input ($M_o = M_i/K$), In this mode, the input frame size, M_i, must be a multiple of *K*.

Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size, but the sample rate of the output is *K* times slower than that of the input. When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an *M*-by-*N* matrix input as *M*N* independent channels, and processes each channel over time. The output sample period (T_{so}) is *K* times longer than the input sample period ($T_{so} = K*T_{si}$), and the input and output sizes are identical.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-*N* matrix input as *N* independent channels. The block processes each column of the input over time by keeping the frame size constant (M_i=M_o), and making the output frame period (T_{fo}) *K* times longer than the input frame period ($T_{fo} = K*T_{fi}$).

Exporting Filter Coefficients to the MATLAB Workspace

To examine or manipulate the coefficients of the filter that this block designs, select **Export filter coefficients to workspace**. Then set the **Coefficient variable name** parameter to the name of a variable that you want the block to create in the MATLAB workspace. Running the simulation causes the block to create the variable, overwriting any previous contents in case the variable already exists.

Latency

For information pertaining to the latency of the block, see details in FIR Decimation.

Parameters

Filter shape

Specify the filter shape as `Square root` or `Normal`.

Rolloff factor

Specify the rolloff factor of the filter. Use a real number between `0` and `1`.

Filter span in symbols

Specify the number of symbols the filter spans as an even, integer-valued positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that this parameter specifies.

Input samples per symbol

An integer greater than 1 representing the number of samples that represent one symbol in the input signal.

Decimation factor

Specify the decimation factor the block applies to the input signal. The output samples per symbol equals the value of the input samples per symbol divided by the decimation factor. If the decimation factor is one, then the block only applies filtering. There is no decimation.

Decimation offset

Specify the decimation offset in samples. Use a value between `0` and **Decimation factor** `-1`.

Linear amplitude filter gain

Specify a positive scalar value that the block uses to scale the filter coefficients. By default, the block normalizes filter coefficients to provide unit energy gain. If you specify a gain other than `1`, the block scales the normalized filter coefficients using the gain value you specify.

Input processing

Specify how the block processes the input signal. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should filter and downsample the input signal. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate and processes the signal by decreasing the output frame size by a factor of *K*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block processes the signal such that the output sample rate is *K* times slower than the input sample rate.

Export filter coefficients to workspace

Select this check box to create a variable in the MATLAB workspace that contains the filter coefficients.

Coefficient variable name

The name of the variable to create in the MATLAB workspace. This field appears only if **Export filter coefficients to workspace** is selected.

Visualize filter with FVTool

If you click this button, then MATLAB launches the Filter Visualization Tool, `fvtool`, to analyze the raised cosine filter whenever you apply any changes to the block's parameters. If you launch `fvtool` for the filter, and subsequently change parameters in the mask, `fvtool` will not update.

You will need to launch a new `fvtool` in order to see the new filter characteristics. Also note that if you have launched `fvtool`, then it will remain open even after the model is closed.

Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see Rounding Modes or "Rounding Mode: Simplest" (Fixed-Point Designer).

Saturate on integer overflow

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator).

See the Coefficients section of the FIR Decimation help page and "Filter Structure Diagrams" for illustrations depicting the use of the coefficient data types in this block:

See the Coefficients subsection of the Digital Filter help page for descriptions of parameter settings.

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See "Filter Structure Diagrams" and "Multiplication Data Types" for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Raised Cosine Transmit Filter

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block is a subsystem that contains a FIR Decimation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See the "HDL Code Generation" section of the Subsystem and FIR Decimation block reference pages for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Receive Filter block and execute this command.

```
set_param(gcb,'LinkStatus','inactive')
```

See Also

Blocks

Raised Cosine Transmit Filter | Symbol Synchronizer

Objects

`comm.RaisedCosineTransmitFilter`

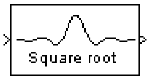
Functions

`rcosdesign`

Introduced before R2006a

Raised Cosine Transmit Filter

Apply pulse shaping by upsampling signal using raised cosine FIR filter



Library

Comm Filters

Description

The Raised Cosine Transmit Filter block upsamples and filters the input signal using a normal raised cosine FIR filter or a square root raised cosine FIR filter. The block's icon shows the filter's impulse response.

Characteristics of the Filter

The **Filter shape** parameter determines which type of filter the block uses; choices are `Normal` and `Square root`.

The impulse response of a normal raised cosine filter with rolloff factor R and symbol period T is

$$h(t) = \frac{\sin(\pi t/T)}{(\pi t/T)} \cdot \frac{\cos(\pi R t/T)}{(1 - 4R^2 t^2/T^2)}$$

The impulse response of a square root raised cosine filter with rolloff factor R is

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

The impulse response of a square root raised cosine filter convolved with itself is approximately equal to the impulse response of a normal raised cosine filter.

Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that the **Filter span in symbols** parameter specifies. The **Filter span in symbols**, N, and the **Output samples per symbol**, L, determine the length of the filter's impulse response, which is *L* * **Filter span in symbols** + 1.

The **Rolloff factor** parameter is the filter's rolloff factor. It must be a real number between 0 and 1. 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.

The block normalizes the filter coefficients to unit energy. If you specify a **Linear amplitude filter gain** other than `1`, then the block scales the normalized filter coefficients using the gain value you specify.

Input Signals and Output Signals

The input must be a discrete-time signal. This block accepts a column vector or matrix input signal. For information about the data types each block port supports, see the "Supported Data Type" on page 5-703 table on this page.

The **Rate options** method and the value of the **Output samples per symbol**, *L*, parameter determine the characteristics of the output signal:

Single-Rate Processing

When you set the **Rate options** parameter to `Enforce single-rate processing`, the input and output of the block have the same sample rate. To generate the output while maintaining the input sample rate, the block resamples the data in each column of the input such that the frame size of the output (M_o) is *L* times larger than that of the input ($M_o = M_i*L$), where *L* represents the value of the **Output samples per symbol** parameter.

Multirate Processing

When you set the **Rate options** parameter to `Allow multirate processing`, the input and output of the block are the same size. However, the sample rate of the output is *L* times faster than that of the input (i.e. the output sample time is 1/L times the input sample time). When the block is in multirate processing mode, you must also specify a value for the **Input processing** parameter:

- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats an *M*-by-*L* matrix input as *M**N* independent channels, and processes each channel over time. The output sample period (T_{so}) is *L* times shorter than the input sample period ($T_{so} = T_{si}/L$), while the input and output sizes remain identical.
- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats an M_i-by-*N* matrix input as *N* independent channels. The block processes each column of the input over time by keeping the frame size constant ($M_i=M_o$), while making the output frame period (T_{fo}) *L* times shorter than the input frame period ($T_{fo} = T_{fi}/L$).

Exporting Filter Coefficients to the MATLAB Workspace

To examine or manipulate the coefficients of the filter that this block designs, select **Export filter coefficients to workspace**. Then set the **Coefficient variable name** parameter to the name of a variable that you want the block to create in the MATLAB workspace. Running the simulation causes the block to create the variable, overwriting any previous contents in case the variable already exists.

Parameters

Filter shape

Specify the filter shape as `Square root` or `Normal`.

Rolloff factor

Specify the rolloff factor of the filter. Use a real number between `0` and `1`.

Filter span in symbols

Specify the number of symbols the filter spans as an even, integer-valued positive scalar. The default is `10`. Because the ideal raised cosine filter has an infinite impulse response, the block truncates the impulse response to the number of symbols that this parameter specifies.

Output samples per symbol

Specify the number of output samples for each input symbol. The default is 8. This property accepts an integer-valued, positive scalar. The number of taps for the raised cosine filter equals the value of this parameter multiplied by the value of the **Filter span in symbols** parameter.

Linear amplitude filter gain

Specify a positive scalar value that the block uses to scale the filter coefficients. By default, the block normalizes filter coefficients to provide unit energy gain. If you specify a gain other than `1`, the block scales the normalized filter coefficients using the gain value you specify.

Input processing

Specify how the block processes the input signal. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.
- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rate options

Specify the method by which the block should upsample and filter the input signal. You can select one of the following options:

- `Enforce single-rate processing` — When you select this option, the block maintains the input sample rate, and processes the signal by increasing the output frame size by a factor of *N*. To select this option, you must set the **Input processing** parameter to `Columns as channels (frame based)`.
- `Allow multirate processing` — When you select this option, the block processes the signal such that the output sample rate is *N* times faster than the input sample rate.

Export filter coefficients to workspace

Select this check box to create a variable in the MATLAB workspace that contains the filter coefficients.

Visualize filter with FVTool

If you click this button, then MATLAB launches the Filter Visualization Tool, `fvtool`, to analyze the raised cosine filter whenever you apply any changes to the block's parameters. If you launch `fvtool` for the filter, and subsequently change parameters in the mask, `fvtool` will not update. You will need to launch a new `fvtool` in order to see the new filter characteristics. Also note that if you have launched `fvtool`, then it will remain open even after the model is closed.

Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see Rounding Modes or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock data type settings against changes by the fixed-point tools

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Raised Cosine Receive Filter

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

This block is a subsystem that contains a FIR Interpolation block. You can set **HDL Properties** on the subsystem, or you can look under the mask and set **HDL Properties** on the filter block. See the "HDL Code Generation" section of the Subsystem and FIR Interpolation block reference pages for a list of properties.

To save setting changes under the mask, you must break the library link. To break the library link, select the Raised Cosine Transmit Filter block and execute this command.

```
set_param(gcb,'LinkStatus','inactive')
```

See Also

Blocks
Raised Cosine Receive Filter

Objects
`comm.RaisedCosineReceiveFilter`

Functions
`rcosdesign`

Introduced before R2006a

Random Deinterleaver

Restore ordering of input symbols using random permutation

Library

Block sublibrary of Interleaving

Description

The Random Deinterleaver block rearranges the elements of its input vector using a random permutation. The **Initial seed** parameter initializes the random number generator that the block uses to determine the permutation. If this block and the Random Interleaver block have the same value for **Initial seed**, then the two blocks are inverses of each other.

This block accepts a column vector input signal. The **Number of elements** parameter indicates how many numbers are in the input vector.

The block accepts the following data types `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

Parameters

Number of elements

The number of elements in the input vector.

Initial seed

The initial seed value for the random number generator.

Pair Block

Random Interleaver

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General Block Deinterleaver | Random Interleaver

Introduced before R2006a

Random Integer Generator

Generate integers randomly distributed in specified range

Library: Communications Toolbox / Comm Sources / Random Data Sources

Description

The Random Integer Generator block generates uniformly distributed random integers in the range [0, *M*-1], where *M* is specified by the Set size parameter. Use this block to generate random binary-valued or integer-valued data.

To ensure that the model uses different initial seeds, set the Simulate using parameter to `Interpreted execution`, and run the simulation in Normal or Accelerator mode. For more information, see “Limitations” on page 5-707.

Limitations

- In Rapid Accelerator simulation mode, when you set Simulate using to `Interpreted execution` and Source of initial seed to `Auto`, the block generates the same numbers every time the simulation runs. This behavior is equivalent to setting Source of initial seed to `Parameter` and setting Initial seed to `0`.
- In all simulation modes (Normal, Accelerator, and Rapid Accelerator), when you set Simulate using to `Code generation` and Source of initial seed to `Auto`, the block generates the same numbers every time the simulation runs. This behavior is equivalent to setting Source of initial seed to `Parameter` and Initial seed to `0`.

For more information, see “Choosing a Simulation Mode” (Simulink).

Ports

Output

`Out` — Random integer output
scalar | vector | matrix

Random integer output, returned as a scalar, vector, or matrix. This port is unnamed on the block. The data type is set using the Output data type parameter.

The number of rows in the output data equals the value of the Samples per frame parameter and corresponds to the number of samples in one frame. The number of columns in the output data equals the number of elements in the Set size parameter and corresponds to the number of channels.

Parameters

Set size — Set size

8 (default) | positive integer | row vector of positive integers

Set size,*M*, specified as a positive integer or row vector of positive integers. The block generates integers in the range [0, (*M* – 1)]. The number of elements in **Set size** corresponds to the number of independent channels output from the block.

- If **Set size** is a scalar, then all output random variables are independent and identically distributed (i.i.d.).
- If **Set size** is a vector, then the length of the vector determines the number of output channels. The channels can have differing output ranges.

Source of initial seed — Source of initial seed

Auto (default) | Parameter

Source of the initial seed for the random number generator, specified as either:

- Auto –– the block uses the global random number stream
- Parameter –– the block sets the random number generator seed to **Initial seed**

Initial seed — Initial seed value

0 (default) | nonnegative integer

Initial seed value for the random number generator, specified as a nonnegative integer. If the **Initial seed** parameter is a constant, then the resulting sequence is repeatable.

Dependencies

To enable this parameter, set the **Source of initial seed** parameter to Parameter.

Sample time — Sample time

1 (default) | -1 | positive scalar

Output sample time, specified as -1 or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to -1, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-709.

Samples per frame — Samples per frame

1 (default) | positive integer

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-709.

Output data type — Output data type

double (default) | single | uint8 | uint16 | uint32 | boolean

Output data type, specified as double, single, uint8, uint16, uint32, or boolean. If this parameter is set to boolean, you must set the Set size parameter to 2.

Simulate using — Type of simulation to run

Interpreted execution (default) | Code generation

Type of simulation to run, specified as:

- `Code generation` –– Simulate the model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations as long as the model does not change. This option requires additional startup time.
- `Interpreted execution` –– Simulate the model using the MATLAB interpreter. This option shortens startup time. In `Interpreted execution` mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Random Integer Generator block update supported in Upgrade Advisor
Behavior changed in R2020a

Starting in R2020a, Random Integer Generator block allows you to use the Upgrade Advisor. You can update to the block version announced in R2015b or keep the block version available before R2015b.

- Use the Upgrade Advisor to update existing models that include the Random Integer Generator block.
- Behavior of the random number generator is changed. The statistics are improved. For more information, see “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Functions
`randi`

Topics
“Sources and Sinks”
“Choosing a Simulation Mode” (Simulink)

Introduced before R2006a

Random Interleaver

Reorder input symbols using random permutation

Library

Block sublibrary of Interleaving

Description

The Random Interleaver block rearranges the elements of its input vector using a random permutation. This block accepts a column vector input signal. The **Number of elements** parameter indicates how many numbers are in the input vector.

The block accepts the following data types: `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, `double`, and fixed-point. The output signal inherits its data type from the input signal.

The **Initial seed** parameter initializes the random number generator that the block uses to determine the permutation. The block is predictable for a given seed, but different seeds produce different permutations.

Parameters

Number of elements

The number of elements in the input vector.

Initial seed

The initial seed value for the random number generator.

Pair Block

Random Deinterleaver

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
General Block Interleaver | Random Deinterleaver

Introduced before R2006a

Rayleigh Noise Generator

(To be removed) Generate Rayleigh distributed noise

Note Rayleigh Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

Library

Noise Generators sublibrary of Comm Sources

Description

The Rayleigh Noise Generator block generates Rayleigh distributed noise. The Rayleigh probability density function is given by

$$f(x) = \begin{cases} \dfrac{x}{\sigma^2}\exp\left(-\dfrac{x^2}{2\sigma^2}\right) & x \geq 0 \\ 0 & x < 0 \end{cases}$$

where σ^2 is known as the *fading envelope* of the Rayleigh distribution.

The block requires you to specify the **Initial seed** for the random number generator. If it is a constant, then the resulting noise is repeatable. The **sigma** parameter can be either a vector of the same length as the **Initial seed**, or a scalar. When **sigma** is a scalar, every element of the output signal shares that same value.

Initial Seed

The **Initial seed** parameter initializes the random number generator that the Rayleigh Noise Generator block uses to add noise to the input signal. When multiple blocks in a model have the **Initial seed** parameter, you can choose different initial seeds for each block to ensure different random streams are used in each block. Set **Initial seed** to an integer value for repeatable results or use the `randi` function to randomize your results.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters. See “Sources and Sinks” for more details.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.

Parameters

Sigma

Specify σ as defined in the Rayleigh probability density function.

Initial seed

The initial seed value for the random number generator.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame

The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D

If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type

The output can be set to `double` or `single` data types.

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

See Also

Blocks
MATLAB Function | MIMO Fading Channel

Functions
`isprime` | `randi` | `randn` | `raylrnd` | `rng`

Introduced before R2006a

Receiver Thermal Noise

Apply receiver thermal noise to complex signal
Library: Communications Toolbox / RF Impairments

Description

The Receiver Thermal Noise block applies receiver thermal noise to a complex signal. The block simulates the effects of thermal noise on a complex signal. The **Specification method** parameter enables specification of thermal noise based on noise temperature, noise figure, or noise factor.

Ports

Input

`In1` — Complex signal

scalar | column vector

Complex signal, specified as a scalar or column vector.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Out1` — Output signal

scalar | column vector

Output signal, returned as a scalar or column vector. This output is the same dimension and data type as the input signal.

Parameters

`Specification method` — Thermal noise specification method

`Noise temperature` (default) | `Noise figure` | `Noise factor`

Thermal noise specification method, specified as one of these options.

- `Noise temperature` specifies the noise in degrees K.
- `Noise figure` specifies the added receiver noise in dB relative to an input noise temperature of 290 K. The noise figure is the decibel equivalent of the noise factor.
- `Noise factor` specifies the added receiver noise relative to an input noise temperature of 290 K. The noise factor is the linear equivalent of the noise figure.

`Noise temperature (K)` — Noise temperature

`290` (default) | scalar

Noise temperature in degrees K, specified as a scalar.

Dependencies

To enable this parameter, set the **Specification method** parameter to `Noise temperature`.

`Noise figure (dB)` — Receiver noise figure

`3.0103` (default) | scalar

Receiver noise figure in dB relative to a noise temperature of 290 degrees K, specified as a scalar.

Note This parameter specifies the noise contribution of the receiver circuitry only. To add the effects of antenna noise, select the **Add 290K antenna noise** parameter.

Dependencies

To enable this parameter, set the **Specification method** parameter to `Noise figure`.

`Noise factor` — Receiver noise factor

`2` (default) | scalar

Receiver noise factor relative to a noise temperature of 290 degrees K, specified as a scalar.

Note This parameter specifies the noise contribution of the receiver circuitry only. To add the effects of antenna noise, select the **Add 290K antenna noise** parameter.

Dependencies

To enable this parameter, set the **Specification method** parameter to `Noise factor`.

`Add 290K antenna noise` — Option to add 290 degrees K antenna noise

`off` (default) | `on`

Select this parameter to add 290 degrees K antenna noise to the signal.

Dependencies

To enable this parameter, set the **Specification method** parameter to `Noise factor` or `Noise factor`.

`Initial seed` — Initial seed value

`67987` (default) | scalar

Initial seed value for the random number generator, specified as a scalar.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

Algorithms

Wireless receiver performance is often expressed as a noise factor or figure. The noise factor is defined as the ratio of the input signal-to-noise ratio, S_i/N_i to the output signal-to-noise ratio, S_o/N_o, such that

$$F = \frac{S_i/N_i}{S_o/N_o} \ .$$

Given receiver gain G and receiver noise power N_{ckt}, the noise factor can be expressed as

$$\begin{aligned} F &= \frac{S_i/N_i}{GS_i/(N_{ckt} + GN_i)} \\ &= \frac{N_{ckt} + GN_i}{GN_i} \ . \end{aligned}$$

The IEEE defines the noise factor assuming that noise temperature at the input is T_0, where T_0 = 290 K. The noise factor is then

$$\begin{aligned} F &= \frac{N_{ckt} + GN_i}{GN_i} \\ &= \frac{GkBT_{ckt} + GkBT_0}{GkBT_0} \\ &= \frac{T_{ckt} + T_0}{T_0} \ . \end{aligned}$$

T_{ckt} is the equivalent input noise temperature of the receiver and is expressed as

$$T_{ckt} = T_0(F - 1) \ .$$

The overall noise temperature of an antenna and receiver, T_{sys}, is

$$T_{sys} = T_{ant} + T_{ckt} \, ,$$

where T_{ant} is the antenna noise temperature.

The noise figure, NF, is the dB equivalent of the noise factor and can be expressed as

$$NF = 10\log_{10}(F) \ .$$

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Free Space Path Loss | I/Q Imbalance | Memoryless Nonlinearity | Phase Noise | Phase/Frequency Offset

Objects

`comm.ThermalNoise`

Introduced before R2006a

Rectangular QAM Demodulator Baseband

Demodulate rectangular-QAM-modulated data

Library

AM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM Demodulator Baseband block demodulates a signal that was modulated using quadrature amplitude modulation with a constellation on a rectangular lattice.

Note All values of power assume a nominal impedance of 1 ohm.

The signal constellation has M points, where M is the **M-ary number** parameter. M must have the form 2^K for some positive integer K. The block scales the signal constellation based on how you set the **Normalization method** parameter. For details, see the reference page for the Rectangular QAM Modulator Baseband block.

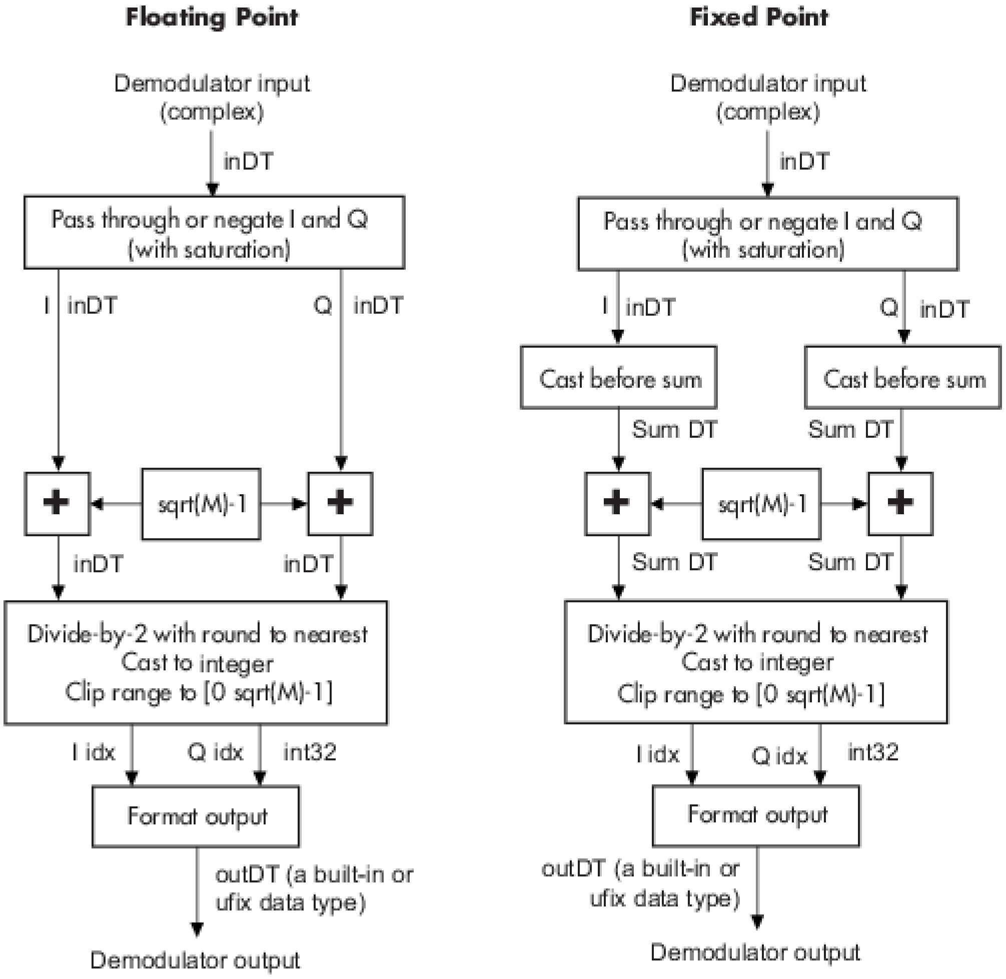
This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see the “Supported Data Types” on page 5-726 table on this page.

Hard Decision Algorithm

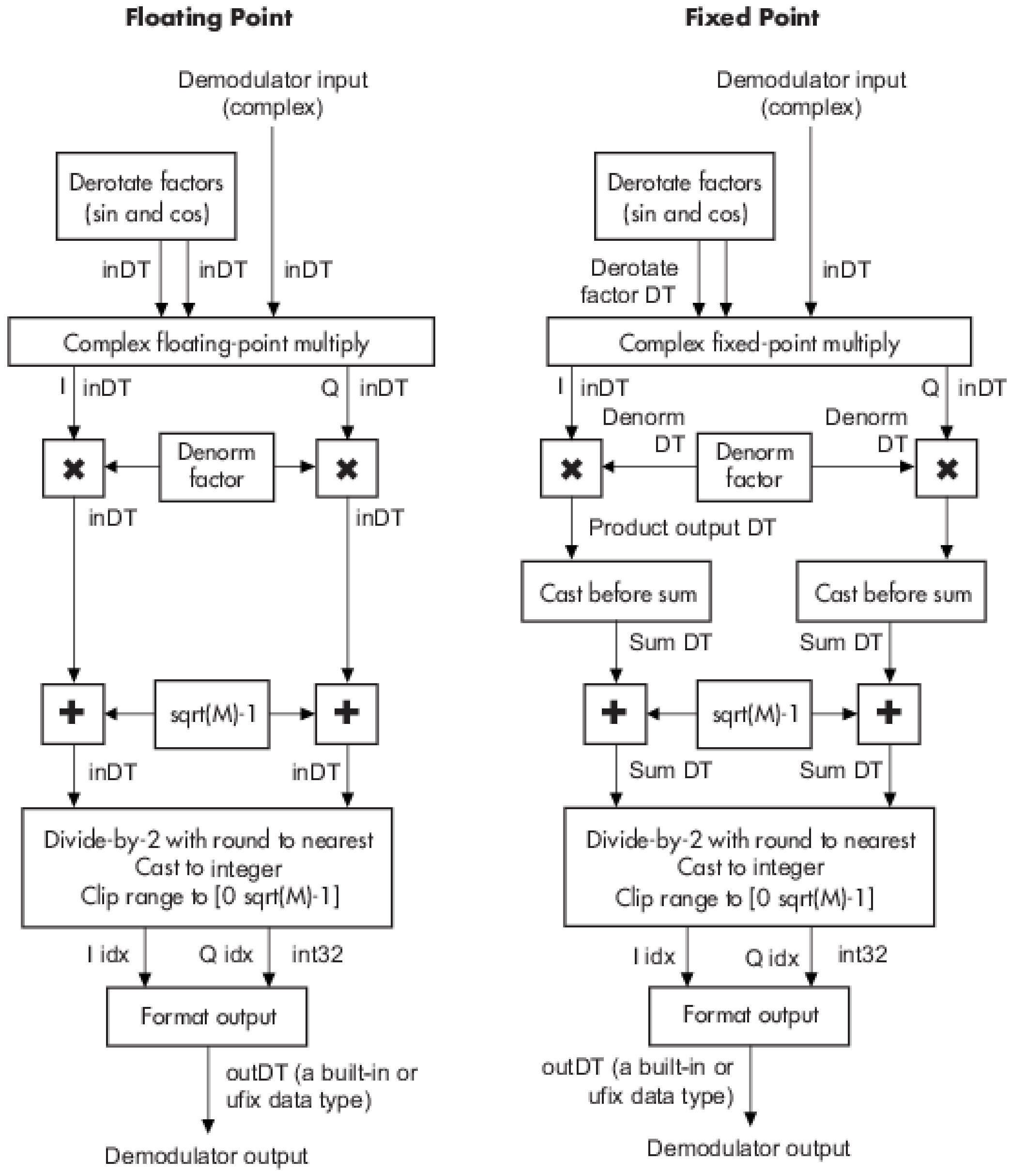
The demodulator algorithm maps received input signal constellation values to M-ary integer I and Q symbol indices between 0 and $\sqrt{M} - 1$ and then maps these demodulated symbol indices to formatted output values.

The integer symbol index computation is performed by first derotating and scaling the complex input signal constellation (possibly with noise) by a derotate factor and denormalization factor, respectively. These factors are derived from the **Phase offset**, **Normalization method**, and related parameters. These derotated and denormalized values are added to $\sqrt{M} - 1$ to translate them into an approximate range between 0 and $2 \times (\sqrt{M} - 1)$ (plus noise). The resulting values are then rescaled via a divide-by-two (or, equivalently, a right-shift by one bit for fixed-point operation) to obtain a range approximately between 0 and $\sqrt{M} - 1$ (plus noise) for I and Q. The noisy index values are rounded to the nearest integer and clipped, via saturation, and mapped to integer symbol values in the range [0 M-1]. Finally, based on other block parameters, the integer index is mapped to a symbol value that is formatted and cast to the selected **Output data type**.

The following figures contains signal flow diagrams for floating-point and fixed-point algorithm operation. The floating-point diagrams apply when the input signal data type is `double` or `single`. The fixed-point diagrams apply when the input signal is a signed fixed-point data type. Note that the diagram is simplified when **Phase offset** is a multiple of π/2, and/or the derived denormalization factor is 1.


Floating Point
Demodulator input (complex)
inDT
Pass through or negate I and Q (with saturation)
I inDT
Q inDT
sqrt(M)-1
inDT
inDT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output
Fixed Point
Demodulator input (complex)
inDT
Pass through or negate I and Q (with saturation)
I inDT
Q inDT
Cast before sum
Cast before sum
Sum DT
Sum DT
sqrt(M)-1
Sum DT
Sum DT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output

Signal-Flow Diagrams with Trivial Phase Offset and Denormalization Factor Equal to 1


Floating Point
Demodulator input (complex)
Derotate factors (sin and cos)
inDT
inDT
inDT
Complex floating-point multiply
I
inDT
Q
inDT
Denorm factor
inDT
inDT
sqrt(M)-1
inDT
inDT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output
Fixed Point
Demodulator input (complex)
Derotate factors (sin and cos)
Derotate factor DT
inDT
Complex fixed-point multiply
I
inDT
Q
inDT
Denorm DT
Denorm DT
Denorm factor
Product output DT
Cast before sum
Cast before sum
Sum DT
Sum DT
sqrt(M)-1
Sum DT
Sum DT
Divide-by-2 with round to nearest
Cast to integer
Clip range to [0 sqrt(M)-1]
I idx
Q idx
int32
Format output
outDT (a built-in or ufix data type)
Demodulator output

Signal-Flow Diagrams with Nontrivial Phase Offset and Nonunity Denormalization Factor

Parameters

M-ary number

The number of points in the signal constellation. It must have the form 2^K for some positive integer K.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

This parameter appears when **Normalization method** is set to `Min. distance between symbols`.

The distance between two nearest constellation points.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Phase offset (rad)

The rotation of the signal constellation, in radians.

Constellation ordering

Determines how the block assigns binary words to points of the signal constellation. More details are on the reference page for the Rectangular QAM Modulator Baseband block.

Selecting `User-defined` displays the field **Constellation mapping**, allowing for user-specified mapping.

Constellation mapping

This parameter appears when `User-defined` is selected in the pull-down list **Constellation ordering**.

This is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point.

Output type

Determines whether the block produces integers or binary representations of integers.

If set to `Integer`, the block produces integers.

If set to `Bit`, the block produces a group of K bits, called a *binary word*, for each symbol, when **Decision type** is set to `Hard decision`. If **Decision type** is set to `Log-likelihood ratio` or `Approximate log-likelihood ratio`, the block outputs bitwise LLR and approximate LLR, respectively.

Decision type

This parameter appears when `Bit` is selected in the pull-down list **Output type**.

Specifies the use of hard decision, LLR, or approximate LLR during demodulation. See “Exact LLR Algorithm” and “Approximate LLR Algorithm” in the *Communications Toolbox User's Guide* for algorithm details.

Noise variance source

This parameter appears when `Approximate log-likelihood ratio` or `Log-likelihood ratio` is selected for **Decision type**.

When set to `Dialog`, the noise variance can be specified in the **Noise variance** field. When set to `Port`, a port appears on the block through which the noise variance can be input.

Noise variance

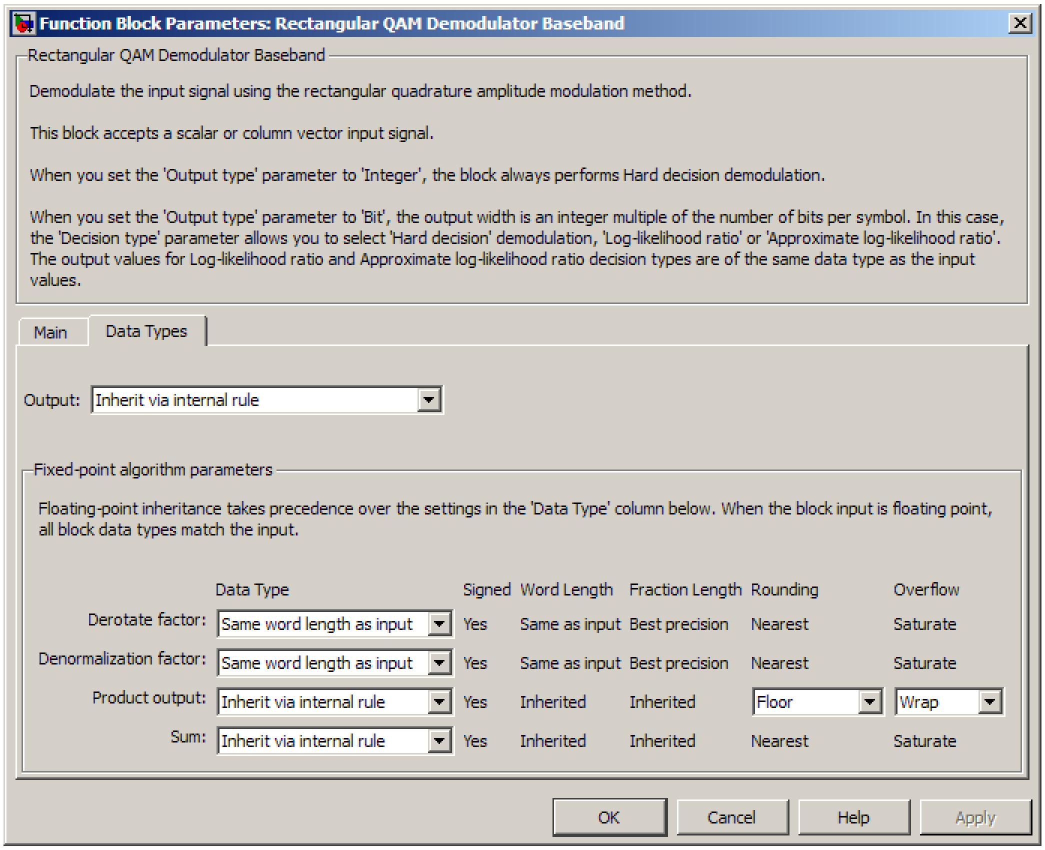
This parameter appears when the **Noise variance source** is set to `Dialog` and specifies the noise variance in the input signal. This parameter is tunable in normal mode, Accelerator mode and Rapid Accelerator mode.

If you use the Simulink Coder rapid simulation (RSIM) target to build an RSIM executable, then you can tune the parameter without recompiling the model. This is useful for Monte Carlo simulations in which you run the simulation multiple times (perhaps on multiple computers) with different amounts of noise.

The LLR algorithm involves computing exponentials of very large or very small numbers using finite precision arithmetic and would yield:

- `Inf` to `-Inf` if **Noise variance** is very high
- `NaN` if **Noise variance** and signal power are both very small

In such cases, use approximate LLR, as its algorithm does not involve computing exponentials.


Function Block Parameters: Rectangular QAM Demodulator Baseband
Main
Data Types
OK
Cancel
Help
Apply

Rectangular QAM Demodulator Baseband

Demodulate the input signal using the rectangular quadrature amplitude modulation method.

This block accepts a scalar or column vector input signal.

When you set the 'Output type' parameter to 'Integer', the block always performs Hard decision demodulation.

When you set the 'Output type' parameter to 'Bit', the output width is an integer multiple of the number of bits per symbol. In this case, the 'Decision type' parameter allows you to select 'Hard decision' demodulation, 'Log-likelihood ratio' or 'Approximate log-likelihood ratio'. The output values for Log-likelihood ratio and Approximate log-likelihood ratio decision types are of the same data type as the input values.

Output: Inherit via internal rule

Fixed-point algorithm parameters

Floating-point inheritance takes precedence over the settings in the 'Data Type' column below. When the block input is floating point, all block data types match the input.

	Data Type	Signed	Word Length	Fraction Length	Rounding	Overflow
Derotate factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Denormalization factor:	Same word length as input	Yes	Same as input	Best precision	Nearest	Saturate
Product output:	Inherit via internal rule	Yes	Inherited	Inherited	Floor	Wrap
Sum:	Inherit via internal rule	Yes	Inherited	Inherited	Nearest	Saturate

Output

When the parameter is set to `'Inherit via internal rule'` (default setting), the block will inherit the output data type from the input port. The output data type will be the same as the input data type if the input is of type `single` or `double`. Otherwise, the output data type will be as if this parameter is set to `'Smallest unsigned integer'`.

When the parameter is set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is the ideal minimum size, i.e., `ufix(1)` for bit outputs, and `ufix(ceil(log2(M)))` for integer outputs. For all other selections, it is an unsigned integer with the smallest available word length large enough to fit the ideal minimum size, usually corresponding to the size of a char (e.g., `uint8`).

For integer outputs, this parameter can be set to `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `single`, and `double`. For bit outputs, the options are `Smallest unsigned integer`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `boolean`, `single`, or `double`.

Derotate factor

This parameter only applies when the input is fixed-point and **Phase offset** is not a multiple of π/2.

This can be set to `Same word length as input` or `Specify word length`, in which case a field is enabled for user input.

Denormalization factor

This parameter only applies when the input is fixed-point and the derived denormalization factor is nonunity (not equal to 1). This scaling factor is derived from **Normalization method** and other parameter values in the block dialog.

This can be set to `Same word length as input` or `Specify word length`, in which case a field is enabled for user input. A best-precision fraction length is always used.

Product output

This parameter only applies when the input is a fixed-point signal and there is a nonunity (not equal to 1) denormalized factor. It can be set to `Inherit via internal rule` or `Specify word length`, which enables a field for user input.

Setting to `Inherit via internal rule` computes the full-precision product word length and fraction length. Internal Rule for Product Data Types in *DSP System Toolbox User's Guide* describes the full-precision **Product output** internal rule.

Setting to `Specify word length` allows you to define the word length. The block computes a best-precision fraction length based on the word length specified and the pre-computed worst-case (min/max) real world value **Product output** result. The worst-case **Product output** result is precomputed by multiplying the denormalized factor with the worst-case (min/max) input signal range, purely based on the input signal data type.

The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. For more information, see “Rounding Modes” or “Rounding Mode: Simplest” (Fixed-Point Designer).

Sum

This parameter only applies when the input is a fixed-point signal. It can be set to `Inherit via internal rule`, `Same as product output`, or `Specify word length`, in which case a field is enabled for user input

Setting to `Inherit via internal rule` computes the full-precision sum word length and fraction length, based on the two inputs to the Sum in the fixed-point Hard Decision Algorithm on page 5-719 signal flow diagram. The rule is the same as the fixed-point inherit rule of the internal **Accumulator data type** parameter in the Simulink Sum (Simulink) block.

Setting to `Specify word length` allows you to define the word length. A best precision fraction length is computed based on the word length specified in the pre-computed maximum range necessary for the demodulated algorithm to produce accurate results. The signed fixed-point data type that has the best precision fully contains the values in the range $2*(\sqrt{M}-1)$ for the specified word length.

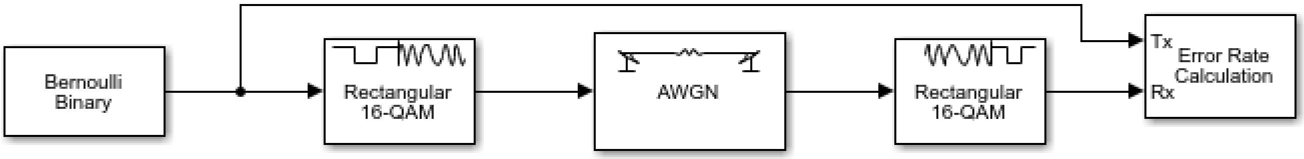
Setting to `Same as product output` allows the Sum data type to be the same as the **Product output** data type (when **Product output** is used). If the **Product output** is not used, then this setting will be ignored and the `Inherit via internal rule` Sum setting will be used.

Examples

Demodulate Noisy QAM Signal

Modulate and demodulate a noisy QAM signal.

Open the QAM demodulation model.



Run the simulation. The results are saved to the base workspace, where the variable `ErrorVec` is a 1-by-3 row vector. The BER is found in the first element.

Display the error statistics. For the Eb/No provided, 2 dB, the resultant BER is approximately 0.1. Your results may vary slightly.

```
ans =

    0.0948
```

Increase the Eb/No to 4 dB. Rerun the simulation, and observe that the BER has decreased.

```
ans =
0.0167
```

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Signed fixed–point when **M-ary number** is an even power of 2 and: • **Output type** is `Integer` • **Output type** is `Bit` and **Decision type** is `Hard-decision`
Var	• Double-precision floating point • Single-precision floating point
Output	• Double-precision floating point • Single-precision floating point • Boolean when **Output type** is `Bit` • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) in ASIC/FPGA when **Output type** is `Bit` • $ufix(\log_2 M)$ in ASIC/FPGA when **Output type** is `Integer`

Pair Block

Rectangular QAM Modulator Baseband

References

[1] Smith, Joel G., “Odd-Bit Quadrature Amplitude-Shift Keying,” *IEEE Transactions on Communications*, Vol. COM-23, March 1975, 385–389.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- The block does not support `single` or `double` data types for HDL code generation.
- HDL Coder supports the following **Output type** options:
  - `Integer`
  - `Bit` is supported only if the **Decision Type** is `Hard decision`.
- You must set **Normalization Method** to `Minimum Distance Between Symbols`, with a **Minimum distance** of 2.
- You must set **Phase offset (rad)** to a value that is a multiple of `pi`/4.

See Also

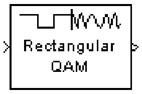
Blocks

General QAM Demodulator Baseband | Rectangular QAM Modulator Baseband

Introduced before R2006a

Rectangular QAM Modulator Baseband

Modulate using rectangular quadrature amplitude modulation



Library

AM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM Modulator Baseband block modulates using M-ary quadrature amplitude modulation with a constellation on a rectangular lattice. The output is a baseband representation of the modulated signal. This block accepts a scalar or column vector input signal. For information about the data types each block port supports, see “Supported Data Types” on page 5-733.

Note All values of power assume a nominal impedance of 1 ohm.

Integer-Valued Signals and Binary-Valued Signals

When you set the **Input type** parameter to `Integer`, the block accepts integer values between `0` and *M*-`1`. *M* represents the **M-ary number** block parameter.

When you set the **Input type** parameter to `Bit`, the block accepts binary-valued inputs that represent integers. The block collects binary-valued signals into groups of $K = \log_2(M)$ bits

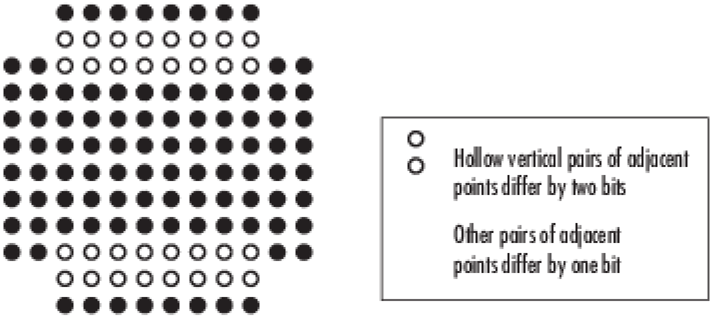
where

K represents the number of bits per symbol.

The input vector length must be an integer multiple of *K*. In this configuration, the block accepts a group of *K* bits and maps that group onto a symbol at the block output. The block outputs one modulated symbol for each group of *K* bits.

The **Constellation ordering** parameter indicates how the block assigns binary words to points of the signal constellation. Such assignments apply independently to the in-phase and quadrature components of the input:

- If **Constellation ordering** is set to `Binary`, the block uses a natural binary-coded constellation.
- If **Constellation ordering** is set to `Gray` and K is even, the block uses a Gray-coded constellation.
- If **Constellation ordering** is set to `Gray` and K is odd, the block codes the constellation so that pairs of nearest points differ in one or two bits. The constellation is cross-shaped, and the schematic below indicates which pairs of points differ in two bits. The schematic uses M = 128, but suggests the general case.


Hollow vertical pairs of adjacent points differ by two bits
Other pairs of adjacent points differ by one bit

For details about the Gray coding, see the reference page for the M-PSK Modulator Baseband block and the paper listed in References on page 5-734. Because the in-phase and quadrature components are assigned independently, the Gray and binary orderings coincide when M = 4.

Constellation Size and Scaling

The signal constellation has M points, where M is the **M-ary number** parameter. M must have the form 2^K for some positive integer K. The block scales the signal constellation based on how you set the **Normalization method** parameter. The following table lists the possible scaling conditions.

Value of Normalization Method Parameter	Scaling Condition
`Min. distance between symbols`	The nearest pair of points in the constellation is separated by the value of the **Minimum distance** parameter
`Average Power`	The average power of the symbols in the constellation is the **Average power** parameter
`Peak Power`	The maximum power of the symbols in the constellation is the **Peak power** parameter

Constellation Visualization

The Rectangular QAM Modulator Baseband block provides the capability to visualize a signal constellation from the block mask. This Constellation Visualization feature allows you to visualize a signal constellation for specific block parameters. For more information, see the Constellation Visualization section of the *Communications Toolbox User's Guide*.

Parameters

M-ary number

The number of points in the signal constellation. It must have the form 2^K for some positive integer K.

Input type

Indicates whether the input consists of integers or groups of bits.

Constellation ordering

Determines how the block maps each symbol to a group of output bits or integer.

Selecting `User-defined` displays the field **Constellation mapping**, which allows for user-specified mapping.

Constellation mapping

This parameter is a row or column vector of size M and must have unique integer values in the range [0, M-1]. The values must be of data type `double`.

The first element of this vector corresponds to the top-leftmost point of the constellation, with subsequent elements running down column-wise, from left to right. The last element corresponds to the bottom-rightmost point.

This field appears when `User-defined` is selected in the drop-down list **Constellation ordering**.

Normalization method

Determines how the block scales the signal constellation. Choices are `Min. distance between symbols`, `Average Power`, and `Peak Power`.

Minimum distance

The distance between two nearest constellation points. This field appears only when **Normalization method** is set to `Min. distance between symbols`.

Average power, referenced to 1 ohm (watts)

The average power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Average Power`.

Peak power, referenced to 1 ohm (watts)

The maximum power of the symbols in the constellation, referenced to 1 ohm. This field appears only when **Normalization method** is set to `Peak Power`.

Phase offset (rad)

The rotation of the signal constellation, in radians.

Output data type

The output data type can be set to `double`, `single`, `Fixed-point`, `User-defined`, or `Inherit via back propagation`.

Setting this parameter to `Fixed-point` or `User-defined` enables fields in which you can further specify details. Setting this parameter to `Inherit via back propagation`, sets the output data type and scaling to match the following block.

Output word length

Specify the word length, in bits, of the fixed-point output data type. This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter.

User-defined data type

Specify any signed built-in or signed fixed-point data type. You can specify fixed-point data types using the `sfix`, `sint`, `sfrac`, and `fixdt` functions from Fixed-Point Designer software. This parameter is only visible when you select `User-defined` for the **Output data type** parameter.

Set output fraction length to

Specify the scaling of the fixed-point output by either of the following methods:

- Choose `Best precision` to have the output scaling automatically set such that the output signal has the best possible precision.
- Choose `User-defined` to specify the output scaling in the **Output fraction length** parameter.

This parameter is only visible when you select `Fixed-point` for the **Output data type** parameter or when you select `User-defined` and the specified output data type is a fixed-point data type.

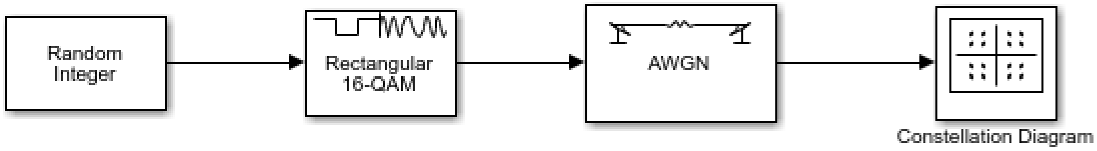
Output fraction length

For fixed-point output data types, specify the number of fractional bits, or bits to the right of the binary point. This parameter is only visible when you select `Fixed-point` or `User-defined` for the **Output data type** parameter and `User-defined` for the **Set output fraction length to** parameter.

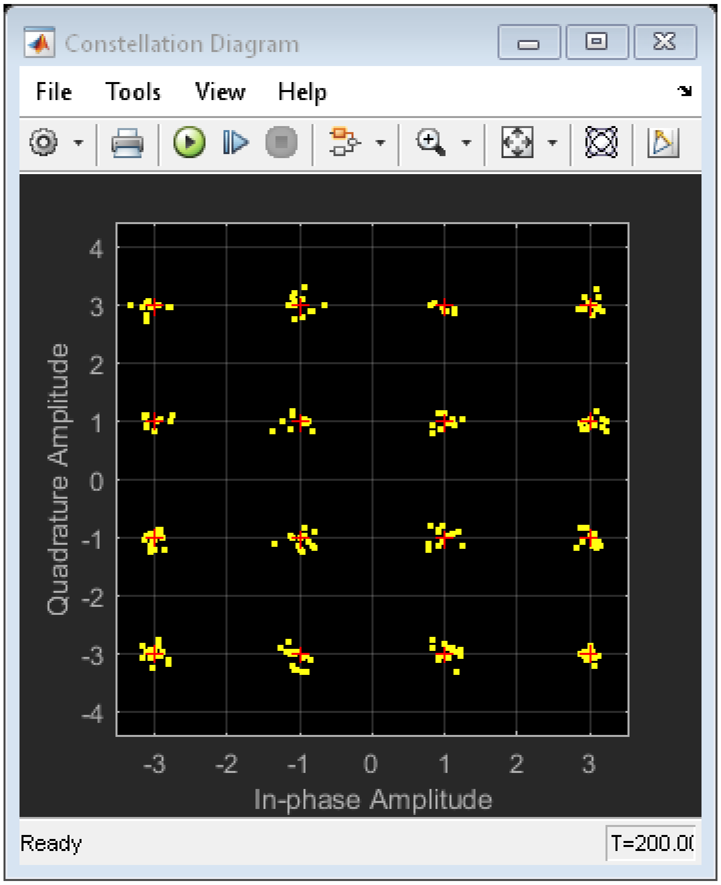
Examples

Plot Noisy 16-QAM Constellation

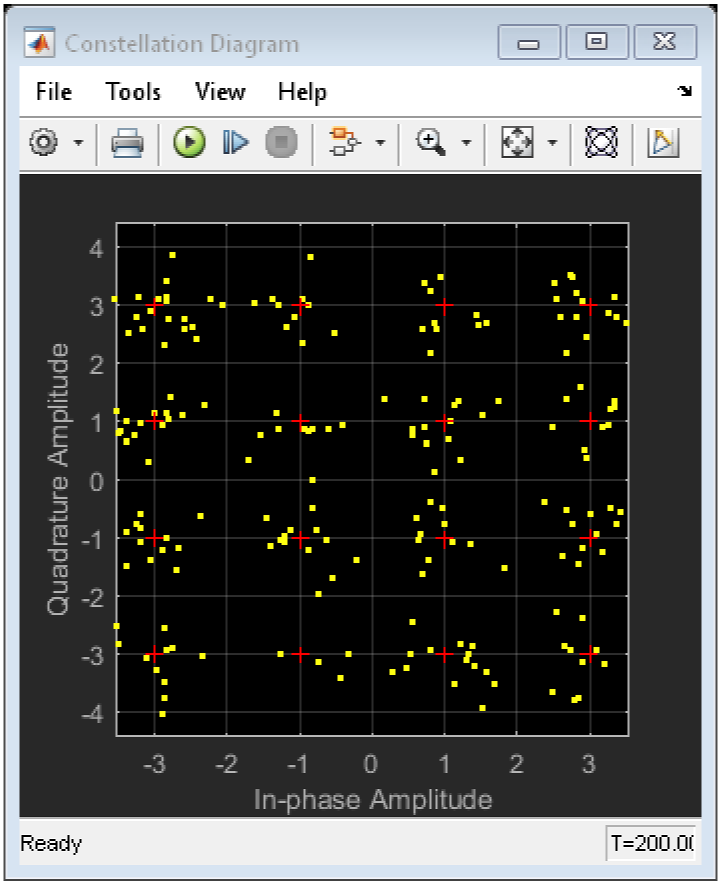
Open the 16-QAM model. The model generates a QAM signal, applies white noise, and displays the resulting constellation diagram.



Run the model.



Change the Eb/No of the AWGN Channel block from 20 dB to 10 dB. Observe the increase in the noise.



Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean when **Input type** is `Bit` • 8-, 16-, 32-bit signed integers • 8-, 16-, 32-bit unsigned integers • $ufix(\log_2 M)$ when **Input type** is `Integer`
Output	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Pair Block

Rectangular QAM Demodulator Baseband

References

[1] Smith, Joel G., “Odd-Bit Quadrature Amplitude-Shift Keying,” *IEEE Transactions on Communications*, Vol. COM-23, March 1975, 385–389.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

HDL Architecture

This block has a single, default HDL architecture.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).

Restrictions

- The block does not support `single` or `double` data types for HDL code generation.
- When **Input Type** is set to `Bit`, the block does not support HDL code generation for input types other than `boolean` or `ufix1`.

When the input type is set to `Bit`, but the block input is actually multibit (`uint16`, for example), the Rectangular QAM Modulator Baseband block does not support HDL code generation.

See Also

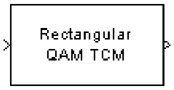
Blocks

General QAM Modulator Baseband | Rectangular QAM Demodulator Baseband

Introduced before R2006a

Rectangular QAM TCM Decoder

Decode trellis-coded modulation data, modulated using QAM method



Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM TCM Decoder block uses the Viterbi algorithm to decode a trellis-coded modulation (TCM) signal that was previously modulated using a QAM signal constellation.

The **M-ary number** parameter represents the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is the number of output bit streams from the convolutional encoder.)

The **Trellis structure** and **M-ary number** parameters in this block should match those in the Rectangular QAM TCM Encoder block, to ensure proper decoding.

Input and Output Signals

This block accepts a column vector input signal containing complex numbers. For information about the data types each block port supports, see “Supported Data Types” on page 5-736.

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the Rectangular QAM TCM Decoder block's output is a binary column vector with a length of *k* times the vector length of the input signal.

Operation Modes

The block has three possible methods for transitioning between successive frames. The **Operation mode** parameter controls which method the block uses. This parameter also affects the range of possible values for the **Traceback depth** parameter, D.

- In `Continuous` mode, the block initializes all state metrics to zero at the beginning of the simulation, waits until it accumulates D symbols, and then uses a sequence of D symbols to compute each of the traceback paths. D can be any positive integer. At the end of each frame, the block saves its internal state metric for use with the next frame.

  If you select **Enable the reset input**, the block displays another input port, labeled `Rst`. This port receives an integer scalar signal. Whenever the value at the `Rst` port is nonzero, the block resets all state metrics to zero and sets the traceback memory to zero.
- In `Truncated` mode, the block treats each frame independently. The traceback path starts at the state with the lowest metric. D must be less than or equal to the vector length of the input.

- In `Terminated` mode, the block treats each frame independently. The traceback path always starts at the all-zeros state. D must be less than or equal to the vector length of the input. If you know that each frame of data typically ends at the all-zeros state, then this mode is an appropriate choice.

Decoding Delay

If you set **Operation mode** to `Continuous`, then this block introduces a decoding delay equal to **Traceback depth***k bits, for a rate k/n convolutional code. The decoding delay is the number of zeros that precede the first decoded bit in the output.

The block incurs no delay for other values of **Operation mode**.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

M-ary number

The number of points in the signal constellation.

Traceback depth

The number of trellis branches (equivalently, the number of symbols) the block uses in the Viterbi algorithm to construct each traceback path.

Operation mode

The operation mode of the Viterbi decoder. Choices are `Continuous`, `Truncated`, and `Terminated`.

Enable the reset input port

When you select this check box, the block has a second input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data. This option appears only if you set **Operation mode** to `Continuous`.

Output data type

Select the data type for the block output signal as `boolean` or `single`. By default, the block sets this to `double`.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point
Reset	• Double-precision floating point • Boolean
Output	• Double-precision floating point • Boolean

Pair Block

Rectangular QAM TCM Encoder

References

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

[2] Proakis, John G., *Digital Communications*, Fourth edition, New York, McGraw-Hill, 2001.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

General TCM Decoder | Rectangular QAM TCM Encoder

Functions

`poly2trellis`

Introduced before R2006a

Rectangular QAM TCM Encoder

Convolutionally encode binary data and modulate using QAM method

Library

TCM, in Digital Baseband sublibrary of Modulation

Description

The Rectangular QAM TCM Encoder block implements trellis-coded modulation (TCM) by convolutionally encoding the binary input signal and mapping the result to a QAM signal constellation.

The **M-ary number** parameter is the number of points in the signal constellation, which also equals the number of possible output symbols from the convolutional encoder. (That is, $\log_2$(**M-ary number**) is equal to n for a rate k/n convolutional code.)

Input Signals and Output Signals

If the convolutional encoder described by the trellis structure represents a rate *k*/*n* code, then the Rectangular QAM TCM Encoder block's input must be a binary column vector with a length of *L*k* for some positive integer *L*.

The output from the Rectangular QAM TCM Encoder block is a complex column vector of length *L*.

Specifying the Encoder

To define the convolutional encoder, use the **Trellis structure** parameter. This parameter is a MATLAB structure whose format is described in “Trellis Description of a Convolutional Code”. You can use this parameter field in two ways:

- If you want to specify the encoder using its constraint length, generator polynomials, and possibly feedback connection polynomials, then use a `poly2trellis` command within the **Trellis structure** field. For example, to use an encoder with a constraint length of 7, code generator polynomials of 171 and 133 (in octal numbers), and a feedback connection of 171 (in octal), set the **Trellis structure** parameter to

  ```
  poly2trellis(7,[171 133],171)
  ```
- If you have a variable in the MATLAB workspace that contains the trellis structure, then enter its name as the **Trellis structure** parameter. This way is faster because it causes Simulink to spend less time updating the diagram at the beginning of each simulation, compared to the usage in the previous bulleted item.

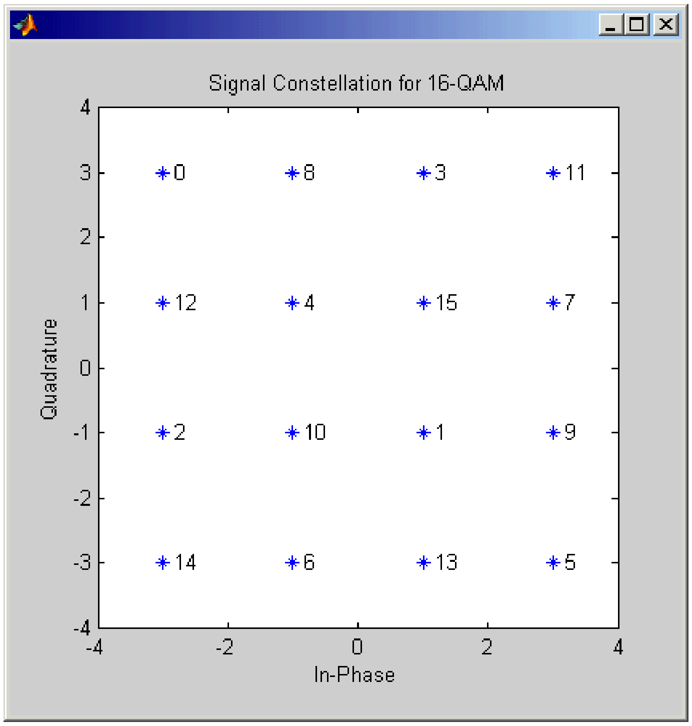
The encoder registers begin in the all-zeros state. You can configure the encoder so that it resets its registers to the all-zeros state during the course of the simulation. To do this, set the `Operation`

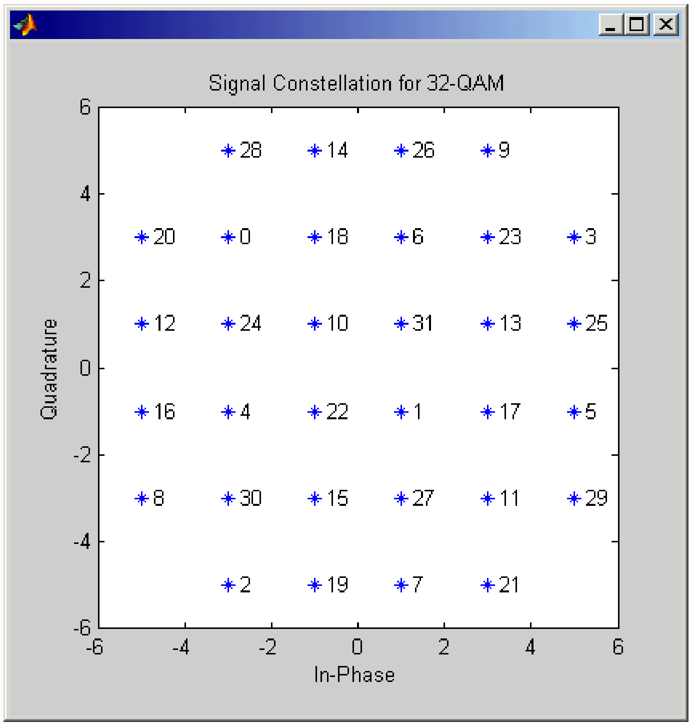
`mode` to **Reset on nonzero input via port**. The block then opens a second input port, labeled `Rst`. The signal at the `Rst` port is a scalar signal. When it is nonzero, the encoder resets before processing the data at the first input port.

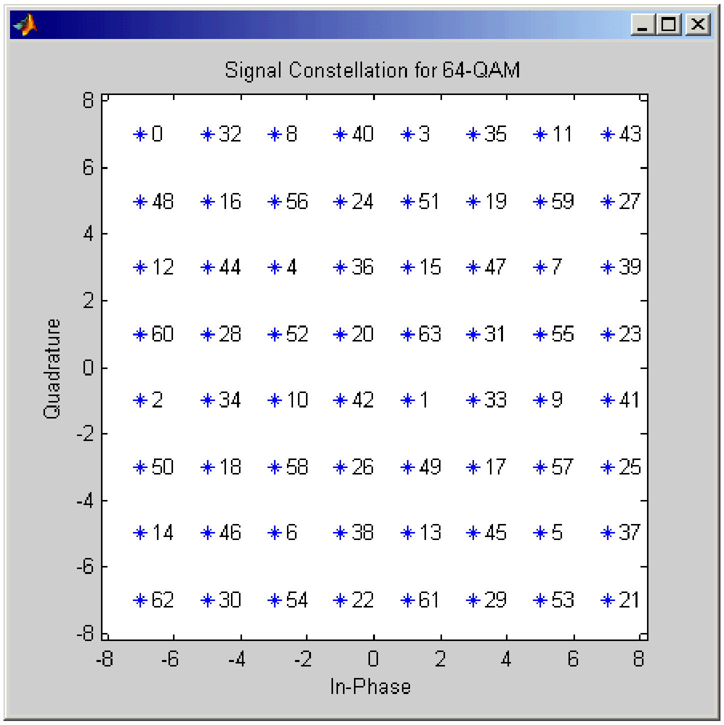
Signal Constellations

The trellis-coded modulation technique partitions the constellation into subsets called cosets, so as to maximize the minimum distance between pairs of points in each coset. This block internally forms a valid partition based on the value you choose for the **M-ary number** parameter.

The figures below show the labeled set-partitioned signal constellations that the block uses when **M-ary number** is `16`, `32`, and `64`. For constellations of other sizes, see Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.







Coding Gains

Coding gains of 3 to 6 decibels, relative to the uncoded case can be achieved in the presence of AWGN with multiphase trellis codes. For more information, see Biglieri, E., D. Divsalar, P. J. McLane and M. K. Simon, *Introduction to Trellis-Coded Modulation with Applications*, New York, Macmillan, 1991.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder.

Operation mode

In `Continuous` mode (default setting), the block retains the encoder states at the end of each frame, for use with the next frame.

In `Truncated (reset every frame)` mode, the block treats each frame independently. I.e., the encoder states are reset to all-zeros state at the start of each frame.

In `Terminate trellis by appending bits` mode, the block treats each frame independently. For each input frame, extra bits are used to set the encoder states to all-zeros state at the end of the frame. The output length is given by $y = n \cdot (x + s)/k$, where x is the number of input bits, and s = constraint length – 1 (or, in the case of multiple constraint lengths, s =`sum(ConstraintLength(i)-1)`). The block supports this mode for column vector input signals.

In `Reset on nonzero input via port` mode, the block has an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the encoder resets to the all-zeros state.

M-ary number

The number of points in the signal constellation.

Output data type

The output type of the block can be specified as a `single` or `double`. By default, the block sets this to `double`.

Pair Block

Rectangular QAM TCM Decoder

References

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

[2] Proakis, John G., Digital Communications, Fourth edition, New York, McGraw-Hill, 2001

[3] Ungerboeck, G., “Channel Coding with Multilevel/Phase Signals”, *IEEE Trans. on Information Theory*, Vol IT28, Jan. 1982, pp. 55–67.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
General TCM Encoder | Rectangular QAM TCM Decoder

Functions
`poly2trellis`

Introduced before R2006a

Rician Noise Generator

(To be removed) Generate Rician distributed noise

Note Rician Noise Generator will be removed in a future release. Use the MATLAB Function block and `randn` function instead.

Library

Noise Generators sublibrary of Comm Sources

Description

The Rician Noise Generator block generates Rician distributed noise. The Rician probability density function is given by

$$f(x) = \begin{cases} \dfrac{x}{\sigma^2} I_0\left(\dfrac{mx}{\sigma^2}\right)\exp\left(-\dfrac{x^2 + m^2}{2\sigma^2}\right) & x \ge 0 \\ 0 & x < 0 \end{cases}$$

where:

- σ is the standard deviation of the Gaussian distribution that underlies the Rician distribution noise
- $m^2 = m_I^2 + m_Q^2$, where m_I and m_Q are the mean values of two independent Gaussian components
- I_0 is the modified 0th-order Bessel function of the first kind given by

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

Note that *m* and σ are *not* the mean value and standard deviation for the Rician noise.

You must specify the **Initial seed** for the random number generator. When it is a constant, the resulting noise is repeatable. The vector length of the Initial seed parameter should equal the number of columns in a frame-based output or the number of elements in a sample-based output. The set of numerical parameters above the **Initial seed** parameter in the dialog box can consist of vectors having the same length as the **Initial seed**, or scalars.

Initial Seed

The scalar **Initial seed** parameter initializes the random number generator that the block uses to generate its Rician-distributed complex random process. When multiple blocks in a model have the **Initial seed** parameter, you can choose different initial seeds for each block to ensure different random streams are used in each block. Set **Initial seed** to an integer value for repeatable results or use the `randi` function to randomize your results.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples**

per frame, and **Interpret vector parameters as 1-D** parameters. See “Sources and Sinks” in *Communications Toolbox User's Guide* for more details.

The number of elements in the **Initial seed** and **Sigma** parameters becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** and **Sigma** parameters becomes the shape of a sample-based two-dimensional output signal.

Parameters

Specification method

Either `K-factor` or `Quadrature components`.

Rician K-factor

$K = m^2/(2\sigma^2)$, where m is as in the Rician probability density function. This field appears only if **Specification method** is `K-factor`.

In-phase component (mean), Quadrature component (mean)

The mean values m_I and m_Q, respectively, of the Gaussian components. These fields appear only if **Specification method** is `Quadrature components`.

Sigma

The variable σ in the Rician probability density function.

Initial seed

The initial seed value for the random number generator.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame

The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D

If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type

The output can be set to `double` or `single` data types.

References

[1] Proakis, John G., *Digital Communications*, Third edition, New York, McGraw Hill, 1995.

See Also

Blocks

MATLAB Function | MIMO Fading Channel

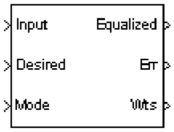
Functions
`isprime` | `randi` | `randn` | `rng`

Introduced before R2006a

RLS Decision Feedback Equalizer

(To be removed) Equalize using decision feedback equalizer that updates weights with RLS algorithm

Note will be removed in a future release. Use Decision Feedback Equalizer instead.



Library

Equalizers

Description

The RLS Decision Feedback Equalizer block uses a decision feedback equalizer and the RLS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the RLS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the weights are updated once every N^{th} sample, for a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Parameters

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps

The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of forward taps in the equalizer.

Forgetting factor

The forgetting factor of the RLS algorithm, a number between 0 and 1.

Inverse correlation matrix

The initial value for the inverse correlation matrix. The matrix must be N-by-N, where N is the total number of forward and feedback taps.

Initial weights

A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Compatibility Considerations

RLS Decision Feedback Equalizer will be removed

Warns starting in R2020a

- RLS Decision Feedback Equalizer will be removed in a future release. Use Decision Feedback Equalizer instead with the adaptive algorithm set to RLS.
- The **Enable training control input** parameter of the Decision Feedback Equalizer block is equivalent to the **Mode input port** parameter of the RLS Decision Feedback Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

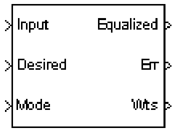
“Equalization”

Introduced before R2006a

RLS Linear Equalizer

(To be removed) Equalize using linear equalizer that updates weights using RLS algorithm

Note will be removed in a future release. Use Linear Equalizer instead.



Library

Equalizers

Description

The RLS Linear Equalizer block uses a linear equalizer and the RLS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the RLS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced (i.e. T-spaced) equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than `1`, the block updates the weights once every N^{th} sample, for a fractionally spaced (i.e. T/N-spaced) equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Parameters

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of taps in the equalizer.

Forgetting factor

The forgetting factor of the RLS algorithm, a number between 0 and 1.

Inverse correlation matrix

The initial value for the inverse correlation matrix. The matrix must be N-by-N, where N is the number of taps.

Initial weights

A vector that lists the initial weights for the taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, and for decision directed, the mode must be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current weights.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Haykin, Simon, *Adaptive Filter Theory*, Third Ed., Upper Saddle River, N.J., Prentice-Hall, 1996.

[3] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

[4] Proakis, John G., *Digital Communications*, Fourth Ed., New York, McGraw-Hill, 2001.

Compatibility Considerations

RLS Linear Equalizer will be removed

Warns starting in R2020a

- RLS Linear Equalizer will be removed in a future release. Use Linear Equalizer instead with the adaptive algorithm set to RLS.
- The **Enable training control input** parameter of the Linear Equalizer block is equivalent to the **Mode input port** parameter of the RLS Linear Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

“Equalization”

Introduced before R2006a

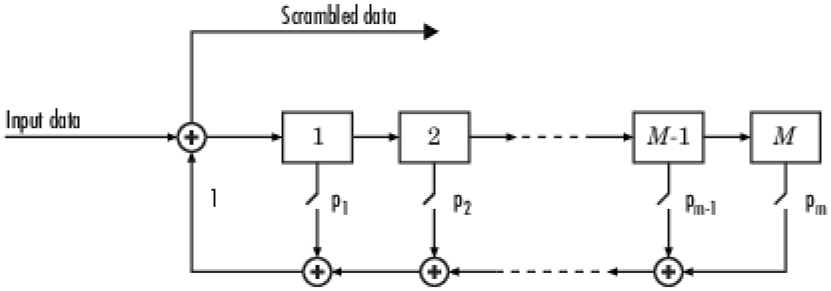
Scrambler

Scramble input signal
Library: Communications Toolbox / Sequence Operations

Description

The Scrambler block scrambles a scalar or column vector input signal.

One purpose of scrambling is to reduce the length of consecutive 0s or 1s in a transmitted signal. Long sequences of 0s or 1s can cause transmission synchronization problems. This schematic shows the scrambler operation. All adders perform addition modulo *N*, where *N* is the value specified by the Calculation base parameter.



At each time step, the input causes the contents of the registers to shift sequentially. Using the Scramble polynomial parameter, you specify the on or off state for each switch in the scrambler.

To achieve repeatable initial scrambler conditions, you can use one of these optional input ports:

- Select the Reset on nonzero input via port parameter and reset the scrambler with Rst.
- Set the Initial states source parameter to `Input port` and provide the initial states with ISt.

This block can accept input sequences that vary in length during simulation. For more information about sequences that vary in length, see Variable-Size Signal Basics (Simulink).

Ports

Input

`in` — Input data signal
vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal. The input values must be integers from `0` to Calculation base – `1`.

Data Types: `double`

`Rst` — Reset scrambler

scalar

Reset scrambler, specified as a scalar. The scrambler is reset if a nonzero input is applied to the port.

Dependencies

To enable this port, set Initial states source to `Dialog Parameter` and select Reset on nonzero input via port.

`ISt` — Initial states

vector

Initial states of the scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **ISt** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from `0` to Calculation base – `1`.

Dependencies

To enable this port, set Initial states source to `Input port`.

Output

`Out1` — Output scrambled data

vector

Output scrambled data, returned as an N_S-by-1 vector. N_S equals the number of samples in the input signal.

Data Types: `double`

Parameters

`Calculation base` — Calculation base

4 (default) | nonnegative integer

Calculation base used in the scrambler for modulo operations, specified as a nonnegative integer. The input and output of this block are integers from `0` to **Calculation base** – `1`.

`Scramble polynomial` — Polynomial that defines connections in scrambler

`'1 + z^-1 + z^-2 + z^-4'` (default) | character vector | integer vector | binary vector

Polynomial that defines the connections in the scrambler, specified as a character vector, integer vector, or binary vector. The **Scramble polynomial** parameter defines if each switch in the scrambler is on or off. Specify the polynomial as:

- A character vector, such as `'1 + z^-6 + z^-8'`. For more details on specifying polynomials in this way, see Character Representation of Polynomials.
- An integer vector, such as `[0 -6 -8]`, listing the descrambler coefficients in order of descending powers of z^{-1}, where $p(z^{-1}) = 1 + p_1 z^{-1} + p_2 z^{-2} + \ldots$
- A binary vector, such as `[1 0 0 0 0 0 1 0 1]`, listing the powers of z that appear in the polynomial that has a coefficient of 1. In this case, the order of the scramble polynomial is one less than the binary vector length.

Example: '1 + z^-6 + z^-8', [0 -6 -8], and [1 0 0 0 0 0 1 0 1] all represent this polynomial:

$p(z^{-1}) = 1 + z^{-6} + z^{-8}$

Initial states source — Set the source for scrambler initial states

Dialog Parameter (default) | Input port

- Dialog Parameter – Specify scrambler initial states by using the Initial states parameter.
- Input port – Specify scrambler initial states by using the ISt port.

Initial states — Initial states of scrambler registers

[0 1 2 3] (default) | nonnegative integer vector

Initial states of scrambler registers when the simulation starts, specified as a nonnegative integer vector. The length of **Initial states** must equal the order of the Scramble polynomial parameter. The vector element values must be integers from 0 to Calculation base – 1.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

Reset on nonzero input via port — Reset scrambler via input port

off (default) | on

Select this parameter to reset the Scrambler block via input port Rst.

Dependencies

This parameter is available when Initial states source is set to Dialog Parameter.

Block Characteristics

Data Types	Boolean | double | integer
Multidimensional Signals	no
Variable-Size Signals	no

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Descrambler | PN Sequence Generator

Objects

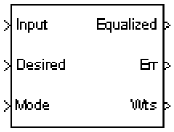
comm.Scrambler

Introduced before R2006a

Sign LMS Decision Feedback Equalizer

(To be removed) Equalize using decision feedback equalizer that updates weights with signed LMS algorithm

Note will be removed in a future release. Consider using Decision Feedback Equalizer instead.



Library

Equalizers

Description

The Sign LMS Decision Feedback Equalizer block uses a decision feedback equalizer and an algorithm from the family of signed LMS algorithms to equalize a linearly modulated baseband signal through a dispersive channel.

The supported algorithms, corresponding to the **Update algorithm** parameter, are

- `Sign LMS`
- `Sign Regressor LMS`
- `Sign Sign LMS`

During the simulation, the block uses the particular signed LMS algorithm to update the weights, once per symbol. If the **Number of samples per symbol** parameter is `1`, then the block implements a symbol-spaced equalizer; otherwise, the block implements a fractionally spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.

- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Parameters

Update algorithm

The specific type of signed LMS algorithm that the block uses to update the equalizer weights.

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps

The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol

The number of input samples for each symbol.

- When you set this parameter to `1`, the filter weights are updated once for each symbol, for a symbol spaced (i.e. T-spaced) equalizer.
- When you set this parameter to a value greater than `1`, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of forward taps in the equalizer.

Step size

The step size of the signed LMS algorithm.

Leakage factor

The leakage factor of the signed LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

Compatibility Considerations

Sign LMS Decision Feedback Equalizer will be removed

Warns starting in R2020a

- Sign LMS Decision Feedback Equalizer will be removed in a future release. Consider using Decision Feedback Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Decision Feedback Equalizer block is equivalent to the **Mode input port** parameter of the Sign LMS Decision Feedback Equalizer block.
- The Decision Feedback Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Sign LMS Decision Feedback Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

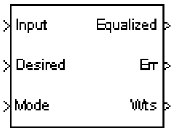
“Equalization”

Introduced before R2006a

Sign LMS Linear Equalizer

(To be removed) Equalize using linear equalizer that updates weights with signed LMS algorithm

Note will be removed in a future release. Consider using Linear Equalizer instead.



Library

Equalizers

Description

The Sign LMS Linear Equalizer block uses a linear equalizer and an algorithm from the family of signed LMS algorithms to equalize a linearly modulated baseband signal through a dispersive channel. The supported algorithms, corresponding to the **Update algorithm** parameter, are

- `Sign LMS`
- `Sign Regressor LMS`
- `Sign Sign LMS`

During the simulation, the block uses the particular signed LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a T/N-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.

- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Equalization".

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**) (5-1)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Parameters

Update algorithm

The specific type of signed LMS algorithm that the block uses to update the equalizer weights.

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of taps in the equalizer.

Step size

The step size of the signed LMS algorithm.

Leakage factor

The leakage factor of the signed LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that lists the initial weights for the taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current weights.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

[2] Kurzweil, Jack, *An Introduction to Digital Communications*, New York, Wiley, 2000.

Compatibility Considerations

Sign LMS Linear Equalizer will be removed

Warns starting in R2020a

- Sign LMS Linear Equalizer will be removed in a future release. Consider using Linear Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Linear Equalizer block is equivalent to the **Mode input port** parameter of the Sign LMS Linear Equalizer block.
- The Linear Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Sign LMS Linear Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

“Equalization”

Introduced before R2006a

SISO Fading Channel

Filter input signal through SISO multipath fading channel
Library: Communications Toolbox / Channels

Description

The SISO Fading Channel block filters an input signal using a single-input/single-output (SISO) multipath fading channel. This block models both Rayleigh and Rician fading. For processing details, see the Algorithms on page 5-768 section.

Ports

Input

`in` — Input data signal
vector

Input data signal, specified as an N_S-by-1 vector. N_S represents the number of samples in the input signal.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Out1` — Output data signal for fading channel
vector

Output data signal for the fading channel, returned as an N_S-by-1 vector. N_S represents the number of samples in the input signal.

`Gain` — Discrete path gains
matrix

Discrete path gains of the underlying fading process, returned as an N_S-by-N_P matrix.

- N_S represents the number of samples in the input signal.
- N_P represents the number of channel paths.

Dependencies

To enable this port, on the **Main** tab, select Output channel path gains.

`Delay` — Channel filter delay
scalar

Channel filter delay, returned as a scalar.

Dependencies

To enable this port, on the **Main** tab, select Output channel filter delay.

Parameters

Main Tab

Multipath parameters (frequency selectivity)

`Inherit sample rate from input` — Option to inherit the sample rate from input
on (default) | off

Select this parameter to use the sample rate of the input signal when processing. When `Inherit sample rate from input` is selected, the sample rate is N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

`Sample rate (Hz)` — Input signal sample rate
`1` (default) | positive scalar

Input signal sample rate in hertz, specified as a positive scalar. To match the model settings, set the sample rate to N_S/T_S, where N_S is the number of input samples, and T_S is the model sample time.

Dependencies

This parameter appears when Inherit sample rate from input is not selected.

Data Types: `double`

`Discrete path delays (s)` — Delays for each discrete path
`0` (default) | nonnegative scalar | row vector

Delays for each discrete path in seconds, specified as a nonnegative scalar or row vector.

- When you set **Discrete path delays (s)** to a scalar, the SISO channel is frequency flat.
- When you set **Discrete path delays (s)** to a vector, the SISO channel is frequency selective.

Data Types: `double`

`Average path gains (dB)` — Average gain for each discrete path
`0` (default) | scalar | row vector

Average gain for each discrete path in decibels, specified as a scalar or row vector. **Average path gains (dB)** must have the same size as Discrete path delays (s).

Data Types: `double`

`Normalize average path gains to 0 dB` — Option to normalize average path gains to 0 dB
on (default) | off

Select this parameter to normalize the fading processes so that the total power of the path gains, averaged over time, is 0 dB.

Fading distribution — Fading distribution of channel

Rayleigh (default) | Rician

Select the fading distribution of the channel, either Rayleigh or Rician.

K-factors — K-factor of Rician fading channel

3 (default) | positive scalar | row vector of nonnegative values

K-factor of a Rician fading channel, specified as a positive scalar or a 1-by-N_P vector of nonnegative values. N_P equals the value of the Discrete path delays (s) parameter.

- If you set **K-factors** to a scalar, the first discrete path is a Rician fading process with a Rician K-factor of **K-factors**. Any remaining discrete paths are independent Rayleigh fading processes.
- If you set **K-factors** to a row vector, the discrete path corresponding to a positive element of the **K-factors** vector is a Rician fading process with a Rician K-factor specified by that element. The discrete path corresponding to any zero-valued elements of the **K-factors** vector are Rayleigh fading processes. At least one element value must be nonzero.

Dependencies

This parameter appears when Fading distribution is Rician.

Data Types: double

LOS path Doppler shifts (Hz) — Doppler shifts for line-of-sight components

0 (default) | scalar | row vector

Doppler shifts for the line-of-sight components of the Rician fading channel in hertz, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path Doppler shifts (Hz)** to a scalar, it represents the line-of-sight component Doppler shift of the first discrete path that is a Rician fading process.
- If you set **LOS path Doppler shifts (Hz)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component Doppler shift specified by the elements of **LOS path Doppler shifts (Hz)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is Rician.

Data Types: double

LOS path initial phases (rad) — Initial phases for line-of-sight components

0 (default) | scalar | row vector

Initial phases for the line-of-sight component of the Rician fading channel in radians, specified as a scalar or row vector. This parameter must have the same size as K-factors.

- If you set **LOS path initial phases (rad)** to a scalar, it is the line-of-sight component initial phase of the first discrete path that is a Rician fading process.
- If you set **LOS path initial phases (rad)** to a row vector, the discrete path that is a Rician fading process has its line-of-sight component initial phase specified by the elements of **LOS path initial phases (rad)** that correspond to positive elements in the K-factors vector.

Dependencies

This parameter appears when Fading distribution is `Rician`.

Data Types: `double`

Doppler parameters (time dispersion)

`Maximum Doppler shift (Hz)` — Maximum Doppler shift for all channel paths

`0.001` (default) | nonnegative scalar

Maximum Doppler shift for all channel paths in hertz, specified as a nonnegative scalar.

Maximum Doppler shift (Hz) must be smaller than (f_s/10)/f_c for each path. f_s is the sampling rate at the input to the SISO Fading Channel block. f_c is the cutoff frequency factor of the path. For more information, see Cutoff Frequency Factor on page 5-517.

Data Types: `double`

`Doppler spectrum` — Doppler spectrum shape for all channel paths

`doppler('Jakes')` (default) | `doppler('Flat')` | `doppler('Rounded', ...)` | `doppler('Bell', ...)` | `doppler('Asymmetric Jakes', ...)` | `doppler('Restricted Jakes', ...)` | `doppler('Gaussian', ...)` | `doppler('BiGaussian', ...)`

Doppler spectrum shape for all channel paths, specified as a single Doppler spectrum structure returned from the `doppler` function or a 1-by-N_P cell array of such structures. The default value of this parameter is the Jakes Doppler spectrum (`doppler('Jakes')`).

- If you assign a single call to `doppler`, all paths have the same specified Doppler spectrum.
- If you assign a 1-by-N_P cell array of calls to `doppler` using any of the specified syntaxes, each path has the Doppler spectrum specified by the corresponding Doppler spectrum structure in the array. In this case, N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter applies when Maximum Doppler shift (Hz) is greater than zero.

Other parameters

`Initial seed` — Random number generator initial seed

`73` (default) | nonnegative integer

Random number generator initial seed for this block, specified as a nonnegative integer.

`Output channel path gains` — Option to output channel path gains

off (default) | on

Select this parameter to add the Gain output port to the block and output the channel path gains of the underlying fading process.

`Output channel filter delay` — Option to output channel filter delay

off (default) | on

Select this parameter to add the Delay output port to the block and output the channel filter delay of the underlying fading process.

`Simulate using` — Compilation type

`Interpreted execution` (default) | `Code generation`

Compilation type, specified as `Interpreted execution` or `Code generation`.

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has slower simulation speed than `Code generation`.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time but provides faster simulation speed than `Interpreted execution`.

Visualization Tab

`Channel visualization` — Select the channel visualization

`Off` (default) | `Impulse response` | `Frequency response` | `Doppler spectrum` | `Impulse and frequency responses`

Select the channel visualization: `Off`, `Impulse response`, `Frequency response`, `Doppler spectrum`, or `Impulse and frequency responses`. When visualization is on, the selected channel characteristics, such as impulse response or Doppler spectrum, display in a separate window. For more information, see Channel Visualization.

`Percentage of samples to display` — Percentage of samples to display

`25%` (default) | `10%` | `50%` | `100%`

Select the percentage of samples to display: `10%`, `25%`, `50%`, or `100%`. Increasing the percentage improves display accuracy at the expense of simulation speed.

Dependencies

This parameter appears when Channel visualization is `Impulse response`, `Frequency response`, or `Impulse and frequency responses`.

`Path for Doppler spectrum display` — Path for which Doppler spectrum is displayed

`1` (default) | positive integer

Path for which the Doppler spectrum is displayed, specified as a positive integer from 1 to N_P, where N_P equals the value of the Discrete path delays (s) parameter.

Dependencies

This parameter appears when Channel visualization is `Doppler spectrum`.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`yes`
Variable-Size Signals	`yes`

Algorithms

The fading process for the SISO channel is described in Methodology for Simulating Multipath Fading Channels.

Cutoff Frequency Factor

The cutoff frequency factor, f_c, is determined for different Doppler spectrum types.

- For any Doppler spectrum type other than Gaussian and biGaussian, f_c equals 1.
- For a `doppler('Gaussian')` spectrum type, f_c equals `NormalizedStandardDeviation` × $\sqrt{2\log 2}$.
- For a `doppler('BiGaussian')` spectrum type:
  - If the `PowerGains(1)` and `NormalizedCenterFrequencies(2)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - If the `PowerGains(2)` and `NormalizedCenterFrequencies(1)` field values are both `0`, then f_c equals `NormalizedStandardDeviation(2)` × $\sqrt{2\log 2}$.
  - If the `NormalizedCenterFrequencies` field value is `[0,0]` and the `NormalizedStandardDeviation` field has two identical elements, then f_c equals `NormalizedStandardDeviation(1)` × $\sqrt{2\log 2}$.
  - In all other cases, f_c equals 1.

References

[1] Oestges, C., and B. Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*. Academic Press, 2007.

[2] Correira, L. M. *Mobile Broadband Multimedia Networks: Techniques, Models and Tools for 4G*. Academic Press, 2006.

[3] Kermoal, J. P., L. Schumacher, K. I. Pedersen, P. E. Mogensen, and F. Frederiksen. "A stochastic MIMO radio channel model with experimental validation." *IEEE Journal on Selected Areas of Communications*. Vol. 20, Number 6, 2002, pp. 1211–1226.

[4] Jeruchim, M., P. Balaban, and K. S. Shanmugan. *Simulation of Communication Systems*. Second Edition. New York: Kluwer Academic/Plenum, 2000.

[5] Pätzold, Matthias, Cheng-Xiang Wang, and Bjorn Olav Hogstand. "Two New Sum-of-Sinusoids-Based Methods for the Efficient Generation of Multiple Uncorrelated Rayleigh Fading Waveforms." *IEEE Transactions on Wireless Communications*. Vol. 8, Number 6, 2009, pp. 3122–3131.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks
AWGN Channel | MIMO Fading Channel

Functions
`doppler`

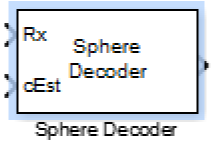
Objects
`comm.MIMOChannel`

Topics
Channel Visualization

Introduced in R2017b

Sphere Decoder

Decode input using a sphere decoder



Library

MIMO

Description

This block decodes the symbols sent over `Nt` antennas using the sphere decoding algorithm.

Data Type

For information about the data types each block port supports, see the “Supported Data Type” on page 5-771 table on this page. The output signal inherits the data type from the inputs.

Algorithm

This block implements the algorithm, inputs, and outputs described on the `comm.SphereDecoder` System object block reference page. The object properties correspond to the block parameters.

Parameters

Signal constellation

Specify the number of points in the signal constellation to which the bits are mapped. This value must be a complex column vector. The length of the vector must be a power of two. The block uses the same constellation for each transmit antenna. The default setting is a QPSK constellation with an average power of 1.

Bit mapping per constellation point

Specify the bit mapping that the block uses for each constellation point. This value must be a numerical matrix. he matrix size must be [ConstellationLength bitsPerSymbol], where ConstellationLength represents the length of the **Signal constellation** parameter value and bitsPerSymbol represents the number of bits that each symbol encodes. The default matrix size is [0 0; 0 1; 1 0; 1 1], which matches the default value of the **Signal constellation** property.

Initial search radius

Specify the initial search radius for the decoding algorithm as `Infinity` or `ZF solution`.

When you select `Infinity`, the block sets the initial search radius to Inf. When you select `ZF solution`, the block 'sets the initial search radius to the zero-forcing solution. The zero-forcing solution is calculated by the pseudo-inverse of the input channel when decoding. Large constellations and/or antenna counts can benefit from the initial reduction in the search radius. In most cases, however, the extra computation of the `ZF Solution` will not provide a benefit.

Decision method

Specify the decoding decision method as `Soft` or `Hard`. When you select `Soft` the block outputs log-likelihood ratios (LLRs), or soft bits. When you select set to `Hard`, the block converts the soft LLRs to bits. The hard decision output logical array follows the mapping of a `0` for a negative LLR and `1` for all other values.

Simulation using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

Supported Data Type

Port	Supported Data Types
Rx	• Double-precision floating point
cEst	• Double-precision floating point
Output	• Double-precision floating point • Boolean (Hard-decision method)

Limitations

- The output LLR values are not scaled by the noise variance. For coded links employing iterative coding (LDPC or turbo) or MIMO OFDM with Viterbi decoding, the output LLR values should be scaled by the channel state information to achieve better performance.

Algorithms

This block implements the algorithm, inputs, and outputs described on the Sphere Decoder System object reference page. The object properties correspond to the block parameters.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

OSTBC Combiner | OSTBC Encoder

Objects

`comm.SphereDecoder`

Introduced in R2013b

SSB AM Demodulator Passband

Demodulate SSB-AM-modulated data

Library

Analog Passband Modulation, in Modulation

Description

The SSB AM Demodulator Passband block demodulates a signal that was modulated using single-sideband amplitude modulation. The input is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The carrier frequency in the corresponding SSB AM Modulator Passband block.

Initial phase (rad)

The phase offset, θ, of the modulated signal.

Lowpass filter design method

The method used to generate the filter. Available methods are Butterworth, Chebyshev type I, Chebyshev type II, and Elliptic.

Filter order

The order of the lowpass digital filter specified in the **Lowpass filter design method** field .

Cutoff frequency

The cutoff frequency of the lowpass digital filter specified in the **Lowpass filter design method** field in Hertz.

Passband ripple

Applies to Chebyshev type I and Elliptic filters only. This is peak-to-peak ripple in the passband in dB.

Stopband ripple

Applies to Chebyshev type II and Elliptic filters only. This is the peak-to-peak ripple in the stopband in dB.

Pair Block

SSB AM Modulator Passband

See Also

Blocks

DSB AM Demodulator Passband | DSBSC AM Demodulator Passband | SSB AM Modulator Passband

Introduced before R2006a

SSB AM Modulator Passband

Modulate using single-sideband amplitude modulation

Library

Analog Passband Modulation, in Modulation

Description

The SSB AM Modulator Passband block modulates using single-sideband amplitude modulation with a Hilbert transform filter. The output is a passband representation of the modulated signal. Both the input and output signals are real scalar signals.

SSB AM Modulator Passband transmits either the lower or upper sideband signal, but not both. To control which sideband it transmits, use the **Sideband to modulate** parameter.

If the input is $u(t)$ as a function of time t, then the output is

$$u(t)\cos(f_c t + \theta) \mp \widehat{u}(t)\sin(f_c t + \theta)$$

where:

- f_c is the **Carrier frequency** parameter.
- θ is the **Initial phase** parameter.
- $\hat{u}(t)$is the Hilbert transform of the input $u(t)$.
- The minus sign indicates the upper sideband and the plus sign indicates the lower sideband.

Hilbert Transform Filter

This block uses the Analytic Signal block from the DSP System Toolbox Transforms block library.

The Analytic Signal block computes the complex analytic signal corresponding to each channel of the real M-by-N input, u

$$y = u + jH\{u\}$$

where $j = \sqrt{-1}$ and $H\{\}$denotes the Hilbert transform. The real part of the output in each channel is a replica of the real input in that channel; the imaginary part is the Hilbert transform of the input. In the frequency domain, the analytic signal retains the positive frequency content of the original signal while zeroing-out negative frequencies and doubling the DC component.

The block computes the Hilbert transform using an equiripple FIR filter with the order specified by the Filter order parameter, n. The linear phase filter is designed using the Remez exchange algorithm, and imposes a delay of n/2 on the input samples.

For best results, use a carrier frequency which is estimated to be larger than 10% of your input signal's sample rate. This is due to the implementation of the Hilbert transform by means of a filter.

In the following example, we sample a 10Hz input signal at 8000 samples per second. We then designate a Hilbert Transform filter of order 100. Below is the response of the Hilbert Transform filter as returned by `fvtool`.

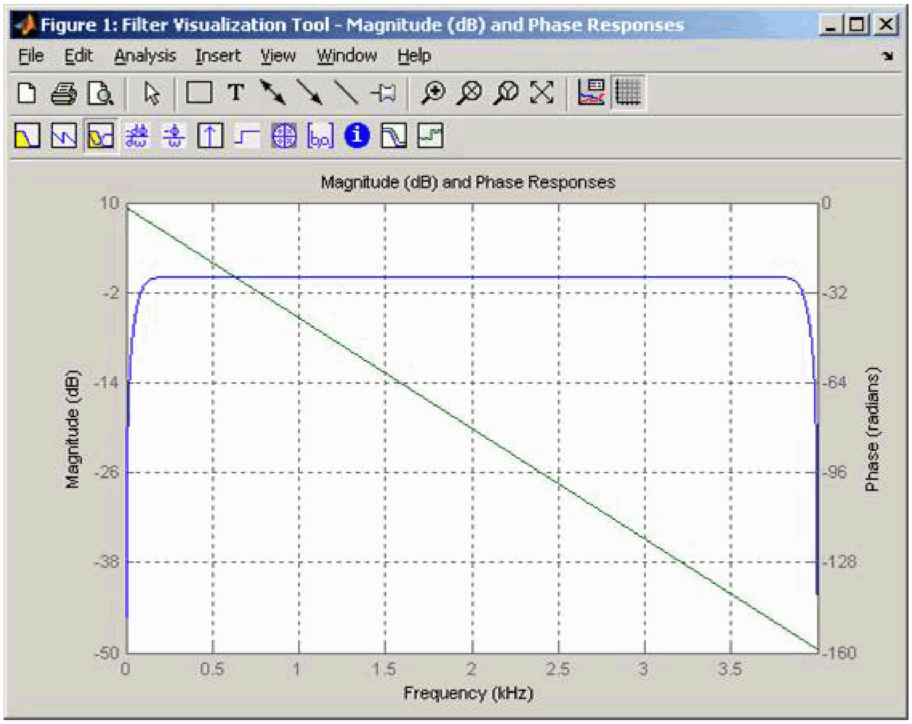


Figure 1: Filter Visualization Tool - Magnitude (dB) and Phase Responses
File Edit Analysis Insert View Window Help
Magnitude (dB) and Phase Responses
10
-2
-14
-26
-38
-50
Magnitude (dB)
0
-32
-64
-96
-128
-160
Phase (radians)
0
0.5
1
1.5
2
2.5
3
3.5
Frequency (kHz)

Note the bandwidth of the filter's magnitude response. By choosing a carrier frequency larger than 10% (but less than 90%) of the input signal's sample time (8000 samples per second, in this example) or equivalently, a carrier frequency larger than 400Hz, we ensure that the Hilbert Transform Filter will be operating in the flat section of the filter's magnitude response (shown in blue), and that our modulated signal will have the desired magnitude and form.

Typically, an appropriate **Carrier frequency** value is much higher than the highest frequency of the input signal. By the Nyquist sampling theorem, the reciprocal of the model's sample time (defined by the model's signal source) must exceed twice the **Carrier frequency** parameter.

This block works only with real inputs of type `double`. This block does not work inside a triggered subsystem.

Parameters

Carrier frequency (Hz)

The frequency of the carrier.

Initial phase (rad)

The phase offset, θ, of the modulated signal.

Sideband to modulate

This parameter specifies whether to transmit the upper or lower sideband.

Hilbert Transform filter order

The length of the FIR filter used to compute the Hilbert transform.

Pair Block

SSB AM Demodulator Passband

References

[1] Peebles, Peyton Z, Jr. *Communication System Principles*. Reading, Mass.: Addison-Wesley, 1976.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

DSB AM Modulator Passband | DSBSC AM Modulator Passband | SSB AM Demodulator Passband

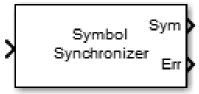
Functions

`hilbiir`

Introduced before R2006a

Symbol Synchronizer

Correct symbol timing clock skew
Library: Communications Toolbox / Synchronization



Symbol Synchronizer

Description

The Symbol Synchronizer block corrects symbol timing clock skew for PAM, PSK, QAM, or OQPSK modulation schemes between a single-carrier transmitter and receiver. For more information, see “Symbol Synchronization Overview” on page 5-779.

Note The input signal operates on a sample rate basis, while the output signal operates on a symbol rate basis.

Ports

Input

`samples` — Input samples
scalar (default) | column vector

Input samples, specified as a scalar or column vector of a PAM, PSK, QAM, or OQPSK modulated single-carrier signal. This port in unnamed on the block.

Data Types: `double` | `single`
Complex Number Support: Yes

Output

`Sym` — Output signal symbols
scalar | column vector

Output signal symbols, returned as a variable-size scalar or column vector that has the same data type as the input. For an input with dimensions of N_{samp}-by-1, the output at **Sym** has dimensions of N_{sym}-by-1. N_{sym} is approximately equal to N_{samp} divided by the N_{sps}. N_{sps} is equal to the **Samples per symbol** parameter. If the output exceeds the maximum output size of $\left\lceil \frac{N_{\mathrm{samp}}}{N_{\mathrm{sps}}} \times 1.1 \right\rceil$, it is truncated.

This port is unnamed when **Normalized timing error output port** is not selected.

`Err` — Estimated timing error
scalar | column vector

Estimated timing error for each input sample, returned as a scalar or column vector with values in the range [0, 1]. The estimated timing error is normalized by the input sample time. **Err** has the same data type and size as the input signal.

Dependencies

To enable this port, select **Normalized timing error output port**.

Parameters

`Modulation type` — Modulation type

`PAM/PSK/QAM` (default) | `OQPSK`

Modulation type, specified as `PAM/PSK/QAM`, or `OQPSK`.

`Timing error detector` — Type of timing error detector

`Zero-Crossing (decision-directed)` (default) | `Gardner (non-data-aided)` | `Early-Late (non-data-aided)` | `Mueller-Muller (decision-directed)`

Type of timing error detector, specified as `Zero-Crossing (decision-directed)`, `Gardner (non-data-aided)`, `Early-Late (non-data-aided)`, or `Mueller-Muller (decision-directed)`. This parameter assigns the timing error detection scheme used in the synchronizer.

For more information, see "Timing Error Detection (TED)" on page 5-780.

`Samples per symbol` — Samples per symbol

`2` (default) | positive integer greater than 1

Samples per symbol, specified as a positive integer greater than 1.

Data Types: `double`

`Damping factor` — Damping factor of the loop filter

`1` (default) | positive scalar

Damping factor of the loop filter, specified as a positive scalar. For more information, see "Loop Filter" on page 5-783.

Tunable: Yes

Data Types: `double` | `single`

`Normalized loop bandwidth` — Normalized bandwidth of loop filter

`0.01` (default) | positive scalar less than 1

Normalized bandwidth of the loop filter, specified as a positive scalar less than 1. The loop bandwidth is normalized by the sample rate of the input signal. For more information, see "Loop Filter" on page 5-783.

Note To ensure that the symbol synchronizer locks, set the **Normalized loop bandwidth** parameter to a value less than `0.1`.

Tunable: Yes

Data Types: `double` | `single`

`Detector gain` — Phase detector gain

`2.7` (default) | positive scalar

Phase detector gain, specified as a positive scalar.

Tunable: Yes

Data Types: `double` | `single`

`Normalized timing error output port` — Enable normalized timing error output port
`on` (default) | `off`

Select this parameter to output normalized timing error data at the output port **Err**.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`yes`

More About

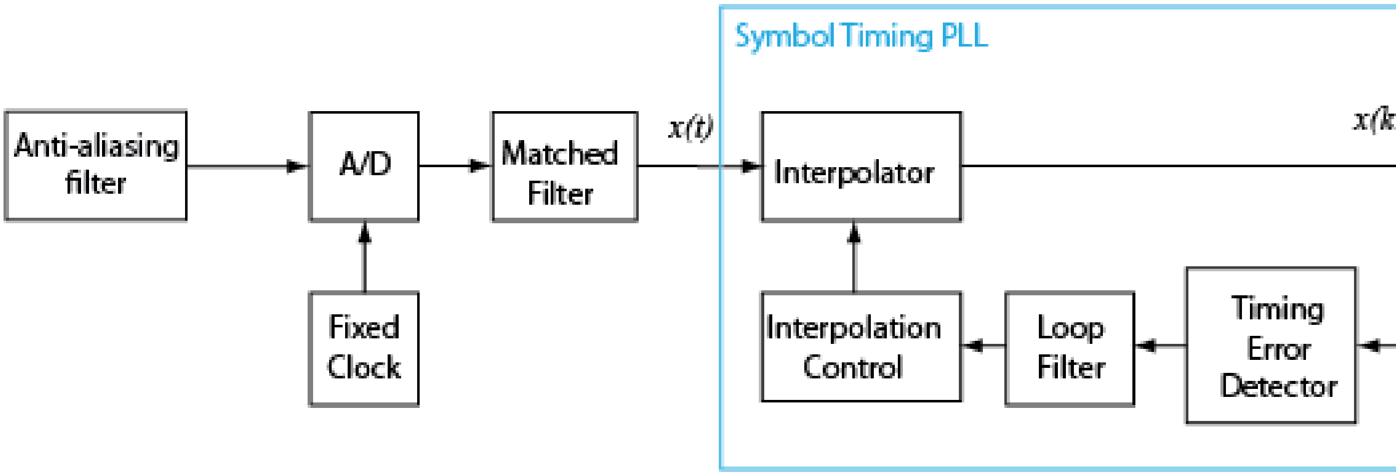
Symbol Synchronization Overview

The symbol timing synchronizer algorithm is based on a phased lock loop (PLL) algorithm that consists of four components:

- A timing error detector (TED)
- An interpolator
- An interpolation controller
- A loop filter

For OQPSK modulation, the in-phase and quadrature signal components are first aligned (as in QPSK modulation) using a state buffer to cache the last half symbol of the previous input. After initial alignment, the remaining synchronization process is the same as for QPSK modulation.

This block diagram shows an example of a timing synchronizer. In the figure, the symbol timing PLL operates on $x(t)$, the received sample signal after matched filtering. The symbol timing PLL outputs the symbol signal, $x(kT_s + \hat{\tau})$, after correcting for the clock skew between the transmitter and receiver.



Timing Error Detection (TED)

The symbol timing synchronizer supports non-data-aided TED and decision-directed TED methods. This table shows the timing estimate expressions for the TED method options.

TED Method	Expression
Zero-crossing (decision-directed)	$e(k) = x((k-1/2)T_s+\hat{\tau})[\hat{a}_0(k-1)-\hat{a}_0(k)] + y((k-1/2)T_s+\hat{\tau})[\hat{a}_1(k-1)-\hat{a}_1(k)]$
Gardner (non-data-aided)	$e(k) = x((k-1/2)T_s+\hat{\tau})[x((k-1)T_s+\hat{\tau}) - x(kT_s+\hat{\tau})] + y((k-1/2)T_s+\hat{\tau})[y((k-1)T_s+\hat{\tau}) - y(kT_s+\hat{\tau})]$
Early-late (non-data-aided)	$e(k) = x(kT_s+\hat{\tau})[x((k+1/2)T_s+\hat{\tau}) - x((k-1/2)T_s+\hat{\tau})] + y(kT_s+\hat{\tau})[y((k+1/2)T_s+\hat{\tau}) - y((k-1/2)T_s+\hat{\tau})]$
Mueller-Muller (decision-directed)	$e(k) = \hat{a}_0(k-1)x(kT_s+\hat{\tau}) - \hat{a}_0(k)x((k-1)T_s+\hat{\tau}) + \hat{a}_1(k-1)y(kT_s+\hat{\tau}) - \hat{a}_1(k)y((k-1)T_s+\hat{\tau})$

Non-data-aided TED uses received samples without any knowledge of the transmitted signal or the results of the channel estimation. Non-data-aided TED is used to estimate the timing error for signals with modulation schemes that have constellation points aligned with the in-phase or quadrature axis. Examples of signals suitable for the Gardner or early-late methods include QPSK-modulated signals with a zero phase offset that has points at {1+0*i*, 0+1*i*, -1+0*i*, 0−1*i*} and BPSK-modulated signals with a zero phase offset.

- **Gardner method** — The Gardner method is a non-data-aided feedback method that is independent of carrier phase recovery. It is used for baseband systems and modulated carrier systems. More specifically, this method is used for systems that use a linear modulation type with Nyquist pulses that have an excess bandwidth between approximately 40% and 100%. Examples include systems that use PAM, PSK, QAM, or OQPSK modulation and that shape the signal using raised cosine filters whose rolloff factor is between 0.4 and 1. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth increases (or rolloff factor increases in the case of a raised cosine filter). The Gardner method is similar to the early-late gate method.

- **Early-late method** — The early-late method is a non-data-aided feedback method. It is used for systems that use a linear modulation type such as PAM, PSK, QAM, or OQPSK modulation. For example, systems using a raised cosine filter with Nyquist pulses. In the presence of noise, the performance of this timing recovery method improves as the excess bandwidth of the pulse increases (or rolloff factor increases in the case of a raised cosine filter).

The early-late method is similar to the Gardner method. The Gardner method performs better in systems with high SNR values because it has lower self noise than the early-late method.

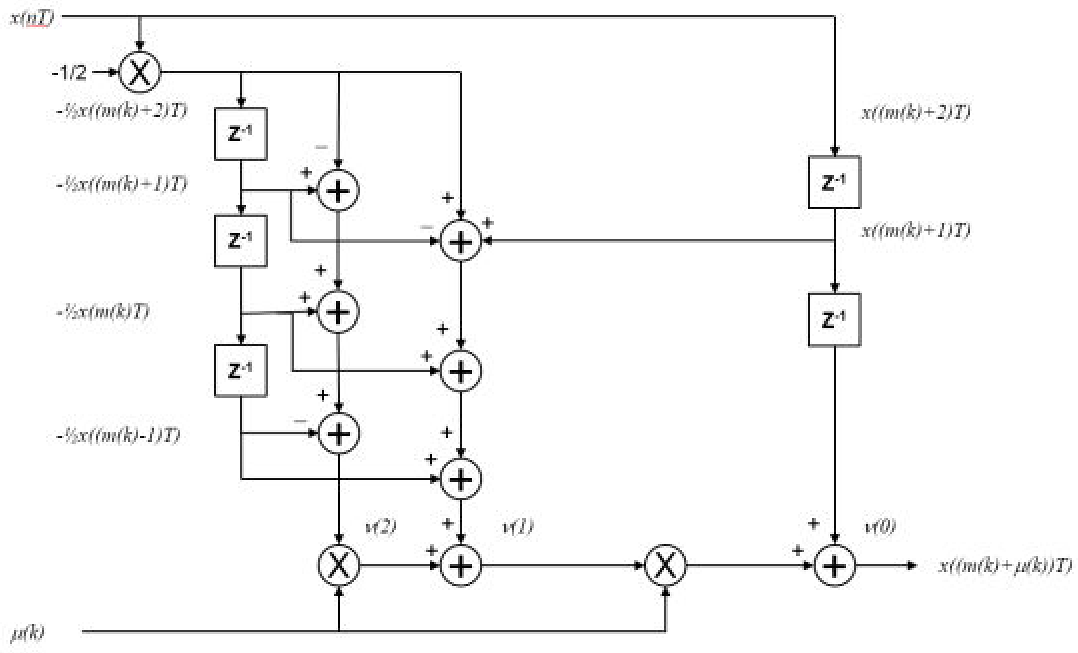
Decision-directed TED uses the `sign` function to estimate the in-phase and quadrature components of received samples, which result in lower computational complexity than non-data-aided TED.

- **Zero-crossing method** — The zero-crossing method is a decision-directed technique that requires 2 samples per symbol at the input to the synchronizer. It is used in low-SNR conditions for all values of excess bandwidth and in moderate-SNR conditions for moderate excess bandwidth factors in the approximate range [0.4, 0.6].
- **Mueller-Muller method** — The Mueller-Muller method is a decision-directed feedback method that requires prior recovery of the carrier phase. When the input signal has Nyquist pulses (for example, when using a raised cosine filter), the Mueller-Muller method has no self noise. For narrowband signaling in the presence of noise, the performance of the Mueller-Muller method improves as the excess bandwidth factor of the pulse decreases.

Because the decision-directed methods (zero-crossing and Mueller-Muller) estimate timing error based on the sign of the in-phase and quadrature components of signals passed to the synchronizer, they are not recommended for constellations that have points with either a zero in-phase or a quadrature component. $x(kT_S + \hat{\tau})$ and $y(kT_S + \hat{\tau})$ are the in-phase and quadrature components of the input signals to the timing error detector, where $\hat{\tau}$ is the estimated timing error. The Mueller-Muller method coefficients $\hat{a}_0(k)$ and $\hat{a}_1(k)$ are the estimates of $x(kT_S + \hat{\tau})$ and $y(kT_S + \hat{\tau})$. The timing estimates are made by applying the `sign` function to the in-phase and quadrature components and are used for only the decision-directed TED methods.

Interpolator

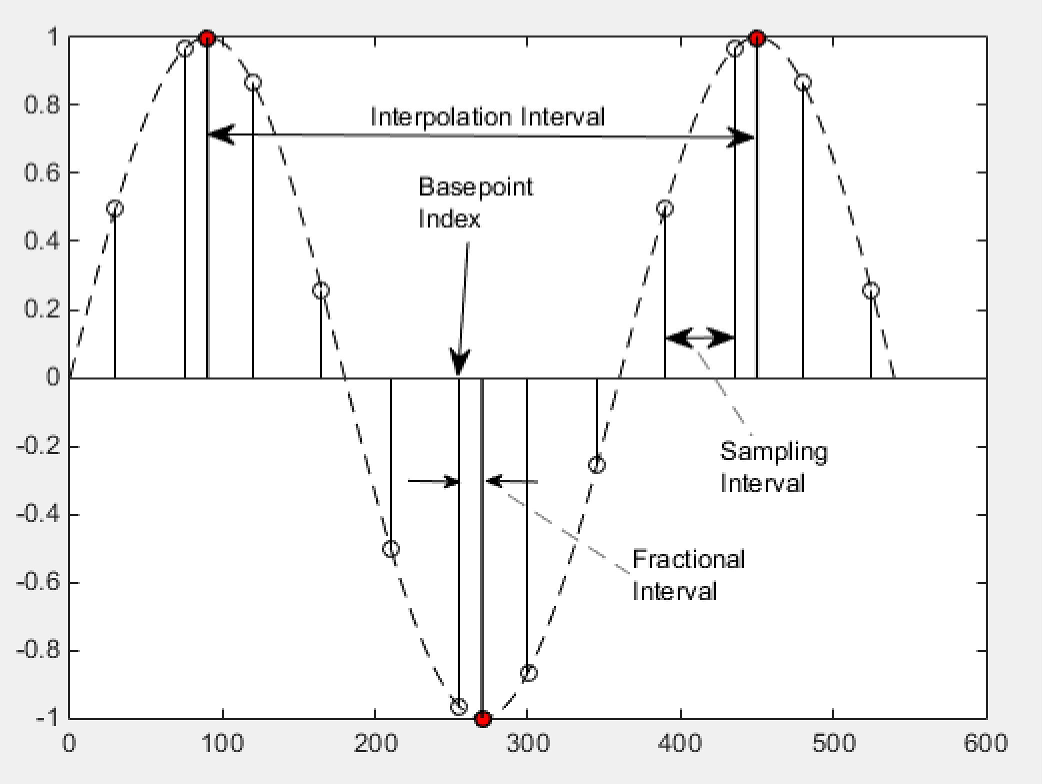
The time delay is estimated from the fixed-rate samples of the matched filter, which are asynchronous with the symbol rate. Because the resulting samples are not aligned with the symbol boundaries, an interpolator is used to "move" the samples. Because the time delay is unknown, the interpolator must be adaptive. Moreover, because the interpolant is a linear combination of the available samples, it can be thought of as the output of a filter.



The interpolator uses a piecewise parabolic interpolator with a Farrow structure and coefficient *α* set to 1/2 (see Rice, Michael, *Digital Communications: A Discrete-Time Approach*).

Interpolation Control

Interpolation control provides the interpolator with the basepoint index and fractional interval. The basepoint index is the sample index nearest to the interpolant. The fractional interval is the ratio of the time between the interpolant and its basepoint index and the interpolation interval.



Interpolation is performed for every sample, and a strobe signal is used to determine if the interpolant is output. The synchronizer uses a modulo-1 counter interpolation control to provide the strobe and the fractional interval for use with the interpolator.

Loop Filter

The synchronizer uses a proportional-plus integrator (PI) loop filter. The proportional gain, K_1, and the integrator gain, K_2, are calculated by

$$K_1 = \frac{-4\zeta\theta}{\left(1 + 2\zeta\theta + \theta^2\right)K_p}$$

and

$$K_2 = \frac{-4\theta^2}{\left(1 + 2\zeta\theta + \theta^2\right)K_p} .$$

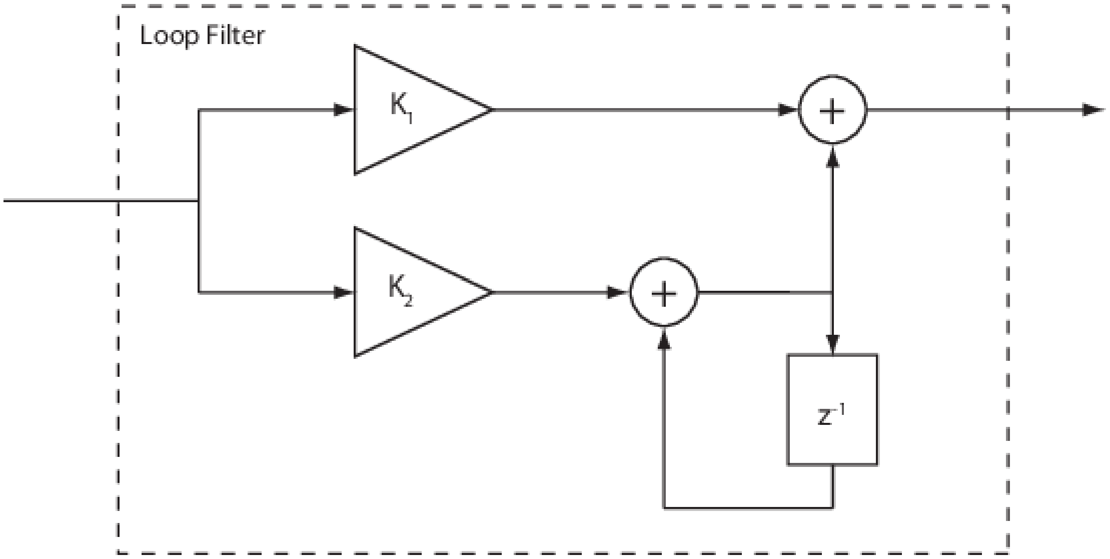
The interim term, θ, is given by

$$\theta = \frac{\frac{B_n T_s}{N}}{\zeta + \frac{1}{4\zeta}} ,$$

where:

- N is the number of samples per symbol.

- ζ is the damping factor.
- B_nT_s is the normalized loop bandwidth.
- K_p is the detector gain.



References

[1] Rice, Michael. *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Prentice Hall, 2008.

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

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Objects
`comm.SymbolSynchronizer`

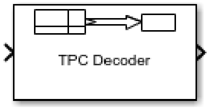
Blocks
Carrier Synchronizer

Introduced in R2015a

TPC Decoder

Turbo product code (TPC) decoder

Library: Communications Toolbox / Error Detection and Correction / Block



Description

The TPC Decoder block performs 2-D turbo product code (TPC) decoding of the soft input LLRs corresponding to the product code iteratively, using Chase-Pyndiah algorithm. The product code is a 2-D concatenation of linear block codes. The linear block code can be a parity check code, a Hamming code, or a BCH code capable of correcting two errors. Extended and shortened codes can be applied independently on each dimension. For a description of 2-D TPC decoding, see “Turbo Product Code Decoding” on page 5-790.

For information about valid code pairs and the error-correcting capability for each valid code pair, see “Component Code Pairs” on page 5-788.

Ports

Input

In — Log likelihood ratios

column vector

Log likelihood ratios, specified as a column vector.

- For full-length input messages, the length of the column vector is the product of Number of rows in code, Nr and Number of columns in code, Nc.
- For shortened input messages, the length of the column vector is the product of (N_R–K_R+S_R) and (N_C–K_C+S_C), where:
  - N_R is the value of Number of rows in code, Nr.
  - K_R is the value of Number of rows in message, Kr.
  - S_R is the value of Number of rows in shortened message, Sr.
  - N_C is the value of Number of columns in code, Nc.
  - K_C is the value of Number of columns in message, Kc.
  - S_C is the value of Number of columns in shortened message, Sc.

Data Types: `double` | `single`

Output

Out — TPC decoded message

column vector

TPC decoded message, returned as a column vector of binary values.

- For full-length input messages, the length of the column vector is the product of Number of rows in message, Kr and Number of columns in message, Kc.
- For shortened input messages, the length of the column vector is the product of Number of rows in shortened message, Sr and Number of columns in shortened message, Sc.

Data Types: `Boolean`

`Iter` — Actual number of decoding iterations

positive integer

Actual number of decoding iterations, returned as a positive integer.

Dependencies

To enable this port, select Output number of itereations executed.

Data Types: `double`

Parameters

Row TPC parameters

`Extended codes` — Extended codes indicator for TPC row parameters

`on` (default) | `off`

- When Extended codes is selected, the lists for Number of rows in code, Nr and Number of rows in message, Kr contain the valid values for extended individual code pairs (N_R,K_R).
- When Extended codes is cleared, the lists for Number of rows in code, Nr and Number of rows in message, Kr contain the valid values for nonextended individual code pairs (N_R,K_R).

`Number of rows in code, Nr` — Number of rows in product code matrix

`16` (default) | integer

Number of rows in the product code matrix, N_R. The list of integer values varies depending on the setting for Extended codes.

`Number of rows in message, Kr` — Number of rows in message matrix

`11` (default) | integer

Number of rows in the message matrix, K_R. The list of integer values varies depending on the setting for Extended codes and Number of rows in code, Nr.

`Specify shortened message length` — Specify shortened message length for rows

`off` (default) | `on`

Select **Specify shortened message length** to specify a value for Number of rows in shortened message, Sr.

`Number of rows in shortened message, Sr` — Number of rows in shortened message matrix

`9` (default) | integer

Number of rows in the shortened message matrix, S_R, specified as an integer less than or equal to K_R. When you specify this parameter, provide full-length N_R and K_R values to specify the (N_R,K_R) code pair. This code pair is then shortened to the (N_R-K_R+S_R,S_R) code pair, where:

- N_R is the value of Number of rows in code, Nr.
- K_R is the value of Number of rows in message, Kr.
- S_R is the value of Number of rows in shortened message, Sr.

Dependencies

To enable this parameter, select Specify shortened message length.

Data Types: `double`

Column TPC parameters

`Extended codes` — Extended codes indicator for TPC column parameters

`on` (default) | `off`

- When Extended codes is selected, the lists for Number of columns in code, Nc and Number of columns in message, Kc contain the valid values for extended individual code pairs (N_C,K_C).
- When Extended codes is cleared, the lists for Number of columns in code, Nc and Number of columns in message, Kc contain the valid values for nonextended individual code pairs (N_C,K_C).

`Number of columns in code, Nc` — Number of columns in product code matrix

`32` (default) | integer

Number of columns in the product code matrix, N_C. The list of integer values varies depending on the setting for Extended codes.

`Number of columns in message, Kc` — Number of columns in message matrix

`26` (default) | integer

Number of columns in the message matrix, K_C. The list of integer values varies depending on the setting for Extended codes and Number of columns in code, Nc.

`Specify shortened message length` — Specify shortened message length for columns

`off` (default) | `on`

Select **Specify shortened message length** to specify a value for Number of columns in shortened message, Sc.

`Number of columns in shortened message, Sc` — Number of columns in shortened message matrix

`22` (default) | integer

Number of columns in the shortened message matrix, S_C, specified as an integer. When you specify this parameter, provide full-length N_C and K_C values to specify the (N_C,K_C) code pair. This code pair is then shortened to the (N_C-K_C+S_C, S_C) code pair, where:

- N_C is the value of Number of columns in code, Nc.
- K_C is the value of Number of columns in message, Kc.
- S_C is the value of Number of columns in shortened message, Sc.

Dependencies

To enable this parameter, select Specify shortened message length.

Data Types: `double`

`Maximum number of iterations` — Maximum number of decoding iterations
`4` (default) | positive integer

Maximum number of decoding iterations, specified as a positive integer.

Data Types: `double`

`Stop iterating when code converges` — Stop decoding based on the calculated syndrome or parity-check of the component code
`on` (default) | `off`

Select **Stop iterating when code converges** to terminate decoding early if the calculated syndrome or parity-check of the component code evaluates to zero before Maximum number of iterations.

`Output number of iterations executed` — Output number of iterations executed
`off` (default) | `on`

Select this parameter to add the `Iter` output port and output the actual number of TPC decoding iterations performed.

`Simulate using` — Type of simulation to run
`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Component Code Pairs

This table lists the supported component code pairs for the row (N_R,K_R) and column (N_C,K_C) parameters.

- N_R and K_R represent the number of rows in the product code matrix and message matrix, respectively.
- N_C and K_C represent the number of columns in the product code matrix and message matrix, respectively.

Within each code type, any two component code pairs can form a 2-D TPC code. The table also includes the error-correction capability for each code pair.

Code type	**Component Code Pairs**(N_R,K_R) and (N_C,K_C)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1
	(15,11)	1
	(7,4)	1
Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2
	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-

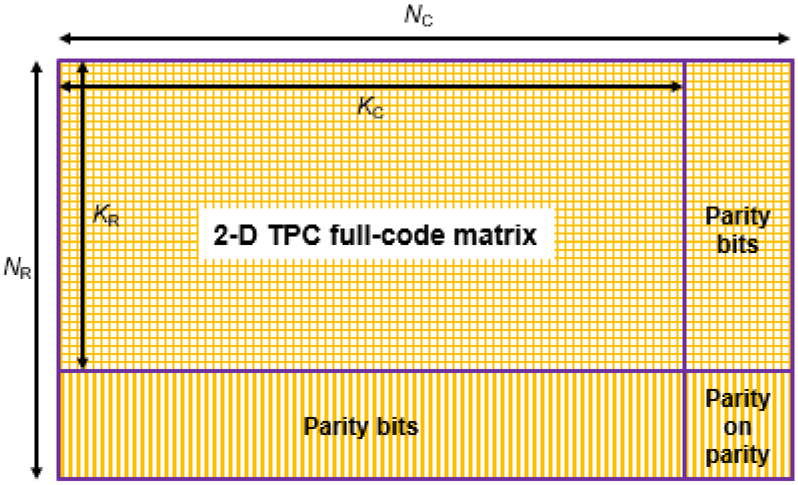
Turbo Product Code Decoding

Turbo product codes (TPC) are a form of concatenated codes used as forward error correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. The TPC decoder achieves near-optimum decoding of product codes using Chase decoding and the Pyndiah algorithm to perform iterative soft input, soft output decoding. For a detailed description, see [1] and [2]. This decoder implements an iterative soft input, soft output 2-D product code decoding, as described in [2], using two “Linear Block Codes”. The decoder expects the soft bit log likelihood ratios (LLRs) obtained from digital demodulation as the input signal.

The TPC decoder accepts either full-length or shortened codes.

TPC Decoding Full-Length Messages

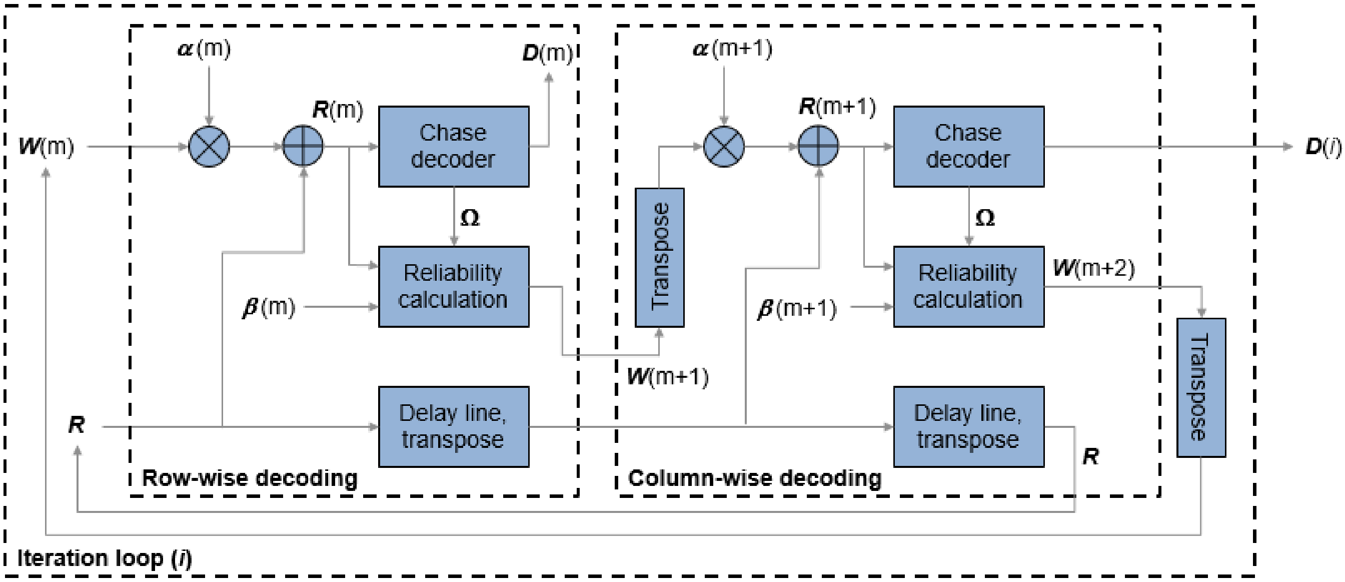
TPC encoded full-length input messages are decoded using specified 2-D TPC code pairs. Row-wise decoding uses the (N_C,K_C) code pair and column-wise decoding uses the (N_R,K_R) code pair. The input vector length must be N_R × N_C. To perform the 2-D TPC decoding, the column vector of the input LLRs, composed of the message and parity bits, is arranged into an N_R-by-N_C matrix.



The TPC decoder achieves near-optimum decoding of product codes using Chase decoding and the Pyndiah algorithm to perform iterative soft input, soft output decoding. Chase decoding forms a set of possible codewords for each row or column. The Pyndiah algorithm calculates soft information required for the next decoding step.

Iterative Soft Input, Soft Output Decoder

The iterative soft input, soft output decoding, as shown in the block diagram, carries out two decoding steps for each iteration.



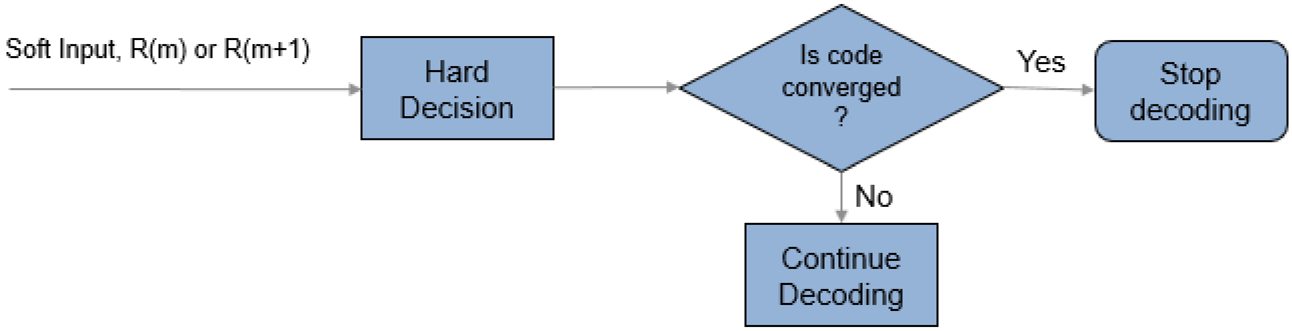
The soft inputs for decoding are $\boldsymbol{R}(m) = \boldsymbol{R} + \alpha(m)\boldsymbol{W}(m)$.

- Iteration loop counter i increments from $i = 1$ to the specified number of iterations.
- $m = 2i - 1$ is the decoding step index.
- $\boldsymbol{R}$ is the received LLR matrix.
- $\boldsymbol{R}(m)$ is the soft input for the mth decoding step.
- $\boldsymbol{W}(m)$ is the input extrinsic information for the mth decoding step.
- $\alpha(m) = [0,0.2,0.3,0.5,0.7,0.9,1,1, \ldots]$, where α is a weighting factor applied based on the decoding step index. For higher decoding steps, $\alpha = 1$.
- $\beta(m) = [0.2,0.4,0.6,0.8,1,1, \ldots]$, where β is a reliability factor applied based on the decoding step index. For higher decoding steps, $\beta = 1$.
- $\boldsymbol{D}$ contains the decoded message bits. The output message bits are formed from $\boldsymbol{D}$ by mapping –1 to 0 and +1 to 1, then reshaping the message block into a column vector.

The output message bits are formed after iterating through the specified number of iterations, or, if early termination is enabled, after code convergence.

Early Termination of TPC Decoding

If early termination is enabled, a code convergence check is performed on the hard decision of the soft input in each row-wise and column-wise decoding step. Early termination can be triggered after either the row-wise decoding or column-wise decoding converges.



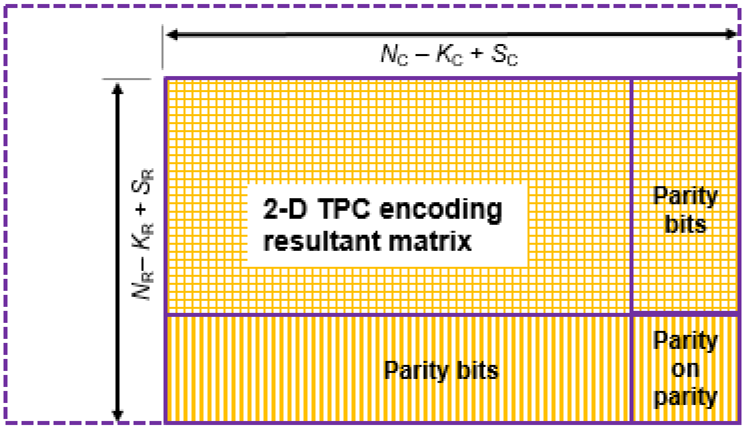
The code is converged if, for all rows or all columns,

- The syndrome evaluates to zero in the codes (Hamming codes, Extended Hamming codes, BCH codes, or Extended BCH codes).
- The parity check is evaluated to zero in parity check codes.

The reported number of iterations evaluates to the iteration value that is currently in progress. For example, if the code convergence check is satisfied after row-wise decoding in the third iteration (after 2.5 decoding steps), then the number of iteration returned is 3.

TPC Decoding Shortened Messages

TPC encoded shortened input messages are decoded using specified 2-D TPC code pairs. Row-wise decoding uses the (N_C - K_C + S_C, S_C) code pair and column-wise decoding uses the (N_R - K_R + S_R, S_R) code pair. The input vector length must be (N_R - K_R + S_R) × (N_C- K_C + S_C). To perform the 2-D TPC decoding of shortened messages, the column vector of the input LLRs, composed of the shortened message and parity bits, is arranged into an (N_R - K_R + S_R)-by-(N_C - K_C + S_C) matrix.



The TPC decoder processes the received shortened message LLRs similar to full length codes, with these exceptions:

- The shortened bit positions in the received codeword are set to –1.
- The Chase algorithm does not consider the shortened bit positions while choosing the least reliable bits.

References

[1] Chase, D. "Class of Algorithms for Decoding Block Codes with Channel Measurement Information." *IEEE Transactions on Information Theory*, Volume 18, Number 1, January 1972, pp. 170–182.

[2] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Vol. 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BCH Decoder | TPC Encoder

Functions

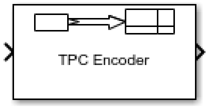
`tpcdec`

Introduced in R2018b

TPC Encoder

Turbo product code (TPC) encoder

Library: Communications Toolbox / Error Detection and Correction / Block



Description

The TPC Encoder block performs 2-D turbo product code (TPC) encoding of an input message. The product code is a 2-D concatenation of linear block codes. The linear block codes can be a parity check code, a Hamming code, or a BCH code capable of correcting two errors. Extended and shortened codes can be applied independently on each dimension. For a description of 2-D TPC encoding, see “Turbo Product Code Construction” on page 5-798.

For information about valid code pairs and the error-correcting capability for each valid code pair, see “Component Code Pairs” on page 5-797.

Ports

Input

In — Message to encode
column vector

Input message bits to encode, specified as a column vector.

- For full-length input messages, the length of the column vector must be the product of Number of rows in message, Kr and Number of columns in message, Kc.
- For shortened input messages, the length of the column vector must be the product of Number of rows in shortened message, Sr and Number of columns in shortened message, Sc.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16 | uint32 | Boolean

Output

Out — TPC-encoded message
column vector

TPC-encoded message, returned as a column vector with the same data type as the input signal.

- For full-length input messages, the length of the column vector is the product of Number of rows in code, Nr and Number of columns in code, Nc.
- For shortened input messages, the length of the column vector is the product of $(N_{\mathrm{R}}–K_{\mathrm{R}}+S_{\mathrm{R}})$ and $(N_{\mathrm{C}}–K_{\mathrm{C}}+S_{\mathrm{C}})$, where:
  - N_{R} is the value of Number of rows in code, Nr.

- K_R is the value of Number of rows in message, Kr.
- S_R is the value of Number of rows in shortened message, Sr.
- N_C is the value of Number of columns in code, Nc.
- K_C is the value of Number of columns in message, Kc.
- S_C is the value of Number of columns in shortened message, Sc.

Parameters

Row TPC parameters

`Extended codes` — Extended codes indicator for TPC row parameters

`on` (default) | `off`

- When Extended codes is selected, the lists for Number of rows in code, Nr and Number of rows in message, Kr contain the valid values for extended individual code pairs (N_R,K_R).
- When Extended codes is cleared, the lists for Number of rows in code, Nr and Number of rows in message, Kr contain the valid values for nonextended individual code pairs (N_R,K_R).

`Number of rows in code, Nr` — Number of rows in product code matrix

`16` (default) | integer

Number of rows in the product code matrix, N_R. The list of integer values varies depending on the setting for Extended codes.

`Number of rows in message, Kr` — Number of rows in message matrix

`11` (default) | integer

Number of rows in the message matrix, K_R. The list of integer values varies depending on the setting for Extended codes and Number of rows in code, Nr.

`Specify shortened message length` — Specify shortened message length for rows

`off` (default) | `on`

Select **Specify shortened message length** to specify a value for Number of rows in shortened message, Sr.

`Number of rows in shortened message, Sr` — Number of rows in shortened message matrix

`9` (default) | integer

Number of rows in the shortened message matrix, S_R, specified as an integer less than or equal to K_R. When you specify this parameter, provide full-length N_R and K_R values to specify the (N_R,K_R) code pair. This code pair is then shortened to the (N_R-K_R+S_R,S_R) code pair, where:

- N_R is the value of Number of rows in code, Nr.
- K_R is the value of Number of rows in message, Kr.
- S_R is the value of Number of rows in shortened message, Sr.

Dependencies

To enable this parameter, select Specify shortened message length.

Data Types: `double`

Column TPC parameters

`Extended codes` — Extended codes indicator for TPC column parameters

`on` (default) | `off`

- When Extended codes is selected, the lists for Number of columns in code, Nc and Number of columns in message, Kc contain the valid values for extended individual code pairs (N_C,K_C).
- When Extended codes is cleared, the lists for Number of columns in code, Nc and Number of columns in message, Kc contain the valid values for nonextended individual code pairs (N_C,K_C).

`Number of columns in code, Nc` — Number of columns in product code matrix

`32` (default) | integer

Number of columns in the product code matrix, N_C. The list of integer values varies depending on the setting for Extended codes.

`Number of columns in message, Kc` — Number of columns in message matrix

`26` (default) | integer

Number of columns in the message matrix, K_C. The list of integer values varies depending on the setting for Extended codes and Number of columns in code, Nc.

`Specify shortened message length` — Specify shortened message length for columns

`off` (default) | `on`

Select **Specify shortened message length** to specify a value for Number of columns in shortened message, Sc.

`Number of columns in shortened message, Sc` — Number of columns in shortened message matrix

`22` (default) | integer

Number of columns in the shortened message matrix, S_C, specified as an integer. When you specify this parameter, provide full-length N_C and K_C values to specify the (N_C,K_C) code pair. This code pair is then shortened to the (N_C–K_C+S_C, S_C) code pair, where:

- N_C is the value of Number of columns in code, Nc.
- K_C is the value of Number of columns in message, Kc.
- S_C is the value of Number of columns in shortened message, Sc.

Dependencies

To enable this parameter, select Specify shortened message length.

Data Types: `double`

`Simulate using` — Type of simulation to run

`Code generation` (default) | `Interpreted execution`

Type of simulation to run, specified as `Code generation` or `Interpreted execution`.

- `Code generation` –– Simulate the model by using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations unless the model changes. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.

- `Interpreted execution` –– Simulate the model by using the MATLAB interpreter. This option requires less startup time than the `Code generation` method, but the speed of subsequent simulations is slower. In this mode, you can debug the source code of the block.

Block Characteristics

Data Types	`Boolean` | `double` | `integer` | `single`
Multidimensional Signals	`no`
Variable-Size Signals	`no`

More About

Component Code Pairs

This table lists the supported component code pairs for the row (N_R,K_R) and column (N_C,K_C) parameters.

- N_R and K_R represent the number of rows in the product code matrix and message matrix, respectively.
- N_C and K_C represent the number of columns in the product code matrix and message matrix, respectively.

Within each code type, any two component code pairs can form a 2-D TPC code. The table also includes the error-correction capability for each code pair.

Code type	**Component Code Pairs**(N_R,K_R) and (N_C,K_C)	**Error-Correction Capability (*T*)**
Hamming code	(255,247)	1
	(127,120)	1
	(63,57)	1
	(31,26)	1
	(15,11)	1
	(7,4)	1
Extended Hamming code	(256,247)	1
	(128,120)	1
	(64,57)	1
	(32,26)	1
	(16,11)	1
	(8,4)	1
BCH code	(255,239)	2
	(127,113)	2
	(63,51)	2
	(31,21)	2

	(15,7)	2
Extended BCH code	(256,239)	2
	(128,113)	2
	(64,51)	2
	(32,21)	2
	(16,7)	2
Parity check code	(256,255)	-
	(128,127)	-
	(64,63)	-
	(32,31)	-
	(16,15)	-
	(8,7)	-
	(4,3)	-

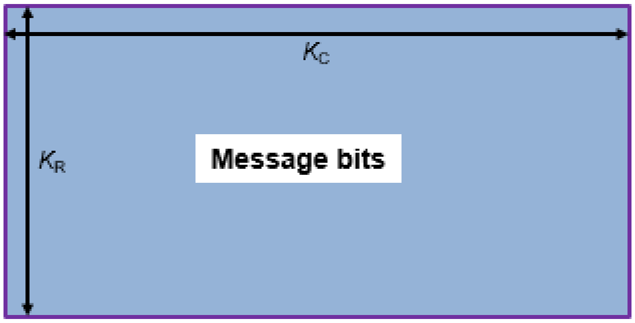
Turbo Product Code Construction

Turbo product codes (TPC) are a form of concatenated codes used as forward error-correcting (FEC) codes. Two or more component block codes, such as systematic linear block codes, are used to construct TPCs. This encoder implements 2-D product code encoding, as described in [1], using two “Linear Block Codes”.

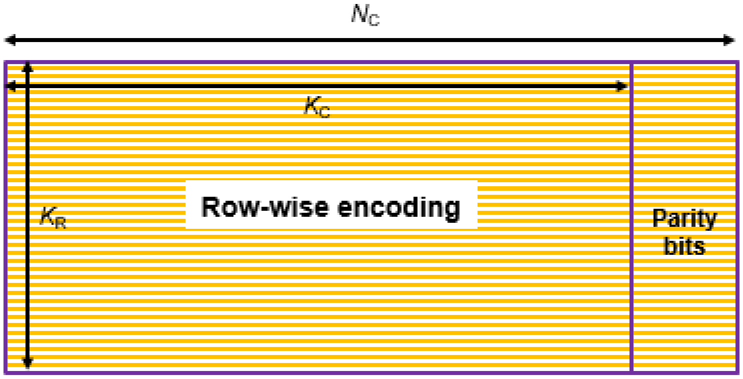
The TPC encoder accepts either full-length or shortened messages.

Construction of Full-Length Message Product Codes

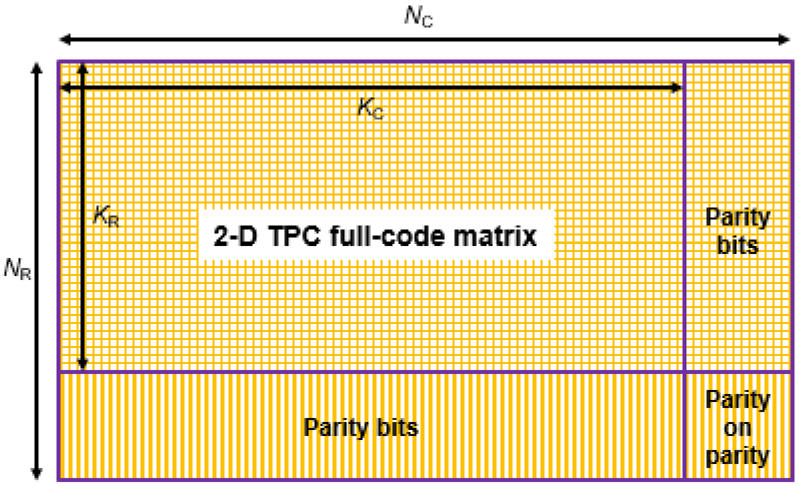
Full-length input messages are encoded using specified 2-D TPC code pairs. Row-wise encoding uses the (N_C,K_C) code pair and column-wise encoding uses the (N_R,K_R) code pair. The input vector length must be $K_R \cdot K_C$. The input message bits vector is arranged into a K_R-by-K_C matrix.



Row-wise encoding uses an (N_C,K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in a K_R-by-N_C matrix that includes parity bits added to each row.



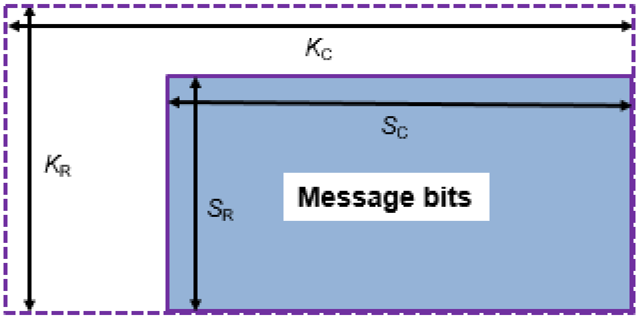
Next, column-wise encoding uses an (N_R,K_R) systematic linear block encoder on each of the N_C columns. Applying this 2-D TPC encoding to the initial K_R-by-K_C matrix results in an N_R-by-N_C matrix that includes parity bits added to each row and column.



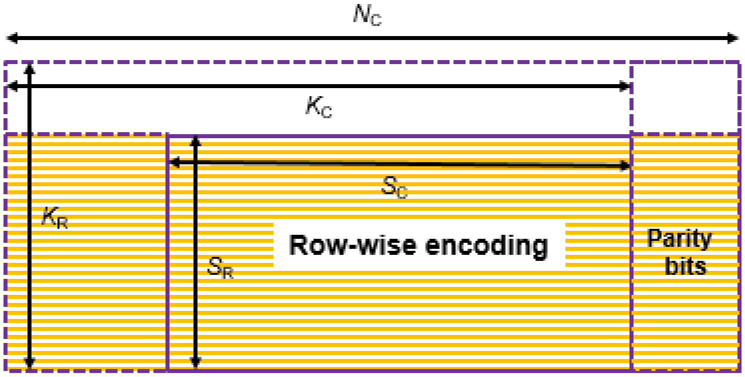
The 2-D TPC full-code matrix is reshaped into a column vector of length $N_R \cdot N_C$ and returned as the TPC-encoded output.

Construction of Shortened Message Product Codes

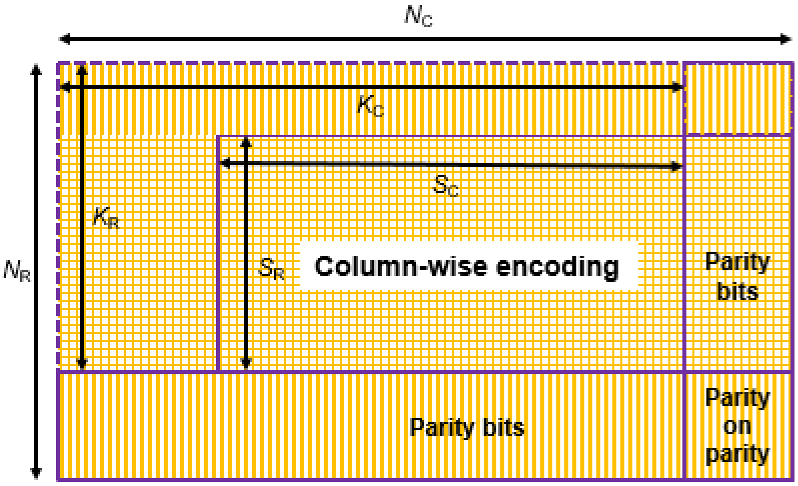
Shortened input messages are encoded using specified 2-D TPC code pairs. Row-wise encoding uses the (N_C,K_C) code pair and column-wise encoding uses an (N_R,K_R) code pair. The input vector length must be $S_R \cdot S_C$. The input shortened message bits vector is arranged into an S_R-by-S_C matrix. The shortened message matrix prepends two dimensions by padding the beginning of the message matrix with zeros. The resulting matrix is a K_R-by-K_C matrix.



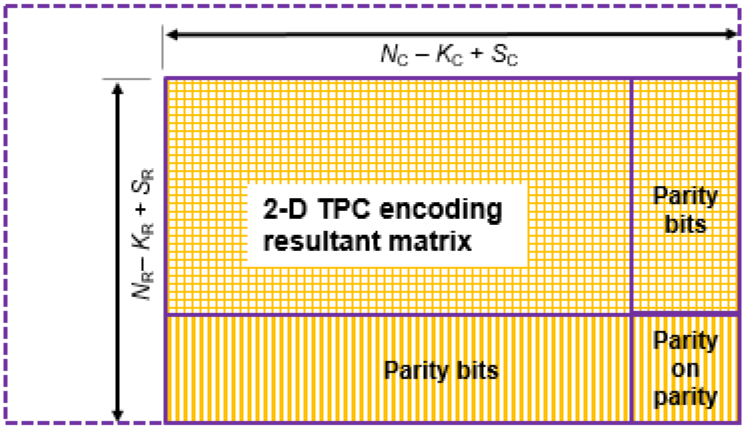
Row-wise encoding uses an (N_C,K_C) systematic linear block encoder with K_C bits per row. The row-wise encoding results in a K_R-by-N_C matrix that includes parity bits added to each row.



Next, the column-wise encoding uses an (N_R,K_R) systematic linear block encoder on each of the N_C columns.



Applying this 2-D TPC encoding to the initial K_R-by-K_C matrix and excluding the zero-padded bits from the output results in an (N_R–K_R+S_R)-by-(N_C–K_C+S_C) matrix. This matrix includes parity bits added to each row and column.



The 2-D TPC shortened-code matrix is reshaped into a column vector of length (N_R–K_R+S_R) · (N_C–K_C+S_C) and returned as the TPC-encoded output.

References

[1] Pyndiah, R. M. "Near-Optimum Decoding of Product Codes: Block Turbo Codes." *IEEE Transactions on Communications*. Vol. 46, Number 8, August 1998, pp. 1003–1010.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

BCH Encoder | TPC Decoder

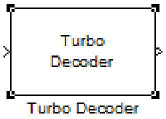
Functions

`tpcenc`

Introduced in R2018b

Turbo Decoder

Decode input signal using parallel concatenated decoding scheme



Library

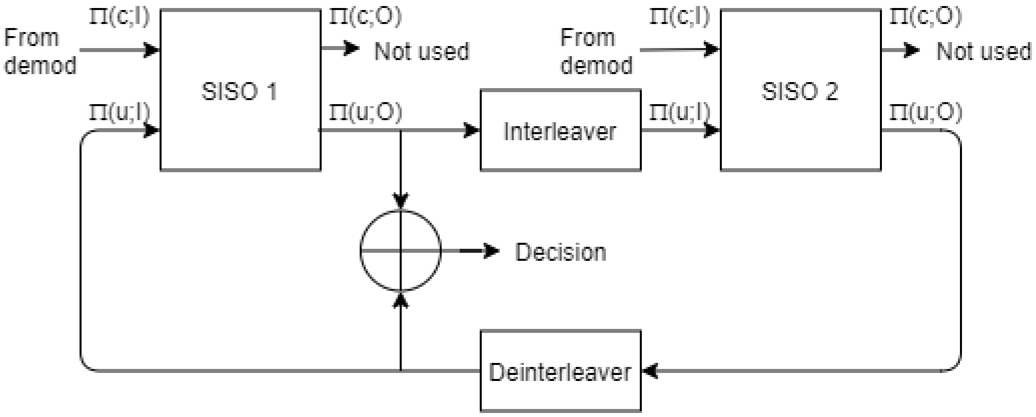
Convolutional sublibrary of Error Detection and Correction

Description

The Turbo Decoder block decodes the input signal using a parallel concatenated decoding scheme. The iterative decoding scheme uses the *a posteriori* probability (APP) decoder as the constituent decoder, an interleaver, and a deinterleaver.

The two constituent decoders use the same trellis structure and decoding algorithm.

Block Diagram of Iterative Turbo Decoding



The previous block diagram illustrates that the APP decoders (labeled as SISO modules in the previous image) output an updated sequence of log-likelihoods of the encoder input bits, π(u;O). This sequence is based on the received sequence of log-likelihoods of the channel (coded) bits, π(c;I), and code parameters.

The decoder block iteratively updates these likelihoods for a fixed number of decoding iterations and then outputs the decision bits. The interleaver (π) that the decoder uses is identical to the one the encoder uses. The deinterleaver (π^{-1}) performs the inverse permutation with respect to the interleaver. The decoder does not assume knowledge of the tail bits and excludes these bits from the iterations.

Dimensions

This block accepts an M-by-1 column vector input signal and outputs an L-by-1 column vector signal. For a given trellis, L and M are related by:

$$L = \frac{(M - 2 \cdot numTails)}{(2 \cdot n - 1)}$$

and

$$M = L \cdot (2 \cdot n - 1) + 2 \cdot numTails$$

where

M = decoder input length

L = decoder output length

n = log2(trellis.NumOutputSymbols), for a rate 1/2 trellis, n = 2

$numTails$ = log2(trellis.numStates) * n

Parameters

Trellis structure

Trellis structure of constituent convolutional code.

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Alternatively, use the poly2trellis function to create a custom trellis using the constraint length, code generator (octal), and feedback connection (octal).

The default structure is the result of poly2trellis(4, [13 15], 13).

Source of interleaver indices

Specify the source of the interleaver indices as `Property` or `Input port`.

When you set this parameter to `Property`, the block uses the **Interleaver indices** parameter to specify the interleaver indices.

When you set this parameter to `Input port`, the block uses the secondary input port, IntrInd, to specify the interleaver indices.

Interleaver indices

Specify the mapping that the Turbo encoder block uses to permute the input bits as a column vector of integers. The default is `(64:-1:1).'`. This mapping is a vector with the number of elements equal to *L*, the length of the output signal. Each element must be an integer between 1 and *L*, with no repeated values.

Decoding algorithm

Specify the decoding algorithm that the constituent APP decoders use to decode the input signal as `True APP`, `Max*`, `Max`. When you set this parameter to:

- `True APP` – the block implements true *a posteriori* probability decoding
- `Max*` or `Max` – the block uses approximations to increase the speed of the computations.

Number of scaling bits

Specify the number of bits which the constituent APP decoders must use to scale the input data to avoid losing precision during computations. The decoder multiplies the input by `2^`**Number of scaling bits** and divides the pre-output by the same factor. The value for this parameter must be a scalar integer between `0` and `8`. This parameter only applies when you set **Decoding algorithm** to `Max*`. The default is `3`.

Number of decoding iterations

Specify the number of decoding iterations the block uses. The default is `6`. The block iterates and provides updates to the log-likelihood ratios (LLR) of the uncoded output bits. The output of the block is the hard-decision output of the final LLR update.

Simulate using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

Supported Data Type

Port	Supported Data Types
In	• Double • Single
Out	• Double

Examples

For an example that uses the Turbo Encoder and Turbo Decoder blocks, see the “Parallel Concatenated Convolutional Coding: Turbo Codes” example.

Pair Block

Turbo Encoder

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima. "Near Shannon limit error correcting coding and decoding: turbo codes,” *Proceedings of the IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara. “ Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes,”*Jet Propulsion Lab TDA Progress Report*, Vol. 42–27, Nov. 1996.

[3] Schlegel, Christian B. and Lance C. Perez. *Trellis and Turbo Coding*, IEEE Press, 2004.

[4] 3GPP TS 36.212 v9.0.0, *3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

APP Decoder | General Block Deinterleaver | General Block Interleaver | Turbo Encoder

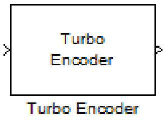
Objects

`comm.TurboDecoder`

Introduced in R2011b

Turbo Encoder

Encode binary data using parallel concatenated encoding scheme



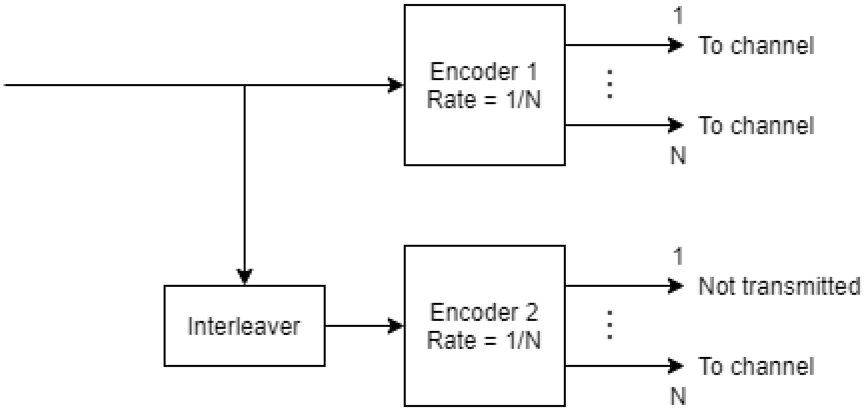
Library

Convolutional sublibrary of Error Detection and Correction

Description

The Turbo Encoder block encodes a binary input signal using a parallel concatenated coding scheme. This coding scheme employs two identical convolutional encoders and one internal interleaver. Each constituent encoder is independently terminated by tail bits.

Block Diagram of Parallel Concatenated Convolutional Code



The previous block diagram illustrates that the output of the Turbo Encoder block consists of the systematic and parity bits streams of the first encoder, and only the parity bit streams of the second encoder.

For a rate one-half constituent encoder, the block interlaces the three streams and multiplexes the tail bits to the end of the encoded data streams.

For more information about tail bits, see the terminate **Operation mode** on the Convolutional Encoder block reference page.

Dimensions

This block accepts an *L*-by-1 column vector input signal and outputs an *M*-by-1 column vector signal. For a given trellis, *M* and *L* are related by:

$$M = L \cdot (2 \cdot n - 1) + 2 \cdot numTails$$

and

$$L = \frac{(M - 2 \cdot numTails)}{(2 \cdot n - 1)}$$

where

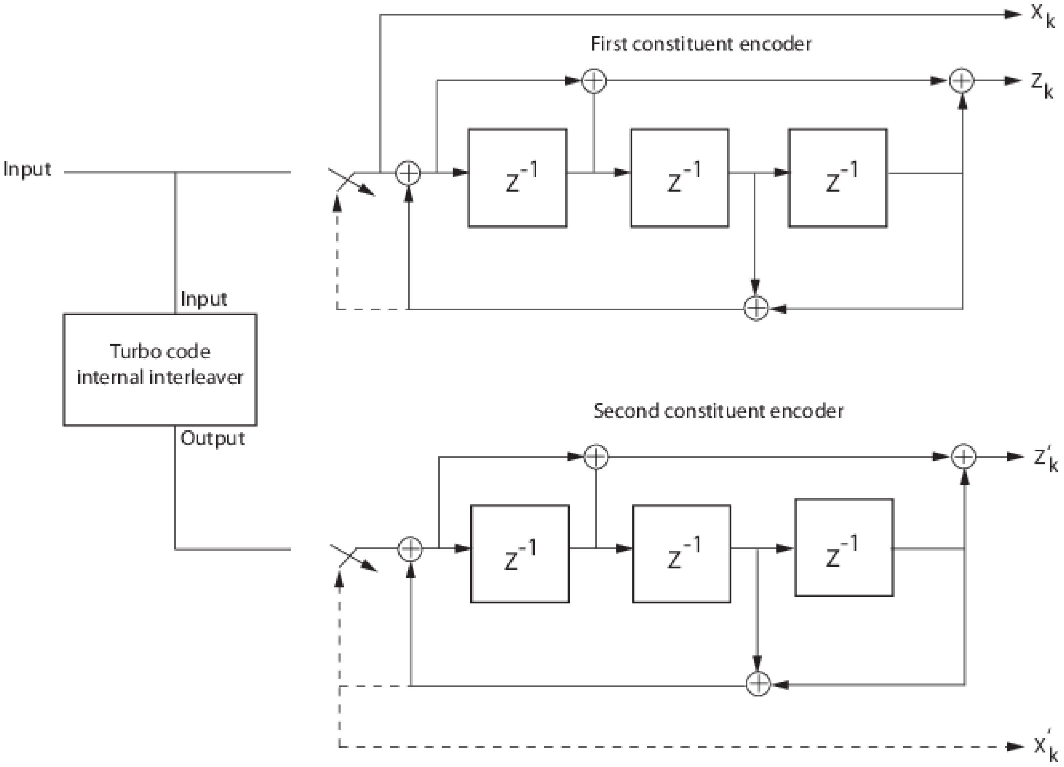
L = encoder input length

M = encoder output length

n = log2(trellis.NumOutputSymbols), for a rate 1/2 trellis, n = 2

$numTails$ = log2(trellis.numStates) * n

Encoder Schematic for Rate 1/3 Turbo Code Example



X_k
First constituent encoder
Z_k
Input
z^-1
z^-1
z^-1
Input
Turbo code internal interleaver
Output
Second constituent encoder
Z'_k
z^-1
z^-1
z^-1
X'_k

The previous schematic shows the encoder configuration for a trellis specified by the default value of the **Trellis structure** parameter, `poly2trellis(4, [13 15], 13)`. For an input vector length of 64 bits, the output of the encoder block is 204 bits. The first 192 bits correspond to the three 64 bit streams (systematic (X_k) and parity (Z_k) bit streams from the first encoder and the parity (Z'_k) bit stream of the second encoder), interlaced as per X_k, Z_k, Z'_k. The last 12 bits correspond to the tail bits from the two encoders, when the switches are in the lower position corresponding to the dashed lines. The first group of six bits (three systematic bits and three parity bits) are the output tail bits from the first constituent encoder. The second group of six bits (three systematic bits and three parity bits) are the output tail bits from the second constituent encoder.

Due to the tail bits, the encoder output code rate is slightly less than 1/3.

Parameters

Trellis structure

Trellis structure of constituent convolutional code.

Specify the trellis as a MATLAB structure that contains the trellis description of the constituent convolutional code. Alternatively, use the `poly2trellis` function to create a custom trellis using the constraint length, code generator (octal), and feedback connections (octal).

This block supports only rate 1-by-*N* trellises where *N* is an integer.

The default structure is the result of poly2trellis(4, [13 15], 13).

Source of interleaver indices

Specify the source of the interleaver indices as `Property` or `Input port`.

When you set this parameter to `Property`, the block uses the **Interleaver indices** parameter to specify the interleaver indices.

When you set this parameter to `Input port`, the block uses the secondary input port, IntrInd, to specify the interleaver indices.

Interleaver indices

Specify the mapping that the block uses to permute the input bits as a column vector of integers. The default is `(64:-1:1).'`. This mapping is a vector with the number of elements equal to the length, *L*, of the input signal. Each element must be an integer between 1 and *L*, with no repeated values.

Simulate using

Specify if the block simulates using `Code generation` or `Interpreted execution`. The default is `Interpreted execution`.

Supported Data Type

Port	Supported Data Types
In	• Double • Single • Fixed-point
Out	• Double • Single • Fixed-point

Examples

For an example that uses the Turbo Encoder and Turbo Decoder blocks, see the “Parallel Concatenated Convolutional Coding: Turbo Codes” example.

Pair Block

Turbo Decoder

References

[1] Berrou, C., A. Glavieux, and P. Thitimajshima. "Near Shannon limit error correcting coding and decoding: turbo codes,” *Proceedings of the IEEE International Conference on Communications*, Geneva, Switzerland, May 1993, pp. 1064–1070.

[2] Benedetto, S., G. Montorsi, D. Divsalar, and F. Pollara. “ Soft-Input Soft-Output Maximum A Posterior (MAP) Module to Decode Parallel and Serial Concatenated Codes,”*Jet Propulsion Lab TDA Progress Report*, Vol. 42–27, Nov. 1996.

[3] Schlegel, Christian B. and Lance C. Perez. *Trellis and Turbo Coding*, IEEE Press, 2004.

[4] 3GPP TS 36.212 v9.0.0, *3rd Generation partnership project; Technical specification group radio access network; Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (release 9),* 2009-12.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Convolutional Encoder | General Block Interleaver | Turbo Decoder

Objects

`comm.TurboEncoder`

Introduced in R2011b

Uniform Noise Generator

(To be removed) Generate uniformly distributed noise between upper and lower bounds

Note Uniform Noise Generator will be removed in a future release. Use the MATLAB Function block and `rand` function instead.

Library

Noise Generators sublibrary of Comm Sources

Description

The Uniform Noise Generator block generates uniformly distributed noise. The output data of this block is uniformly distributed between the specified lower and upper bounds. The upper bound must be greater than or equal to the lower bound.

You must specify the **Initial seed** in the simulation. When it is a constant, the resulting noise is repeatable.

If all the elements of the output vector are to be independent and identically distributed (i.i.d.), then you can use a scalar for the **Noise lower bound** and **Noise upper bound** parameters. Alternatively, you can specify the range for each element of the output vector individually, by using vectors for the **Noise lower bound** and **Noise upper bound** parameters. If the bounds are vectors, then their length must equal the length of the **Initial seed** parameter.

Attributes of Output Signal

The output signal can be a frame-based matrix, a sample-based row or column vector, or a sample-based one-dimensional array. These attributes are controlled by the **Frame-based outputs**, **Samples per frame**, and **Interpret vector parameters as 1-D** parameters.

The number of elements in the **Initial seed** parameter becomes the number of columns in a frame-based output or the number of elements in a sample-based vector output. Also, the shape (row or column) of the **Initial seed** parameter becomes the shape of a sample-based two-dimensional output signal.

Parameters

Noise lower bound, Noise upper bound

The lower and upper bounds of the interval over which noise is uniformly distributed.

Initial seed

The initial seed value for the random number generator.

Sample time

The period of each sample-based vector or each row of a frame-based matrix.

Frame-based outputs

Determines whether the output is frame-based or sample-based. This box is active only if **Interpret vector parameters as 1-D** is unchecked.

Samples per frame

The number of samples in each column of a frame-based output signal. This field is active only if **Frame-based outputs** is checked.

Interpret vector parameters as 1-D

If this box is checked, then the output is a one-dimensional signal. Otherwise, the output is a two-dimensional signal. This box is active only if **Frame-based outputs** is unchecked.

Output data type

The output can be set to `double` or `single` data types.

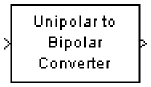
See Also

Random Source (DSP System Toolbox documentation); `rand` (built-in MATLAB function)

Introduced before R2006a

Unipolar to Bipolar Converter

Map unipolar signal in range [0, M-1] into bipolar signal



Library

Utility Blocks

Description

The Unipolar to Bipolar Converter block maps the unipolar input signal to a bipolar output signal. If the input consists of integers between 0 and M-1, where M is the **M-ary number** parameter, then the output consists of integers between -(M-1) and M-1. If M is even, then the output is odd. If M is odd, then the output is even. This block is only designed to work when the input value is within the set {0,1,2...(M-1)}, where M is the **M-ary number** parameter. If the input value is outside of this set of integers the output may not be valid.

The table below shows how the block's mapping depends on the **Polarity** parameter.

Polarity Parameter Value	Output Corresponding to Input Value of k
`Positive`	2k-(M-1)
`Negative`	-2k+(M-1)

Parameters

M-ary number

The number of symbols in the bipolar or unipolar alphabet.

Polarity

A value of `Positive` causes the block to maintain the relative ordering of symbols in the alphabets. A value of `Negative` causes the block to reverse the relative ordering of symbols in the alphabets.

Output Data Type

The type of bipolar signal produced at the block's output.

The block supports the following output data types:

- `Inherit via internal rule`
- `Same as input`
- `double`
- `int8`
- `int16`

- `int32`

When the parameter is set to its default setting, `Inherit via internal rule`, the block determines the output data type based on the input data type.

- If the input signal is floating-point (either `single` or `double`), the output data type is the same as the input data type.
- If the input data type is not floating-point:
  - Based on the **M-ary** number parameter, an ideal signed integer output word length required to contain the range [-(M-1)M-1] is computed as follows:

    ideal word length = ceil(log2(M))+1

    Note The +1 is associated with the need for the sign bit.
  - The block sets the output data type to be a signed integer, based on the smallest word length (in bits) that can fit best the computed ideal word length.

Note The selections in the “Hardware Implementation Pane” (Simulink) pertaining to word length constraints do not affect how this block determines output data types.

Examples

If the input is [0; 1; 2; 3], the **M-ary number** parameter is `4`, and the **Polarity** parameter is `Positive`, then the output is [-3; -1; 1; 3]. Changing the **Polarity** parameter to `Negative` changes the output to [3; 1; -1; -3].

If the value for the **M-ary number** is 2^7 the block gives an output of int8.

If the value for the **M-ary number** is 2^7+1 the block gives an output of int16.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

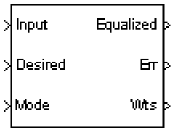
Bipolar to Unipolar Converter

Introduced before R2006a

Variable Step LMS Decision Feedback Equalizer

(To be removed) Equalize using decision feedback equalizer that updates weights with variable-step-size LMS algorithm

Note will be removed in a future release. Consider using Decision Feedback Equalizer instead.



Library

Equalizers

Description

The Variable Step LMS Decision Feedback Equalizer block uses a decision feedback equalizer and the variable-step-size LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the variable-step-size LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a *T*/*N*-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of forward taps** parameter.

The port labeled `Equalized` outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see "Equalization".

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Because the channel delay is typically unknown, a common practice is to set the reference tap to the center tap of the forward filter.

Parameters

Number of forward taps

The number of taps in the forward filter of the decision feedback equalizer.

Number of feedback taps

The number of taps in the feedback filter of the decision feedback equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of forward taps in the equalizer.

Initial step size

The step size that the variable-step-size LMS algorithm uses at the beginning of the simulation.

Increment step size

The increment by which the step size changes from iteration to iteration

Minimum step size

The smallest value that the step size can assume.

Maximum step size

The largest value that the step size can assume.

Leakage factor

The leakage factor of the variable-step-size LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that concatenates the initial weights for the forward and feedback taps.

Mode input port

When you select this check box, the block has an input port that enables you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. The equalizer will train for the length of the Desired signal. If the mode input is not present, the equalizer will train at the beginning of every frame for the length of the Desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current forward and feedback weights, concatenated into one vector.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Compatibility Considerations

Variable Step LMS Decision Feedback Equalizer will be removed

Warns starting in R2020a

- Variable Step LMS Decision Feedback Equalizer will be removed in a future release. Consider using Decision Feedback Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Decision Feedback Equalizer block is equivalent to the **Mode input port** parameter of the Variable Step LMS Decision Feedback Equalizer block.
- The Decision Feedback Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Variable Step LMS Decision Feedback Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

Topics

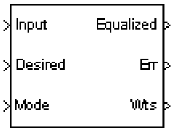
“Equalization”

Introduced before R2006a

Variable Step LMS Linear Equalizer

(To be removed) Equalize using linear equalizer that updates weights with variable-step-size LMS algorithm

Note will be removed in a future release. Consider using Linear Equalizer instead.



Library

Equalizers

Description

The Variable Step LMS Linear Equalizer block uses a linear equalizer and the variable-step-size LMS algorithm to equalize a linearly modulated baseband signal through a dispersive channel. During the simulation, the block uses the variable-step-size LMS algorithm to update the weights, once per symbol. When you set the **Number of samples per symbol** parameter to `1`, then the block implements a symbol-spaced equalizer and updates the filter weights once for each symbol. When you set the **Number of samples per symbol** parameter to a value greater than 1, the weights are updated once every N^{th} sample, for a *T*/*N*-spaced equalizer.

Input and Output Signals

The `Input` port accepts a column vector input signal. The `Desired` port receives a training sequence with a length that is less than or equal to the number of symbols in the `Input` signal. Valid training symbols are those symbols listed in the **Signal constellation** vector.

Set the **Reference tap** parameter so it is greater than zero and less than the value for the **Number of taps** parameter.

The `Equalized` port outputs the result of the equalization process.

You can configure the block to have one or more of these extra ports:

- `Mode` input.
- `Err` output for the error signal, which is the difference between the `Equalized` output and the reference signal. The reference signal consists of training symbols in training mode, and detected symbols otherwise.
- `Weights` output.

Decision-Directed Mode and Training Mode

To learn the conditions under which the equalizer operates in training or decision-directed mode, see “Equalization”.

Equalizer Delay

For proper equalization, you should set the **Reference tap** parameter so that it exceeds the delay, in symbols, between the transmitter's modulator output and the equalizer input. When this condition is satisfied, the total delay, in symbols, between the modulator output and the equalizer *output* is equal to

1+(**Reference tap**-1)/(**Number of samples per symbol**)

Since the channel delay is typically unknown, a common practice is to set the reference tap to the center tap.

Parameters

Number of taps

The number of taps in the filter of the linear equalizer.

Number of samples per symbol

The number of input samples for each symbol.

Signal constellation

A vector of complex numbers that specifies the constellation for the modulation.

Reference tap

A positive integer less than or equal to the number of taps in the equalizer.

Initial step size

The step size that the variable-step-size LMS algorithm uses at the beginning of the simulation.

Increment step size

The increment by which the step size changes from iteration to iteration

Minimum step size

The smallest value that the step size can assume.

Maximum step size

The largest value that the step size can assume.

Leakage factor

The leakage factor of the LMS algorithm, a number between 0 and 1. A value of 1 corresponds to a conventional weight update algorithm, and a value of 0 corresponds to a memoryless update algorithm.

Initial weights

A vector that lists the initial weights for the taps.

Mode input port

When you select this check box, the block has an input port that allows you to toggle between training and decision-directed mode. For training, the mode input must be 1, for decision directed, the mode should be 0. For every frame in which the mode input is 1 or not present, the equalizer trains at the beginning of the frame for the length of the desired signal.

Output error

When you select this check box, the block outputs the error signal, which is the difference between the equalized signal and the reference signal.

Output weights

When you select this check box, the block outputs the current weights.

References

[1] Farhang-Boroujeny, B., *Adaptive Filters: Theory and Applications*, Chichester, England, Wiley, 1998.

Compatibility Considerations

Variable Step LMS Linear Equalizer will be removed

Warns starting in R2020a

- Variable Step LMS Linear Equalizer will be removed in a future release. Consider using Linear Equalizer instead with the adaptive algorithm set to LMS.
- The **Enable training control input** parameter of the Linear Equalizer block is equivalent to the **Mode input port** parameter of the Variable Step LMS Linear Equalizer block.
- The Linear Equalizer block does not have a leakage factor parameter. This is equivalent to setting the **Leakage factor** parameter to 1 in the Variable Step LMS Linear Equalizer block.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

Decision Feedback Equalizer | Linear Equalizer | MLSE Equalizer

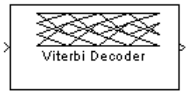
Topics

“Equalization”

Introduced before R2006a

Viterbi Decoder

Decode convolutionally encoded data using Viterbi algorithm



Library

Convolutional sublibrary of Error Detection and Correction

Description

The Viterbi Decoder block decodes input symbols to produce binary output symbols. This block can process several symbols at a time for faster performance.

This block can output sequences that vary in length during simulation. For more information about sequences that vary in length, or variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Input and Output Sizes

If the convolutional code uses an alphabet of 2^n possible symbols, this block's input vector length is L*n for some positive integer L. Similarly, if the decoded data uses an alphabet of 2^k possible output symbols, this block's output vector length is L*k.

This block accepts a column vector input signal with any positive integer value for L. For variable-sized inputs, the L can vary during simulation. The operation of the block is governed by the operation mode parameter.

For information about the data types each block port supports, see the “Supported Data Types” on page 5-828 table on this page.

Input Values and Decision Types

The entries of the input vector are either bipolar, binary, or integer data, depending on the **Decision type** parameter.

Decision type Parameter	Possible Entries in Decoder Input	Interpretation of Values	Branch metric calculation
`Unquantized`	Real numbers	Positive real: logical zero Negative real: logical one	Euclidean distance
`Hard Decision`	0, 1	0: logical zero 1: logical one	Hamming distance

Decision type Parameter	Possible Entries in Decoder Input	Interpretation of Values	Branch metric calculation
`Soft Decision`	Integers between 0 and 2^b-1, where *b* is the **Number of soft decision bits** parameter.	0: most confident decision for logical zero 2^b-1: most confident decision for logical one Other values represent less confident decisions.	Hamming distance

To illustrate the soft decision situation more explicitly, the following table lists interpretations of values for 3-bit soft decisions.

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

Operation Modes for Inputs

The Viterbi decoder block has three possible methods for transitioning between successive input frames. The **Operation mode** parameter controls which method the block uses:

- In `Continuous` mode, the block saves its internal state metric at the end of each input, for use with the next frame. Each traceback path is treated independently.
- In `Truncated` mode, the block treats each input independently. The traceback path starts at the state with the best metric and always ends in the all-zeros state. This mode is appropriate when the corresponding Convolutional Encoder block has its **Operation mode** set to `Truncated (reset every frame)`.
- In `Terminated` mode, the block treats each input independently, and the traceback path always starts and ends in the all-zeros state. This mode is appropriate when the uncoded message signal (that is, the input to the corresponding Convolutional Encoder block) has enough zeros at the end of each input to fill all memory registers of the feed-forward encoder. If the encoder has `k` input streams and constraint length vector `constr` (using the polynomial description), “enough” means `k*max(constr-1)`. For feedback encoders, this mode is appropriate if the corresponding Convolutional Encoder block has **Operation mode** set to `Terminate trellis by appending bits`.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated` or `Terminated`, the block's state resets at every input time step.

Use the `Continuous` mode when the input signal contains only one symbol.

Reset Port

The reset port is usable only when the **Operation mode** parameter is set to `Continuous`. Selecting **Enable reset input port** gives the block an additional input port, labeled `Rst`. When the `Rst` input is nonzero, the decoder returns to its initial state by configuring its internal memory as follows:

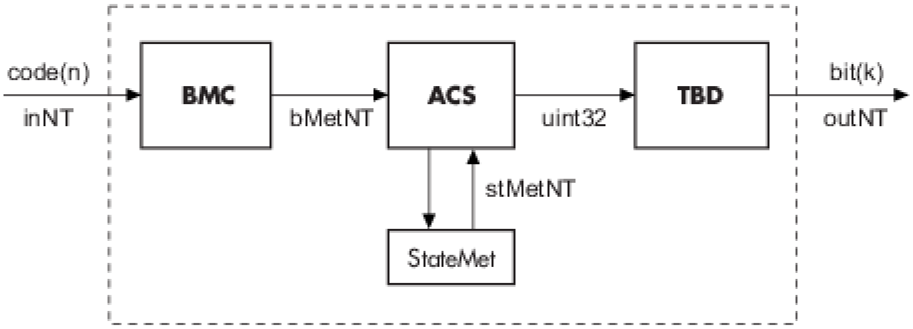
- Sets the all-zeros state metric to zero.
- Sets all other state metrics to the maximum value.
- Sets the traceback memory to zero.

Using a reset port on this block is analogous to setting **Operation mode** in the Convolutional Encoder block to `Reset on nonzero input via port`.

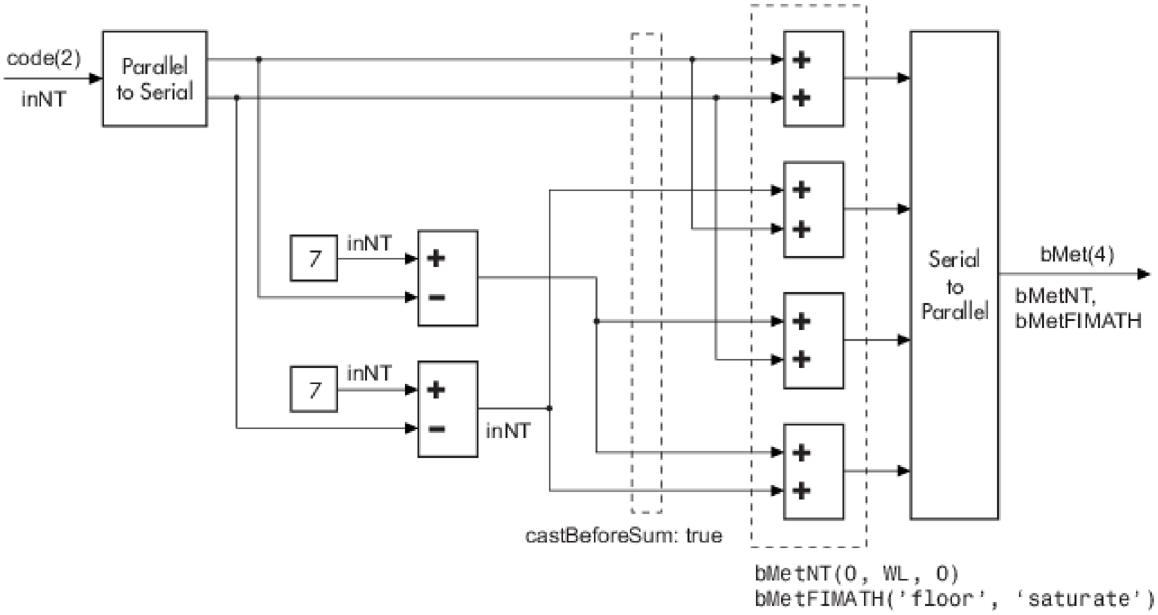
The reset port supports `double` or `boolean` typed signals.

Fixed-Point Signal Flow Diagram

There are three main components to the Viterbi decoding algorithm. They are branch metric computation (BMC), add-compare and select (ACS), and traceback decoding (TBD). The following diagram illustrates the signal flow for a *k/n* rate code.

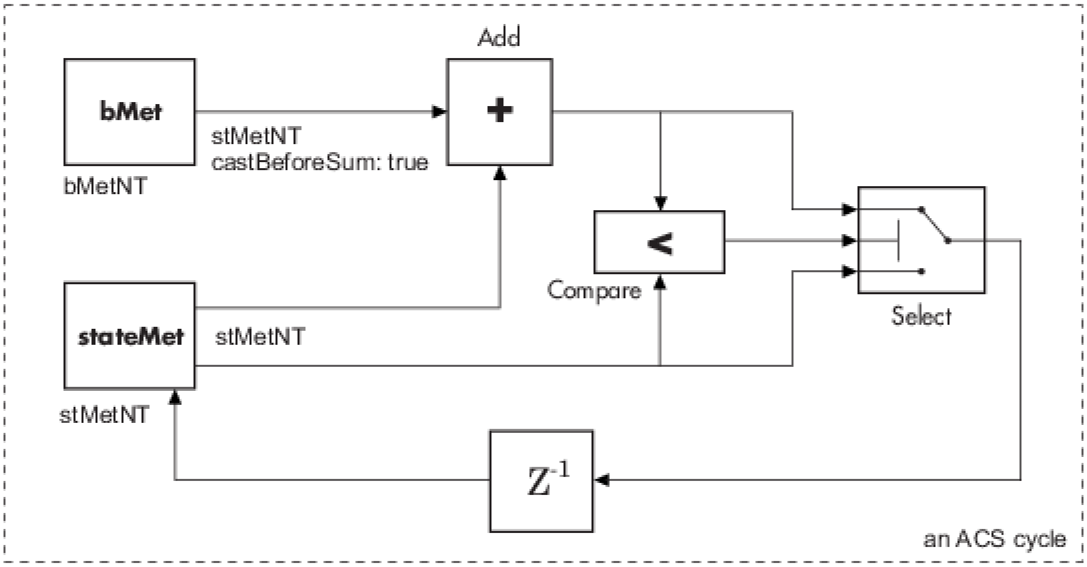


As an example of a BMC diagram, a 1/2 rate, *nsdec* = 3 signal flow would be as follows.



$$\begin{aligned} WL &= nsdec + n - 1 \\ n = 2 &\Rightarrow WL = 4 \end{aligned}$$

The ACS component is generally illustrated as shown in the following diagram.



```
stMetNT(0, WL2, 0)
stMetFIMATH('floor', 'saturate')
```

Where WL2 is specified on the mask by the user.

In the flow diagrams above, inNT, bMetNT , stMetNT, and outNT are `numerictype` (Fixed-Point Designer) objects, and bMetFIMATH and stMetFIMATH, are `fimath (Fixed-Point Designer)` objects.

Puncture Pattern Examples

For some commonly used puncture patterns for specific rates and polynomials, see the last three references.

Fixed-Point Viterbi Decoding Examples

The “Perform Fixed-Point Hard Decision Viterbi Decoding” and “Perform Fixed-Point Soft Decision Viterbi Decoding” examples configure the Viterbi decoder block to use fixed-point processing for hard- and soft-decision convolutional decoding.

The examples highlight the fixed-point modeling attributes of the Viterbi decoder, using a familiar layout.

Overview of the Simulations

The two simulations have a similar structure and have most parameters in common. A data source produces a random binary sequence that is convolutionally encoded, BPSK modulated, and passed through an AWGN channel.

The Convolutional encoder is configured as a rate 1/2 encoder. For every 2 bits, the encoder adds another 2 redundant bits. To accommodate this, and add the correct amount of noise, the **Eb/No (dB)** parameter of the AWGN block is in effect halved by subtracting 10*log10(2).

For the hard-decision case, the BPSK demodulator produces hard decisions, at the receiver, which are passed onto the decoder.

For the soft-decision case, the BPSK demodulator produces soft decisions, at the receiver, using the log-likelihood ratio. These soft outputs are 3-bit quantized and passed onto the decoder.

After the decoding, the simulation compares the received decoded symbols with the original transmitted symbols in order to compute the bit error rate. The simulation ends after processing 100 bit errors or 1e6 bits, whichever comes first.

Fixed-Point Modeling

Fixed-point modeling enables bit-true simulations which take into account hardware implementation considerations and the dynamic range of the data/parameters. For example, if the target hardware is a DSP microprocessor, some of the possible word lengths are 8, 16, or 32 bits, whereas if the target hardware is an ASIC or FPGA, there may be more flexibility in the word length selection.

To enable fixed-point Viterbi decoding, the block input must be of type ufix1 (unsigned integer of word length 1) for hard decisions. Based on this input (either a 0 or a 1), the internal branch metrics are calculated using an unsigned integer of word length = (number of output bits), as specified by the trellis structure (which equals 2 for the hard-decision example).

For soft decisions, the block input must be of type ufixN (unsigned integer of word length N), where N is the number of soft-decision bits, to enable fixed-point decoding. The block inputs must be integers in the range 0 to 2^{N-1}. The internal branch metrics are calculated using an unsigned integer of word length = (N + number of output bits - 1), as specified by the trellis structure (which equals 4 for the soft-decision example).

The **State metric word length** is specified by the user and usually must be greater than the branch metric word length already calculated. You can tune this to be the most suitable value (based on hardware and/or data considerations) by reviewing the logged data for the system.

Enable the logging by selecting **Apps > Fixed-Point Tool**. In the Fixed-Point Setting GUI, set the **Fixed-point instruments mode** to `Minimums, maximums and overflows`, and rerun the simulation. If you see overflows, it implies the data did not fit in the selected container. You could either increase the size of the word length (if your hardware allows it) or try scaling the data prior to processing it. Based on the minimum and maximum values of the data, you are also able to determine whether the selected container is of the appropriate size.

Try running simulations with different values of **State metric word length** to get an idea of its effect on the algorithm. You should be able to narrow down the parameter to a suitable value that has no adverse effect on the BER results.

Comparisons with Double-Precision Data

To run the same model with double precision data, Select **Apps > Fixed-Point Tool**. In the Fixed-Point Tool GUI, select the **Data type override** to be `Double`. This selection overrides all data type settings in all the blocks to use double precision. For the Viterbi Decoder block, as **Output type** was set to `Boolean`, this parameter should also be set to double.

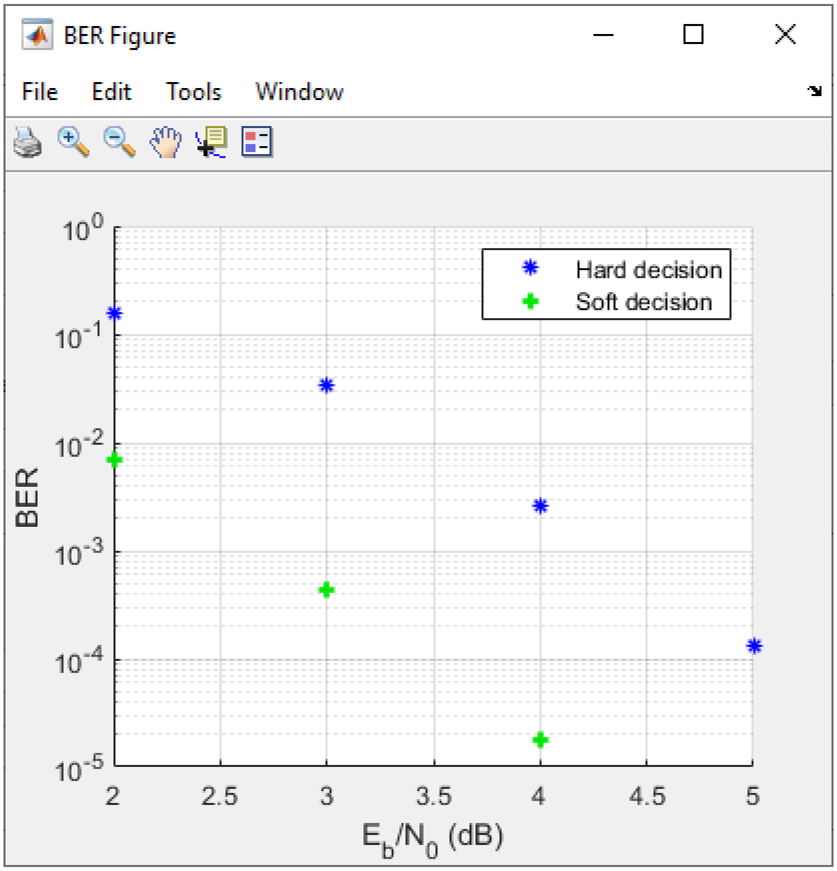
Upon simulating the model, note that the double-precision and fixed-point BER results are the same. They are the same because the fixed-point parameters for the model have been selected to avoid any loss of precision while still being most efficient.

Comparisons Between Hard and Soft-Decision Decoding

The two models are set up to run from within BERTool to generate a simulation curve that compares the BER performance for hard-decision versus soft-decision decoding.

To generate simulation results for “Perform Fixed-Point Hard Decision Viterbi Decoding”, do the following:

1. Open the “Perform Fixed-Point Hard Decision Viterbi Decoding” example to get the simulink model on path.
2. Type `bertool` at the MATLAB command prompt.
3. Go to the **Monte Carlo** pane.
4. Set the **Eb/No range** to `2:5`.
5. Set the **Simulation model** to `cm_viterbi_harddec_fixpt`. Make sure that the model is on path.
6. Set the **BER variable name** to `BER`.
7. Set the **Number of errors** to `100`, and the **Number of bits** to `1e6`.
8. Press **Run** and a plot is generated.



To generate simulation results for “Perform Fixed-Point Soft Decision Viterbi Decoding”, open the example to get simulink model on path, change the change the **Simulation model** to `cm_viterbi_softdec_fixpt`, and press **Run**.

Notice that, as expected, 3-bit soft-decision decoding is better than hard-decision decoding, roughly to the tune of 1.7 dB, and not 2 dB as commonly cited. The difference in the expected results could be attributed to the imperfect quantization of the soft outputs from the demodulator.

Parameters

Trellis structure

MATLAB structure that contains the trellis description of the convolutional encoder. Use the same value here and in the corresponding Convolutional Encoder block.

Punctured code

Select this check box to specify a punctured input code. The field, **Punctured code**, appears.

Puncture vector

Constant puncture pattern vector used at the transmitter (encoder). The puncture vector is a pattern of `1`s and `0`s. The `0`s indicate the punctured bits. When you select **Punctured code**, the **Punctured vector** field appears.

Enable erasures input port

When you check this box, the decoder opens an input port labeled `Era`. Through this port, you can specify an erasure vector pattern of `1`s and `0`s, where the `1`s indicate the erased bits.

For these erasures in the incoming data stream, the decoder does not update the branch metric. The widths and the sample times of the erasure and the input data ports must be the same. The erasure input port can be of data type `double` or `Boolean`.

Decision type

Specifies the use of `Unquantized`, `Hard Decision`, or `Soft Decision` for the branch metric calculation.

- `Unquantized` decision uses the Euclidean distance to calculate the branch metrics.
- `Soft Decision` and `Hard Decision` use the Hamming distance to calculate the branch metrics, where **Number of soft decision bits** equals `1`.

Number of soft decision bits

The number of soft decision bits to represent each input. This field is active only when **Decision type** is set to `Soft Decision`.

Error if quantized input values are out of range

Select this check box to throw an error when quantized input values are out of range. This check box is active only when **Decision type** is set to `Soft Decision` or `Hard Decision`.

Traceback depth

The number of trellis branches to construct each traceback path.

Operation mode

Method for transitioning between successive input frames: `Continuous`, `Terminated`, and `Truncated`.

Note When this block outputs sequences that vary in length during simulation and you set the **Operation mode** to `Truncated` or `Terminated`, the block's state resets at every input time step.

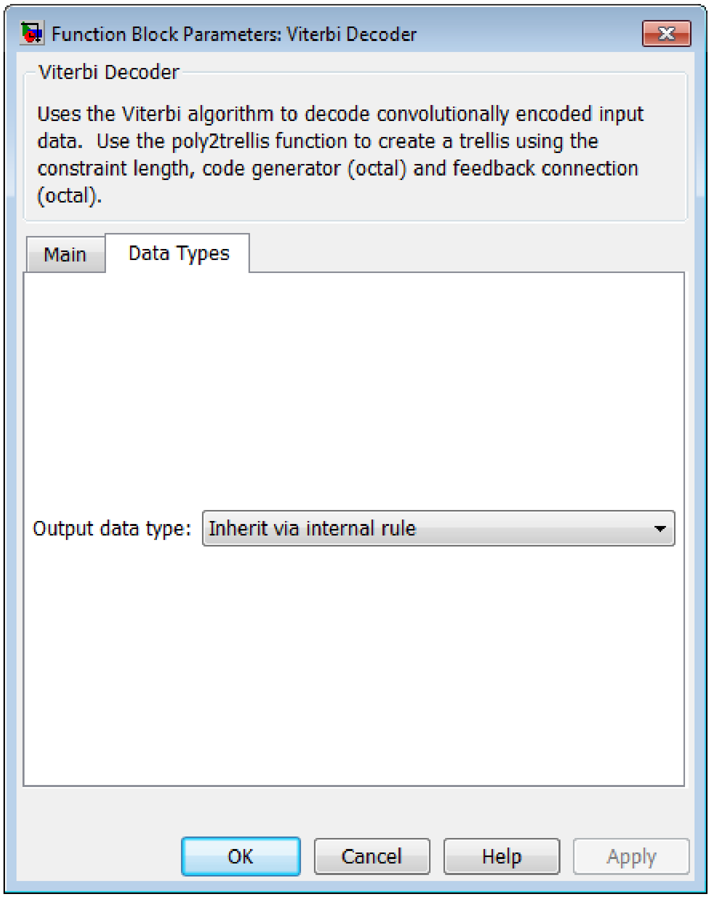
Enable reset input port

When you check this box, the decoder opens an input port labeled `Rst`. Providing a nonzero input value to this port causes the block to set its internal memory to the initial state before processing the input data.

Delay reset action to next time step

When you select this option, the Viterbi Decoder block resets after decoding the encoded data. This option is available only when you set **Operation mode** to `Continuous` and select **Enable reset input port**. You must enable this option for HDL support.

Output data type



Function Block Parameters: Viterbi Decoder

Viterbi Decoder

Uses the Viterbi algorithm to decode convolutionally encoded input data. Use the poly2trellis function to create a trellis using the constraint length, code generator (octal) and feedback connection (octal).

Output data type: Inherit via internal rule

The output signal's data type can be `double`, `single`, `boolean`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, or set to `'Inherit via internal rule'` or `'Smallest unsigned integer'`.

When set to `'Smallest unsigned integer'`, the output data type is selected based on the settings used in the **Hardware Implementation** pane of the Configuration Parameters dialog box of the model. If `ASIC/FPGA` is selected in the **Hardware Implementation** pane, the output data type is `ufix(1)`. For all other selections, it is an unsigned integer with the smallest specified wordlength corresponding to the char value (e.g., `uint8`).

When set to `'Inherit via internal rule'` (the default setting), the block selects double-typed outputs for double inputs, single-typed outputs for single inputs, and behaves similarly to the `'Smallest unsigned integer'` option for all other typed inputs.

Supported Data Types

Port	Supported Data Types
Input	• Double-precision floating point • Single-precision floating point • Boolean for `Hard decision` mode • 8-, 16-, and 32-bit signed integers (for `Hard decision` and `Soft decision` modes) • 8-, 16-, and 32-bit unsigned integers (for `Hard decision` and `Soft decision` modes) • ufix(n), where n represents the **Number of soft decision bits**
Output	• Double-precision floating point • Single-precision floating point • Boolean • 8-, 16-, and 32-bit signed integers • 8-, 16-, and 32-bit unsigned integers • ufix(1) for `ASIC/FPGA` mode

References

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

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

[3] Heller, J., and I. Jacobs. “Viterbi Decoding for Satellite and Space Communication.” *IEEE Transactions on Communication Technology* 19, no. 5 (October 1971): 835–48. https://doi.org/10.1109/TCOM.1971.1090711.

[4] Yasuda, Y., K. Kashiki, and Y. Hirata. “High-Rate Punctured Convolutional Codes for Soft Decision Viterbi Decoding.” *IEEE Transactions on Communications* 32, no. 3 (March 1984): 315–19. https://doi.org/10.1109/TCOM.1984.1096047.

[5] Haccoun, D., and G. Begin. “High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 37, no. 11 (November 1989): 1113–25. https://doi.org/10.1109/26.46505.

[6] Begin, G., D. Haccoun, and C. Paquin. “Further Results on High-Rate Punctured Convolutional Codes for Viterbi and Sequential Decoding.” *IEEE Transactions on Communications* 38, no. 11 (November 1990): 1922–28. https://doi.org/10.1109/26.61470.

[7] Moision, B. "A Truncation Depth Rule of Thumb for Convolutional Codes." *In Information Theory and Applications Workshop* (January 27 2008-February 1 2008, San Diego, California), 555-557. New York: IEEE, 2008.

More About

Traceback Depth and Decoding Delay

The traceback depth influences the decoding delay. The decoding delay is the number of zero symbols that precede the first decoded symbol in the output.

- For the continuous operating mode, the decoding delay is equal to the number of traceback depth symbols.
- For the truncated or terminated operating mode, the decoding delay is zero. In this case, the traceback depth must be less than or equal to the number of symbols in each input.

Traceback Depth Estimates

As a general estimate, a typical traceback depth value is approximately two to three times (*ConstraintLength* – 1) / (1 – *coderate*). The constraint length of the code, *ConstraintLength*, is equal to (log2(`trellis.numStates`) + 1). The *coderate* is equal to (*K* / *N*) × (length(*PuncturePattern*) / sum(*PuncturePattern*).

K is the number of input symbols, *N* is the number of output symbols, and *PuncturePattern* is the puncture pattern vector.

For example, applying this general estimate, results in these approximate traceback depths.

- A rate 1/2 code has a traceback depth of 5(*ConstraintLength* – 1).
- A rate 2/3 code has a traceback depth of 7.5(*ConstraintLength* – 1).
- A rate 3/4 code has a traceback depth of 10(*ConstraintLength* – 1).
- A rate 5/6 code has a traceback depth of 15(*ConstraintLength* – 1).

For more information, see [7].

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

HDL Code Generation

Generate Verilog and VHDL code for FPGA and ASIC designs using HDL Coder™.

HDL Coder provides additional configuration options that affect HDL implementation and synthesized logic.

Note For decoding data encoded with truncated or terminated modes, or punctured codes, use the Viterbi Decoder block from Wireless HDL Toolbox™.

HDL Coder supports the following features of the Viterbi Decoder block:

- Non-recursive encoder/decoder with feed-forward trellis and simple shift register generation configuration
- Continuous mode
- Sample-based input
- Decoder rates from 1/2 to 1/7
- Constraint length from 3 to 9

When **Decision type** is set to `Soft decision`, the HDL implementation of the Viterbi Decoder block supports fixed-point inputs and output. For input, the fixed-point data type must be `ufixN`. `N` is the number of soft-decision bits. Signed built-in data types (`int8`, `int16`, `int32`) are not supported. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types.

When **Decision type** is set to `Hard decision`, the block supports input with data types `ufix1` and `Boolean`. For output, the HDL implementation of the Viterbi Decoder block supports block-supported output data types. The HDL implementation of the Viterbi Decoder block does not support double and single input data types. The block does not support floating point output for fixed-point inputs.

The Viterbi Decoder block decodes every bit by tracing back through a traceback depth that you define for the block. The block implements a complete traceback for each decision bit, using registers to store the minimum state index and branch decision in the traceback decoding unit. There are two methods to optimize the traceback logic: a pipelined register-based implementation or a RAM-based architecture. See the “HDL Code Generation for Viterbi Decoder” example.

Register-Based Traceback

You can specify that the traceback decoding unit be pipelined to improve the speed of the generated circuit. You can add pipeline registers to the traceback unit by specifying the number of traceback stages per pipeline register.

Using the `TracebackStagesPerPipeline` implementation parameter, you can balance the circuit performance based on system requirements. A smaller parameter value indicates the requirement to add more registers to increase the speed of the traceback circuit. Increasing the parameter value results in fewer registers along with a decrease in the circuit speed.

RAM-Based Traceback

Instead of using registers, you can choose to use RAMs to save the survivor branch information. The coder does not support **Enable reset input port** when using RAM-based traceback.

1. Right-click the block and open **HDL Code > HDL Block Properties**. Set the **Architecture** property to `RAM-based Traceback`.
2. Double-click the block and set the **Traceback depth** on the Viterbi Decoder block mask.

RAM-based traceback and register-based traceback differ in the following ways:

- The RAM-based implementation traces back through one set of data to find the initial state to decode the previous set of data. The register-based implementation combines the traceback and decode operations into one step. It uses the best state found from the minimum operation as the decoding initial state.
- RAM-based implementation traces back through M samples, decodes the previous M bits in reverse order, and releases one bit in order at each clock cycle. The register-based implementation decodes one bit after a complete traceback.

Because of the differences in the two traceback algorithms, the RAM-based implementation produces different numerical results than the register-based traceback. A longer traceback depth, for example, 10 times the constraint length, is recommended in the RAM-based traceback. This depth achieves a similar bit error rate (BER) as the register-based implementation. The size of RAM required for the implementation depends on the trellis and the traceback depth.

HDL Block Properties

ConstrainedOutputPipeline	Number of registers to place at the outputs by moving existing delays within your design. Distributed pipelining does not redistribute these registers. The default is `0`. For more details, see “ConstrainedOutputPipeline” (HDL Coder).
InputPipeline	Number of input pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “InputPipeline” (HDL Coder).
OutputPipeline	Number of output pipeline stages to insert in the generated code. Distributed pipelining and constrained output pipelining can move these registers. The default is `0`. For more details, see “OutputPipeline” (HDL Coder).
TracebackStagesPerPipeline	See “Register-Based Traceback” on page 5-830.

Restrictions

- **Punctured code**: Do not select this option. Punctured code requires frame-based input, which HDL Coder does not support.
- **Decision type**: The coder does not support the `Unquantized` decision type.
- **Error if quantized input values are out of range**: The coder does not support this option.
- **Operation mode**: The coder supports only the `Continuous` mode.
- **Enable reset input port**: When you enable both **Enable reset input port** and **Delay reset action to next time step**, HDL support is provided. You must select `Continuous` operation mode, and use register-based traceback.
- You cannot use the Viterbi Decoder block inside a Resettable Synchronous Subsystem.

See Also

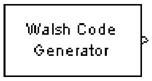
Blocks
APP Decoder | Convolutional Encoder

Topics
“HDL Code Generation for Viterbi Decoder”

Introduced before R2006a

Walsh Code Generator

Generate Walsh code from orthogonal set of codes



Library

Sequence Generators sublibrary of Comm Sources

Description

Walsh codes are defined as a set of *N* codes, denoted W_{j}, for *j* = 0, 1, ... , *N* - 1, which have the following properties:

- W_{j} takes on the values +1 and -1.
- $W_{\mathrm{j}}[0] = 1$ for all *j*.
- W_{j} has exactly *j* zero crossings, for *j* = 0, 1, ... , *N* - 1.
- $$W_j W_k^T = \begin{cases} 0 & j \neq k \\ N & j = k \end{cases}$$
- Each code W*j* is either even or odd with respect to its midpoint.

Walsh codes are defined using a Hadamard matrix of order *N*. The Walsh Code Generator block outputs a row of the Hadamard matrix specified by the **Walsh code index**, which must be an integer in the range [0, ..., *N* - 1]. If you set **Walsh code index** equal to an integer *j*, the output code has exactly *j* zero crossings, for *j* = 0, 1, ... , *N* - 1.

Note, however, that the indexing in the Walsh Code Generator block is different than the indexing in the Hadamard Code Generator block. If you set the **Walsh code index** in the Walsh Code Generator block and the **Code index parameter** in the Hadamard Code Generator block, the two blocks output different codes.

Parameters

Code length

Integer scalar that is a power of 2 specifying the length of the output code.

Code index

Integer scalar in the range [0, 1, ... , N - 1], where N is the **Code length**, specifying the number of zero crossings in the output code.

Sample time

Output sample time, specified as `-1` or a positive scalar that represents the time between each sample of the output signal. If **Sample time** is set to `-1`, the sample time is inherited from downstream. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-833.

Samples per frame

Samples per frame, specified as a positive integer indicating the number of samples per frame in one channel of the output data. If **Samples per frame** is greater than the **Code length**, the code is cyclically repeated. For information on the relationship between **Sample time** and **Samples per frame**, see “Sample Timing” on page 5-833.

Output data type

The output type of the block can be specified as an `int8` or `double`. By default, the block sets this to `double`.

More About

Sample Timing

The time between output updates is equal to the product of **Samples per frame** and **Sample time**. For example, if **Sample time** and **Samples per frame** equal one, the block outputs a sample every second. If **Samples per frame** is increased to 10, then a 10-by-1 vector is output every 10 seconds. This ensures that the equivalent output rate is not dependent on the **Samples per frame** parameter.

Compatibility Considerations

Existing models automatically update this block to current version

Behavior changed in R2020a

Starting in R2020a, Simulink no longer allows you to use the Walsh Code Generator block version available before R2015b.

Existing models automatically update to load the Walsh Code Generator block version announced in “Source blocks output frames of contiguous time samples but do not use the frame attribute” in the R2015b Release Notes. For more information on block forwarding, see “Forwarding Tables” (Simulink).

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Does not support integer only code generation.

See Also

Blocks

Hadamard Code Generator | OVSF Code Generator

Introduced before R2006a

Windowed Integrator

Integrate over time window of fixed length

Library

Comm Filters

Description

The Windowed Integrator block creates cumulative sums of the input signal values over a sliding time window of fixed length. If the **Integration period** parameter is N and the input samples are denoted by x(1), x(2), x(3),..., then the nth output sample is the sum of the x(k) values for k between n-N+1 and n. In cases where n-N+1 is less than 1, the block uses an initial condition of 0 to represent those samples.

Input and Output Signals

This block accepts scalar, column vector, and *M*-by-*N* matrix input signals. The block filters an *M*-by-*N* input matrix as follows:

- When you set the **Input processing** parameter to `Columns as channels (frame based)`, the block treats each column as a separate channel. In this mode, the block creates *N* instances of the same filter, each with its own independent state buffer. Each of the *N* filters process *M* input samples at every Simulink time step.
- When you set the **Input processing** parameter to `Elements as channels (sample based)`, the block treats each element as a separate channel. In this mode, the block creates *M***N* instances of the same filter, each with its own independent state buffer. Each filter processes one input sample at every Simulink time step.

The output dimensions always equal those of the input signal. For information about the data types each block port supports, see the “Supported Data Type” on page 5-836 table on this page.

Parameters

Integration period

The length of the interval of integration, measured in samples.

Input processing

Specify how the block processes the input signal. You can set this parameter to one of the following options:

- `Columns as channels (frame based)` — When you select this option, the block treats each column of the input as a separate channel.

- `Elements as channels (sample based)` — When you select this option, the block treats each element of the input as a separate channel.

Rounding mode

Select the rounding mode for fixed-point operations. The block uses the **Rounding mode** when the result of a fixed-point calculation does not map exactly to a number representable by the data type and scaling storing the result. The filter coefficients do not obey this parameter; they always round to `Nearest`. For more information, see “Rounding Modes” or “Rounding Mode: Simplest” (Fixed-Point Designer).

Saturate on integer overflow

Select the overflow mode for fixed-point operations. The filter coefficients do not obey this parameter; they are always saturated.

Coefficients

The block implementation uses a Direct-Form FIR filter with all tap weights set to one. The **Coefficients** parameter controls which data type represents the taps (i.e. ones) when the input data is a fixed-point signal.

Choose how you specify the word length and the fraction length of the filter coefficients (numerator and/or denominator). See “Filter Structure Diagrams” for illustrations depicting the use of the coefficient data types in this block:

- When you select `Same word length as input`, the word length of the filter coefficients match that of the input to the block. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Specify word length`, you are able to enter the word length of the coefficients, in bits. In this mode, the fraction length of the coefficients is automatically set to the binary-point only scaling that provides you with the best precision possible given the value and word length of the coefficients.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the coefficients, in bits. If applicable, you are able to enter separate fraction lengths for the numerator and denominator coefficients.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the coefficients. If applicable, you are able to enter separate slopes for the numerator and denominator coefficients. This block requires power-of-two slope and a bias of zero.
- The filter coefficients do not obey the **Rounding mode** and the **Saturate on integer overflow** parameters; they are always saturated and rounded to `Nearest`.

Product output

Use this parameter to specify how you would like to designate the product output word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the product output data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the product output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the product output. This block requires power-of-two slope and a bias of zero.

Accumulator

Use this parameter to specify how you would like to designate the accumulator word and fraction lengths. See “Filter Structure Diagrams” and “Multiplication Data Types” for illustrations depicting the use of the accumulator data type in this block:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as product output`, these characteristics match those of the product output.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the accumulator, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the accumulator. This block requires power-of-two slope and a bias of zero.

Output

Choose how you specify the output word length and fraction length:

- When you select `Same as input`, these characteristics match those of the input to the block.
- When you select `Same as accumulator`, these characteristics match those of the accumulator.
- When you select `Binary point scaling`, you are able to enter the word length and the fraction length of the output, in bits.
- When you select `Slope and bias scaling`, you are able to enter the word length, in bits, and the slope of the output. This block requires power-of-two slope and a bias of zero.

Lock scaling against changes by the autoscaling tool

Select this parameter to prevent any fixed-point scaling you specify in this block mask from being overridden by the autoscaling tool in the Fixed-Point Tool.

Supported Data Type

Port	Supported Data Types
In	• Double-precision floating point • Single-precision floating point • Signed Fixed-point
Out	• Double-precision floating point • Single-precision floating point • Signed fixed-point

Examples

If **Integration period** is `3` and the input signal is a ramp (`1`, `2`, `3`, `4`,...), then some of the sums that form the output of this block are as follows:

- 0+0+1 = 1
- 0+1+2 = 3
- 1+2+3 = 6

- 2+3+4 = 9
- 3+4+5 = 12
- 4+5+6 = 15
- etc.

The zeros in the first few sums represent initial conditions. With the **Input processing** parameter set to `Elements as channels`, then the values 1, 3, 6,... are successive values of the scalar output signal. With the **Input processing** parameter set to `Columns as channels`, the values 1, 3, 6,... are organized into output frames that have the same vector length as the input signal.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

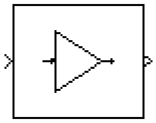
Discrete-Time Integrator | Integrate and Dump

Functions

Introduced before R2006a

Amplifier

Complex baseband model of amplifier with noise and nonlinearities
Library: RF Blockset / Idealized Baseband



Description

The Amplifier block generates a complex baseband model of an amplifier with thermal noise. This block provides four nonlinearity models and three options to specify noise representation.

Note This block assumes a nominal impedance of 1 ohm.

Ports

Input

`Port_1` — Input baseband signal
real scalar | real column | complex scalar | complex column

Input baseband signal, specified as a real scalar, real column, complex scalar, or complex column.

Data Types: `double` | `single`

Output

`Port_1` — Output baseband signal
real scalar | real column | complex scalar | complex column

Output baseband signal, specified as a real scalar, real column, complex scalar, or complex column. The output port mimics the properties of the input port. For example, if the input baseband signal is specified as a real scalar with a data type double, then the output baseband signal is also specified as a real signal with the data type double.

Data Types: `double` | `single`

Parameters

Main Tab

`Model` — Amplifier nonlinearity model
`Cubic polynomial` (default) | `AM/AM - AM/PM` | `Modified Rapp` | `Saleh`

Specify the amplifier nonlinearity model as one of the following:

- `Cubic polynomial`
- `AM/AM - AM/PM`
- `Modified Rapp`
- `Saleh`

`Linear power gain (dB)` — Linear gain of amplifier

`0` (default) | real scalar

Linear gain, specified as a scalar in dB.

`Type of Non-Linearity` — Third - order nonlinearity type

`IIP3` (default) | `OIP3` | `IP1dB` | `OP1dB` | `IPsat` | `OPsat`

Third order nonlinearity type, specified as `IIP3`, `OIP3`, `IP1dB`, `OP1dB`, `IPsat`, or `OPsat`.

IIP3 — Input third-order intercept point

`Inf` (default) | real positive number

Input third-order intercept point, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `IIP3`.

OIP3 — Output third-order intercept point

`Inf` (default) | real positive number

Output third-order intercept point, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `OIP3`.

IP1dB — Input 1 dB compression point

`Inf` (default) | real positive number

Input 1 dB compression point, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `IP1dB`.

OP1dB — Output 1 dB compression point

`Inf` (default) | real positive number

Output 1 dB compression point, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `OP1dB`.

`IPsat` — Input saturation point

`Inf` (default) | real positive number

Input saturation point, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `IPsat`.

`OPsat` — Output saturation point

`Inf` (default) | real positive number

Output saturation point, specified as a positive real number in dBm.

Dependencies

To enable this parameter, set **Model** to `Cubic polynomial` and **Type of Non-Linearity** to `OPsat`.

`Simulate using` — Specify type of simulation to run

`Code generation` (default) | `Interpreted execution`

- `Code generation` – Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is faster than `Interpreted execution`.
- `Interpreted execution` – Simulate model using the MATLAB interpreter. This option shortens startup time speed, but the speed of the subsequent simulations is slower than `Code generation`. In this mode, you can debug the source code of the block.

`Plot power characteristics` — Plot power characteristics

button (default)

This button plots the power characteristics based on the parameters specified on the **Main** tab.

`Lookup table (Pin(dBm), Pout(dBm), deg)` — Stub termination

`[ -25, 5, -1; -10, 20, -2; 0, 27, 5; 5, 28, 12 ]` (default) | real vector

Table lookup entries specified as a real *M*-by-3 matrix. This table expresses the model output power dBm level in matrix column 2 and the model phase change in degrees in matrix column 3 as related to the absolute value of the input signal power of matrix column 1 for the AM/AM - AM/PM model. The column 1 input power must increase monotonically.

Dependencies

To enable this parameter, set **Model** to `AM/AM - AM/PM`

`Output saturation level (V)` — Output saturation level

`1` (default) | real positive number

Voltage output saturation level, specified as a real positive number in dBm.

Dependencies

To enable this parameter, set **Model** to `Modified Rapp`

`Magnitude smoothness factor` — Magnitude smoothness factor

`2` (default) | real positive number

Magnitude smoothness factor for the `Modified Rapp` amplifier model AM/AM calculations, specified as a positive real number.

Dependencies

To enable this parameter, set **Model** to `Modified Rapp`

`Phase gain (rad)` — Phase gain

`-0.45` (default) | real scalar

Phase gain for the `Modified Rapp` amplifier model AM/PM calculations, specified as a real scalar in radians.

Dependencies

To enable this parameter, set **Model** to `Modified Rapp`

`Phase saturation` — Phase saturation

`.88` (default) | real positive number

Phase saturation for the `Modified Rapp` amplifier model AM/PM calculations, specified as a positive real number.

Dependencies

To enable this parameter, set **Model** to `Modified Rapp`

`Phase smoothness factor` — Phase smoothness factor

`3.43` (default) | real positive number

Phase smoothness factor for the `Modified Rapp` amplifier model AM/PM calculations, specified as a positive real number.

Dependencies

To enable this parameter, set **Model** to `Modified Rapp`

`Input scaling (dB)` — Scaling factor for input signal level

`0` (default) | nonnegative real number

Scaling factor for input signal level for the `Saleh` amplifier model, specified as a nonnegative real number in dB.

Dependencies

To enable this parameter, set **Model** to `Saleh`

`AM / AM parameters [alpha beta]` — AM/AM conversion parameters

`[ 2.1587, 1.1517 ]` (default) | two-element vector

AM/AM two-tuple conversion parameters for `Saleh` amplifier model, specified as a two-element vector of nonnegative real numbers.

Dependencies

To enable this parameter, set **Model** to `Saleh`

`AM / PM parameters [alpha beta]` — AM/PM conversion parameters

`[ 4.0033, 9.1040 ]` (default) | two-element vector

AM/PM two-tuple conversion parameters for Saleh amplifier model, specified as a two-element vector of nonnegative real numbers.

Dependencies

To enable this parameter, set **Model** to Saleh

Output scaling (dB) — Scaling factor for output signal level

0 (default) | nonnegative real number

Scaling factor for output signal level for Saleh amplifier model, specified as nonnegative real number in dB.

Dependencies

To enable this parameter, set **Model** to Saleh

Noise Tab

Include Noise — Add noise to system

off (default) | on

Select this parameter to add system noise to the input signal. Once you select this parameter, the parameters associated with the **Noise** tab are displayed.

Specify noise type — Noise representation

Noise temperature (default) | Noise figure | Noise factor

Noise descriptive type, specified as Noise temperature, Noise figure, or Noise factor.

Noise temperature (K) — Noise temperature to model noises in amplifier

290 (default) | nonnegative real number

Noise temperature to model noise in the amplifier, specified as a nonnegative real number in degrees (K).

Dependencies

To enable this parameter, select **Include noise** and set **Specify noise type** to Noise temperature.

Noise figure (dB) — Noise figure to model noise in amplifier

10 * log10(2) (default) | nonnegative real number

Noise figure to model noise in the amplifier, specified as a nonnegative real number in dB.

Dependencies

To enable this parameter, select **Include noise** and set **Specify noise type** to Noise figure.

Noise factor — Noise factor to model noise in amplifier

2 (default) | positive integer scalar greater than or equal to 1

Noise factor to model noise in the amplifier, specified as a positive integer scalar greater than or equal to 1

Dependencies

To enable this parameter, select **Include noise** and set **Specify noise type** to Noise factor.

Seed source — Source of initial seed

Auto (default) | User specified

Source of initial seed used to prepare the Gaussian random number noise generator, specified as one of the following:

- Auto - When **Seed source** is set to Auto, seeds for each amplifier instance are generated using a random number generator. The reset method of the instance has no effect.
- User specified - When **Seed source** is set to User specified, the value provided in the **Seed** is used to initialize the random number generator and the reset method resets the random number generator using the **Seed** property value.

Seed — Seed for random number generator

67987 (default) | nonnegative integer

Seed for the random number generator, specified as a nonnegative integer less than 2^{32}. Use this value to initialize the random number generator.

Dependencies

To enable this parameter, click **Include noise** check box and choose User specified in the **Seed source** parameter.

Algorithms

Model

Cubic Polynomial

- The Cubic polynomial model uses linear power gain to determine the linear coefficient of a third-order polynomial and either IP3, P1dB, or Psat to determine the third - order coefficient of the polynomial.
- The general form of cubic nonlinearity models the AM/AM characteristics as

  $$F_{AM/AM}(u) = c_1 \times u + \frac{3}{4}c_3 \times u^3$$

  where $F_{AM/AM}(u)$ is the magnitude of the output signal, u is the magnitude of the input signal, c_1 is the coefficient of the linear gain term, and c_3 is the coefficient of the cubic gain term.

- The results for IIP3, OIP3, PI,1dB, and Po,1dB are taken from [5] (RF Blockset).
  - If “IIP3” (RF Blockset) is specified, then

    $$c_3 = -\frac{4c_1}{3 \times 10^{[(IIP3-30)/10]}}$$

    where IIP3 is given in dBm.
  - If “OIP3” (RF Blockset) is specified, then

    $$c_3 = -\frac{4c_1^3}{3 \times 10^{[(OIP3-30)/10]}}$$

    where OIP3 is given in dBm.

- If the input 1 dB gain compression power “IP1dB” (RF Blockset) is specified, then

$$c_3 = -\frac{2c_1(10^{19/20} - 10)}{15 \times 10^{[(IP1dB - 30)/10]}}$$

  where IP1dB is given in dBm.
- If the output 1 dB gain compression power “OP1dB” (RF Blockset) is specified, then

$$c_3 = -\frac{2c_1(10^{19/20} - 10)}{15 \times 10^{[(OP1dB - 30 - LGdB + 1)/10]}}$$

  where OP1dB is given in dBm, and LGdB is the linear gain in dB
- If the input saturation power “IPsat” (RF Blockset) is specified, then

$$c_3 = -\frac{4c_1}{9 \times 10^{[(IPsat - 30)/10]}}$$

  where IPsat is given in dBm.
- If the output saturation power “OPsat” (RF Blockset) is specified, then

$$c_3 = -\frac{16c_1^3}{81 \times 10^{[(OPsat - 30)/10]}}$$

  where OPsat is given in dBm.

AM/AM-AM/PM

The `AM/AM-AM/PM` model uses a lookup table to specify the amplifier power characteristics. The table returns interpolated or extrapolated values using linear interpolation. Each row in the table expresses the relationship between output power or phase change as a function of input power.

$$u_{\mathrm{out}} = Table_{AM/AM}(u) \times e^{((Table_{AM/PM}(u) + \angle u) \times i)}$$

where u_{out} is the output signal and u is the magnitude of input signal.

Saleh

- The `Saleh` model is based on normalized transfer function. Use the input / output scaling parameters to adjust signal levels from their normalized values.
- For `Saleh`, the AM/AM parameters $alpha_{AM/AM}$ and $beta_{AM/AM}$ are used to compute the amplitude gain for an input signal using the following equation:

$$F_{AM/AM}(u) = \frac{alpha_{AM/AM} \times |u|}{1 + beta_{AM/AM} \times |u|^2}$$

  where $|u|$ is the magnitude of the scaled signal and `u` is calculated as:

$$u = InputScale \times u_{\mathrm{in}}$$

- For `Saleh`, the AM/PM parameters $alpha_{AM/PM}$ and $beta_{AM/PM}$ are used to compute the phase change for an input signal using the following equation:

$$F_{AM/PM}(u) = \frac{alpha_{AM/PM} \times |u|^2}{1 + beta_{AM/PM} \times |u|^2} + angle(u)$$

where $|u|$ is the magnitude of the scaled signal and *angle* is a MATLAB function that returns the phase angle of *u*.

- The scaled output signal, u_{out} is calculated as:

$$u_{\mathrm{out}} = F_{AM/AM} \times e^{(F_{AM/PM} \times i)} \times OutputScale$$

Modified Rapp

- The `Modified Rapp` model is based on normalized transfer functions. Use the input and output scaling parameters to adjust the signal levels from their normalized values.

  The AM/AM characteristics for `Modified Rapp` are given by:

$$F_{AM/AM}(u) = \frac{g_{lin} \times |u|}{(1 + \left|\frac{g_{lin} \times u}{V_{sat}}\right|^{2p})^{1/2p}}$$

  where $|u|$ is the magnitude of input signal,

  g_{lin} is $10^{(\text{Linear Gain (dB)/20})}$, and is the amplitude gain of the amplifier, V_{sat} is “Output saturation level (V)” (RF Blockset), and p is “Magnitude smoothness factor” (RF Blockset).
- The AM/PM characteristics for `Modified Rapp` is given by

$$F_{AM/PM}(u) = \frac{A \times u^q}{[1 + (\frac{u}{B})^q]} + angle(u)$$

  where u is the magnitude of input signal, A is the “Phase gain (rad)” (RF Blockset), B is “Phase saturation” (RF Blockset), q is “Phase smoothness factor” (RF Blockset) and *angle* is a MATLAB function which returns phase angle of `u`.
- The output signal u_{out} is calculated as:

$$u_{\mathrm{out}} = u_{\mathrm{mag}} \times e^{(u_{angle} \times i)}$$

  where u_{mag} is the magnitude and u_{angle} is the phase of the output signal, respectively.

Thermal Noise Simulation

According to the “Specify noise type” (RF Blockset) parameter, you can specify the amount of thermal noise in three ways,

- `Noise temperature` — Specifies the noise in kelvin.
- `Noise factor` — Specifies the noise by using the equation:

$$Noise\ factor\ =\ 1\ +\ \frac{Noise\ temperature}{290}$$

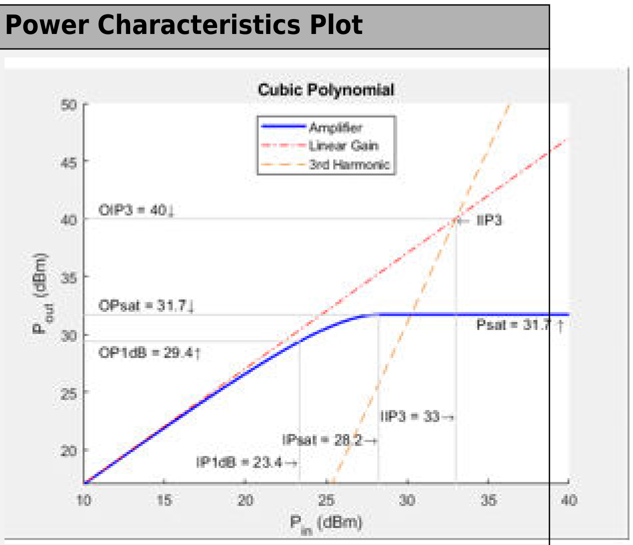
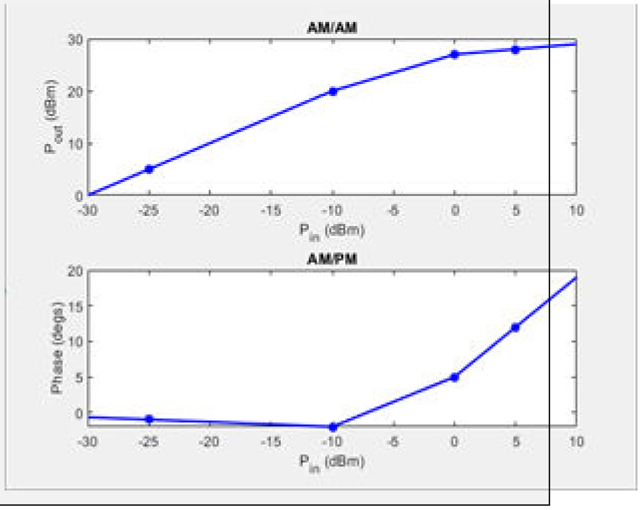
- `Noise figure` — Specifies the noise in decibels relative to a noise temperature of 290 kelvin. In terms of noise factor

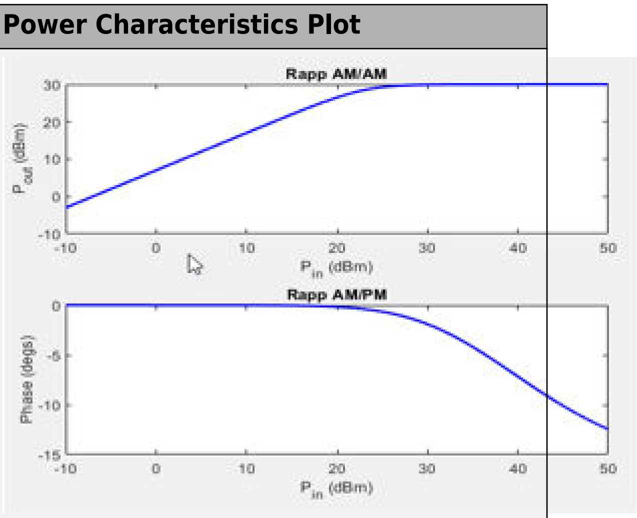
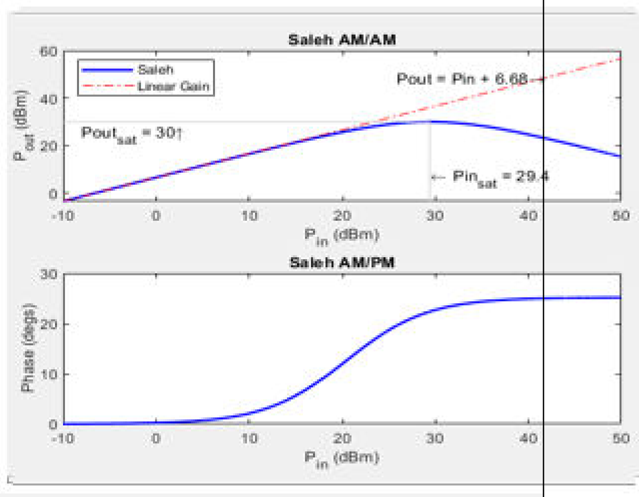
$$\mathit{Noise\ figure} \;=\; 10 \times \log(\mathit{Noise\ factor})$$

Note Some RF Blockset™ blocks require the sample time to perform baseband modeling calculations. To ensure accuracy in these calculations, the Input Port block, as well as the mathematical RF blocks compare the input sample time to the sample time you provide in the mask. If these times do not match, or if the input sample time is missing because the blocks are not connected, an error message appears.

Plot power characteristics

To visualize the functionality of **Plot power characteristics** button, set the to the Amplifier block listed in the table.

Model	Parameters	Power Characteristics Plot
Cubic Polynomial	**Main** tab: • **Linear power gain (dB)**: `7` • **Type of Non-linearity**: `IIP3` • **IIP3 (dBm)**: `33` • **Simulate using**: `Code generation` **Noise** tab: • **Specify noise type**: `Noise temperature` • **Noise temperature**: `290` • **Seed source**: `Auto`	
AM/AM - AM/PM	**Main** tab: • **Lookup table (Pin(dBm), Pout(dBm), deg)**: `[-25, 5, -1; -10, 20, -2; 0, 27, 5; 5, 28, 12]` • **Simulate using**: `Code generation` **Noise** tab: • **Specify noise type**: `Noise temperature` • **Noise temperature**: `290` • **Seed source**: `User specified` • **Seed** : `67987`	

Model	Parameters	Power Characteristics Plot
Modified Rapp	**Main** tab: • **Linear power gain (dB)**: `7` • **Output saturation level (V)**: `1` • **Magnitude smoothness factor**: `2` • **Phase gain (rad)**: `-.45` • **Phase saturation**: `0.88` • **Phase smoothness factor**: `3.43` • **Simulate using**: `Code generation` **Noise** tab: • **Specify noise type**: `Noise temperature` • **Noise temperature**: `290` • **Seed source**: `User specified` • **Seed** : `67987`	
Saleh	**Main** tab: • **Input scaling (dB)**: `0` • **AM/AM parameters [alpha beta]**: `[ 2.1587, 1.1517 ]` • **AM/PM parameters [alpha beta]**: `[ 4.0033, 9.1040 ]` • **Output scaling (dB)**: `0` • **Simulate using**: `Interpreted execution` **Noise** tab: • **Specify noise type**: `Noise figure` • **Noise figure**: `10 * log10( 2 )` • **Seed source**: `Auto`	

Application of Nonlinearity

All four subsystems for the amplifier nonlinearity models apply a memoryless nonlinearity to the complex baseband input signal. Each model

1. Multiplies the signal by a gain factor.
2. Splits the complex signal into its magnitude and angle components.
3. Applies an AM/AM conversion to the magnitude of the signal, according to the selected nonlinearity model, to produce the magnitude of the output signal.
4. Applies an AM/PM conversion to the phase of the signal, according to the selected nonlinearity model, and adds the result to the angle of the signal to produce the angle of the output signal.

References

[1] Razavi, Behzad. “Basic Concepts “ in *RF Microelectronics*, 2nd edition, Prentice Hall, 2012.

[2] Rapp, C., “Effects of HPA-Nonlinearity on a 4-DPSK/OFDM-Signal for a Digital Sound Broadcasting System.” *Proceedings of the Second European Conference on Satellite Communications*, Liege, Belgium, Oct. 22-24, 1991, pp. 179-184.

[3] Saleh, A.A.M., “Frequency-independent and frequency-dependent nonlinear models of TWT amplifiers.” *IEEE Trans. Communications*, vol. COM-29, pp.1715-1720, November 1981.

[4] IEEE 802.11-09/0296r16. “TGad Evaluation Methodology.“ Institute of Electrical and Electronics Engineers.https://www.ieee.org/

[5] Kundert, Ken.“ Accurate and Rapid Measurement of IP_2 and IP_3,“ *The Designer Guide Community*, May 22, 2002.

See Also

Introduced in R2020a